

Kafka Topic and Schema Registry



- How do you create a topic in Kafka using the Confluent CLI?
- Explain the role of the Schema Registry in Kafka. kindly try to answer given questions in details along with example in Java



Creating a Topic in Kafka Using Confluent CLI

To create a topic in Kafka using the **Confluent CLI**, follow these steps:

Prerequisites:

1. Confluent Platform is installed and running.
2. Ensure the Kafka broker is up and running.

Command to Create a Topic:

The syntax for creating a Kafka topic is:

```
bash

confluent kafka topic create <topic_name> --partitions <num_partitions> --config <key=value>
```

Example:

```
bash
```

```
confluent kafka topic create my-topic --partitions 3 --config retention.ms=604800000
```

- ``my-topic``: The name of the topic being created.
- ``--partitions 3``: Specifies the number of partitions for the topic.
- ``--config retention.ms=604800000``: Sets the retention time for the topic to 7 days (in milliseconds).

Verify the Topic:

To list all topics and verify that the topic has been created:

```
bash
```

```
confluent kafka topic list
```

Additional Notes:

- You can also set replication factors using configuration options. Ensure the replication factor doesn't exceed the number of Kafka brokers in your cluster.

Role of Schema Registry in Kafka

The **Schema Registry** is a component in Kafka that manages schemas for data serialization and deserialization. It provides a centralized repository for storing schemas and ensures schema compatibility, making Kafka a more robust and interoperable messaging system.

Key Features of Schema Registry:

1. **Schema Management**: Stores schemas for Avro, Protobuf, or JSON data formats.
2. **Schema Evolution**: Manages schema versions and ensures compatibility (e.g., backward, forward).
3. **Serialization/Deserialization**: Helps producers serialize data before sending and consumers deserialize it after receiving.
4. **Validation**: Ensures that data conforms to the registered schema.

Benefits:

- Reduces data corruption risks.
- Enables interoperability between producers and consumers.
- Facilitates schema evolution without breaking existing services.

Example with Java:

Let's use **Avro** and **Schema Registry** in a Kafka application.

1. Maven Dependencies: Add the following dependencies:

```
xml

<dependency>
  <groupId>io.confluent</groupId>
  <artifactId>kafka-avro-serializer</artifactId>
  <version>7.5.0</version>
</dependency>
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>3.6.0</version>
</dependency>
```

2. Producer Code:

```
java

import io.confluent.kafka.serializers.KafkaAvroSerializer;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;

import java.util.Properties;

public class KafkaAvroProducer {
  public static void main(String[] args) {
    String topic = "avro-topic";
    String schemaRegistryUrl = "http://localhost:8081";

    // Configure Kafka Producer
```

```

Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer", KafkaAvroSerializer.class.getName());
props.put("value.serializer", KafkaAvroSerializer.class.getName());
props.put("schema.registry.url", schemaRegistryUrl);

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

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

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

// Send data
producer.send(new ProducerRecord<>(topic, "user1", avroRecord));
producer.close();
}

```

3. Consumer Code:

```

java

import io.confluent.kafka.serializers.KafkaAvroDeserializer;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

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

public class KafkaAvroConsumer {
    public static void main(String[] args) {
        String topic = "avro-topic";
        String schemaRegistryUrl = "http://localhost:8081";

        // Configure Kafka Consumer
        Properties props = new Properties();

```

```

props.put("bootstrap.servers", "localhost:9092");
props.put("key.deserializer", KafkaAvroDeserializer.class.getName());
props.put("value.deserializer", KafkaAvroDeserializer.class.getName());
props.put("schema.registry.url", schemaRegistryUrl);
props.put("group.id", "avro-group");

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

// Poll for data
while (true) {
    ConsumerRecords<String, GenericRecord> records = consumer.poll(1000);
    records.forEach(record -> {
        System.out.println("Key: " + record.key() + ", Value: " + record.value());
    });
}
}
}

```

In this example:

- **Schema Registry** ensures that the Avro schema used by the producer and consumer is validated and compatible.
- Any schema changes must adhere to the compatibility rules defined in the Schema Registry.



- How do you register a new schema in the Schema Registry?
- What is the importance of key-value messages in Kafka? kindly try to answer given questions in details along with example in Java



1. How Do You Register a New Schema in the Schema Registry?

A schema can be registered with Kafka's **Schema Registry** using its REST API or programmatically in Java. When you register a schema, it is associated with a specific subject in the Schema Registry, typically corresponding to a Kafka topic.

Using REST API to Register a Schema

You can use the following `curl` command to register a new schema:

```
bash
```

```
curl -X POST \
-H "Content-Type: application/vnd.schemaregistry.v1+json" \
--data '{"schema": "{\\"type\\":\\"record\\",\\"name\\":\\"User\\",\\"fields\\":[{\\"name\\":\\"name\\",\\"type\\":\\"string\\"},{\\"name\\":\\"age\\",\\"type\\":\\"int\\"}]}"}' \
http://localhost:8081/subjects/my-topic-value/versions
```

- **Subject:** In this case, `my-topic-value` represents the subject associated with the value of the Kafka topic `my-topic`.
- **Schema:** The Avro schema is provided as JSON.

Example: Registering a Schema in Java

To programmatically register a schema, you can use the **Schema Registry Client** provided by Confluent.

1. **Maven Dependency:** Add the following dependency:

```
xml

<dependency>
  <groupId>io.confluent</groupId>
  <artifactId>kafka-schema-registry-client</artifactId>
  <version>7.5.0</version>
</dependency>
```

2. **Java Code:**

```
java

import io.confluent.kafka.schemaregistry.client.SchemaRegistryClient;
import io.confluent.kafka.schemaregistry.client.CachedSchemaRegistryClient;
import io.confluent.kafka.schemaregistry.client.rest.exceptions.RestClientException;
import org.apache.avro.Schema;

import java.io.IOException;

public class SchemaRegistryExample {
    public static void main(String[] args) {
```

```
String schemaRegistryUrl = "http://localhost:8081";
SchemaRegistryClient schemaRegistryClient = new CachedSchemaRegistryClient(schemaRegistryUrl, 100);

// Define Avro schema
String userSchema = "{"
    + "\"type\": \"record\", \"
    + "\"name\": \"User\", \"
    + "\"fields\": [\"
    + \"{ \"name\": \"name\", \"type\": \"string\" }, \"
    + \"{ \"name\": \"age\", \"type\": \"int\" }\"
    + "\]";

try {
    // Parse and register schema
    Schema schema = new Schema.Parser().parse(userSchema);
    String subject = "my-topic-value";
    int schemaId = schemaRegistryClient.register(subject, schema);
    System.out.println("Schema registered with ID: " + schemaId);
} catch (IOException | RestClientException e) {
    e.printStackTrace();
}
}
```

- `CachedSchemaRegistryClient`: Manages schema registration and caching.
- `register`: Registers the schema and associates it with a subject.

2. Importance of Key-Value Messages in Kafka

In Kafka, each message consists of a **key** and a **value**. Both play a critical role in organizing, partitioning, and retrieving data efficiently.

Key-Value Concept:

- **Key**: Used to determine the partition within a Kafka topic where the message will be stored.
- **Value**: Represents the actual data payload being sent in the message.

Importance:

1. Partitioning:

- The key determines the partition where the message is stored using Kafka's partitioning strategy.

- Ensures that all messages with the same key are sent to the same partition, preserving ordering.
- Example: In an e-commerce system, using the `userId` as the key ensures that all messages related to a specific user are stored in the same partition.

2. Efficient Data Retrieval:

- Consumers can process messages with the same key in a specific order.
- Helps in use cases like sessionization, where you need to group and process related events together.

3. Message Routing:

- Producers can use keys to route messages based on business logic.
- Example: Route high-priority transactions to a specific partition for faster processing.

4. Analytics and Aggregations:

- Grouping messages by key is crucial for analytics and aggregation operations in Kafka Streams.

5. Default Behavior:

- If no key is provided, messages are distributed randomly across partitions, which may not guarantee order for specific entities.

Example in Java: Using Key-Value Messages

1. Producer Code:

```
java

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;

import java.util.Properties;

public class KeyValueProducer {
    public static void main(String[] args) {
        String topic = "key-value-topic";

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

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

        // Produce key-value messages
        for (int i = 0; i < 10; i++) {
```



```

        String key = "user" + (i % 3); // Users are grouped by key
        String value = "message-" + i;
        producer.send(new ProducerRecord<>(topic, key, value));
        System.out.println("Sent: Key=" + key + ", Value=" + value);
    }
    producer.close();
}
}

```

2. Consumer Code:

java

```

import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

import java.time.Duration;
import java.util.Collections;
import java.util.Properties;

public class KeyValueConsumer {
    public static void main(String[] args) {
        String topic = "key-value-topic";

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

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

        // Consume messages
        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(1000));
            records.forEach(record -> {
                System.out.println("Received: Key=" + record.key() + ", Value=" + record.value());
            });
        }
    }
}

```

- **Key Insight:** In this example, messages are grouped by `key` (e.g., `user0`, `user1`). Consumers can leverage this grouping for processing related events together.

Summary:

- **Schema Registration:** Centralizes schema management, ensures compatibility, and facilitates serialization/deserialization.
- **Key-Value Messages:** Crucial for partitioning, ordering, efficient retrieval, and analytics in Kafka. Both concepts are fundamental to building robust Kafka applications.



- Write a simple Kafka producer code that sends JSON messages to a topic.
- How do you serialize a custom object before sending it to a Kafka topic? kindly try to answer given questions in details along with example in Java



1. Simple Kafka Producer Code to Send JSON Messages

Steps to Send JSON Messages:

1. Use a **JSON serializer** to convert Java objects or strings into JSON format.
2. Configure the Kafka producer with a JSON serializer.
3. Send the JSON message to a Kafka topic.

Maven Dependencies:

Add the following dependencies in `pom.xml`:

```
xml

<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>3.6.0</version>
</dependency>
<dependency>
```

```
<groupId>com.fasterxml.jackson.core</groupId>
<artifactId>jackson-databind</artifactId>
<version>2.15.2</version>
</dependency>
```

Producer Code:

java

```
import com.fasterxml.jackson.databind.ObjectMapper;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;

import java.util.HashMap;
import java.util.Map;
import java.util.Properties;

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

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

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

        // Example JSON message
        Map<String, Object> message = new HashMap<>();
        message.put("id", 1);
        message.put("name", "John Doe");
        message.put("email", "johndoe@example.com");

        try {
            // Convert the message to JSON
            ObjectMapper objectMapper = new ObjectMapper();
            String jsonMessage = objectMapper.writeValueAsString(message);

            // Send JSON message to Kafka
            producer.send(new ProducerRecord<>(topic, "user1", jsonMessage));
            System.out.println("Sent message: " + jsonMessage);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
        }
```

```

        producer.close();
    }
}

```

Explanation:

1. **ObjectMapper**: Converts the Java object (`message` map) into a JSON string.
2. **ProducerRecord**: Sends the JSON string as the value, with a key (`user1`).

2. How to Serialize a Custom Object Before Sending It to a Kafka Topic

To serialize a custom object, implement a **custom serializer** by extending Kafka's `Serializer` interface. The serialized object will be sent as the message value.

Steps:

1. Define the custom object.
2. Implement a custom serializer for the object.
3. Configure the Kafka producer to use the custom serializer.

Example in Java:

Step 1: Define the Custom Object

```

java

public class User {
    private int id;
    private String name;
    private String email;

    // Constructor
    public User(int id, String name, String email) {
        this.id = id;
        this.name = name;
        this.email = email;
    }
}

```

```
}

// Getters and setters
public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}
}
```

Step 2: Implement the Custom Serializer

java

```
import com.fasterxml.jackson.databind.ObjectMapper;
import org.apache.kafka.common.serialization.Serializer;

import java.util.Map;

public class UserSerializer implements Serializer<User> {
    private final ObjectMapper objectMapper = new ObjectMapper();

    @Override
    public void configure(Map<String, ?> configs, boolean isKey) {}

    @Override
    public byte[] serialize(String topic, User data) {
        try {
            // Convert User object to JSON bytes

```

```
        return objectMapper.writeValueAsBytes(data);
    } catch (Exception e) {
        throw new RuntimeException("Error serializing User object", e);
    }
}

@Override
public void close() {}
}
```

Step 3: Producer Code to Send the Custom Object

java

```
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;

import java.util.Properties;

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

        // Kafka producer configuration
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer", UserSerializer.class.getName()); // Use custom serializer

        KafkaProducer<String, User> producer = new KafkaProducer<>(props);

        // Create a User object
        User user = new User(1, "Jane Doe", "janedoe@example.com");

        // Send the custom object to Kafka
        producer.send(new ProducerRecord<>(topic, "user1", user));
        System.out.println("Sent User object: " + user.getName());

        producer.close();
    }
}
```

Key Concepts and Advantages:

1. Custom Serialization:

- Converts objects into bytes suitable for transmission.
- Allows sending complex objects directly to Kafka.

2. Why Use a Custom Serializer?:

- Reduces boilerplate code for serialization in the producer.
- Improves code maintainability and clarity.

3. Integration with Deserializer:

- A corresponding custom deserializer can be implemented for the consumer to deserialize the bytes back into the original object.

By using custom serializers and JSON messages, you ensure efficient communication in Kafka while maintaining flexibility in message formats.



- Describe how you can handle serialization errors in Kafka producers.
- Write a Kafka consumer code that reads messages from a topic and deserializes them from JSON. kindly try to answer given questions in details along with example in Java



1. Handling Serialization Errors in Kafka Producers

Serialization errors occur when the Kafka producer fails to serialize a message key or value before sending it to a Kafka topic. Proper error handling ensures these errors are caught, logged, and managed effectively.

Strategies to Handle Serialization Errors:

- 1. Logging and Exception Handling:** Wrap the `send()` method in a try-catch block to handle exceptions. Log errors for debugging and alerting.

```
java

try {
    producer.send(new ProducerRecord<>(topic, key, value));
} catch (SerializationException e) {
    System.err.println("Serialization error: " + e.getMessage());
}
```

```
e.printStackTrace();
}
```

2. **Custom Serializer with Validation:** Implement a custom serializer to validate the data before serialization. If invalid data is detected, throw a descriptive exception.

```
java

public class ValidatingSerializer<T> implements Serializer<T> {
    @Override
    public byte[] serialize(String topic, T data) {
        if (data == null) {
            throw new SerializationException("Data cannot be null");
        }
        // Serialization logic here
        return serializeObject(data);
    }
}
```

3. **Error Notification or Dead Letter Queue (DLQ):**

- Redirect problematic messages to a **Dead Letter Queue (DLQ)** for later analysis.
- Example: If serialization fails, send the message (or its error details) to a dedicated error-handling topic.

4. **Producer Callbacks:** Use the `send()` method with a callback to handle errors asynchronously.

```
java

producer.send(new ProducerRecord<>(topic, key, value), (metadata, exception) -> {
    if (exception != null) {
        System.err.println("Error sending message: " + exception.getMessage());
    } else {
        System.out.println("Message sent successfully to topic: " + metadata.topic());
    }
});
```

5. **Testing and Validation:**

- Before sending messages, validate the object structure and data types.
- Use unit tests to ensure the custom serializer handles edge cases.

2. Kafka Consumer Code to Read and Deserialize JSON Messages

Steps:

1. Configure the consumer to use a JSON deserializer.
2. Deserialize the JSON message into a Java object.
3. Process the deserialized data.

Maven Dependencies:

Ensure the following dependencies are added:

xml

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>3.6.0</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.15.2</version>
</dependency>
```

Consumer Code:

java

```
import com.fasterxml.jackson.databind.ObjectMapper;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerRecord;

import java.time.Duration;
import java.util.Collections;
import java.util.Properties;
```

```
public class JsonConsumer {
    public static void main(String[] args) {
        String topic = "json-topic";

        // Kafka consumer configuration
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.put("group.id", "json-group");
        props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
        props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

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

        // JSON ObjectMapper for deserialization
        ObjectMapper objectMapper = new ObjectMapper();

        try {
            while (true) {
                ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(1000));
                for (ConsumerRecord<String, String> record : records) {
                    try {
                        // Deserialize JSON message
                        User user = objectMapper.readValue(record.value(), User.class);
                        System.out.println("Received message: " + user);
                    } catch (Exception e) {
                        System.err.println("Error deserializing message: " + e.getMessage());
                    }
                }
            }
        } finally {
            consumer.close();
        }
    }
}

// Define the User class
class User {
    private int id;
    private String name;
    private String email;

    // Getters and setters
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }

    @Override
    public String toString() {
```

```
        return "User{id=" + id + ", name='" + name + "', email='" + email + "'}";  
    }  
}
```

Explanation:

1. ObjectMapper:

- Used for deserializing JSON strings into Java objects.
- Example: Converts the JSON payload into a `User`` object.

2. Error Handling:

- If deserialization fails (e.g., due to a malformed JSON), the exception is logged, and the consumer continues processing other messages.

3. Kafka Consumer:

- Configured with a **StringDeserializer** for both the key and value since the value is initially received as a JSON string.

JSON Example:

Suppose the message in the Kafka topic is:

```
json  
  
{  
  "id": 1,  
  "name": "John Doe",  
  "email": "johndoe@example.com"  
}
```

The deserialized `User`` object will be:

```
bash  
  
User{id=1, name='John Doe', email='johndoe@example.com'}
```

Summary:

- **Serialization Error Handling:** Implement strategies like logging, validation, DLQs, and callbacks to handle errors robustly.
- **JSON Deserialization:** Use `ObjectMapper` to convert JSON messages into Java objects for processing in the consumer. The approach ensures that data from Kafka is type-safe and usable directly.



- What is a consumer group in Kafka, and why is it important?
 - Describe a scenario where multiple consumer groups are used for a single topic. kindly try to answer given questions in details along with example in Java



1. What is a Consumer Group in Kafka, and Why is it Important?

Consumer Group Definition:

A **consumer group** in Kafka is a group of one or more consumers that work together to consume messages from a Kafka topic. Each consumer in the group is assigned a subset of the topic's partitions, ensuring that every message is consumed by exactly one consumer within the group.

Key Characteristics:

1. **Exclusive Partition Ownership:** Each partition of a topic is assigned to only one consumer in a group at a time, preventing duplicate message consumption within the group.
2. **Scalability:** As more consumers are added to the group, they share the workload by consuming different partitions.
3. **Fault Tolerance:** If a consumer fails, the partitions assigned to it are reassigned to other consumers in the group, ensuring continuity.
4. **Offset Tracking:** The group tracks the offset of messages consumed by each consumer, enabling rebalancing and recovery in case of failure.

Importance:

1. **Parallelism:** Consumer groups allow parallel processing of messages by distributing partitions across multiple consumers.

2. **Message Routing:** Each consumer group processes messages independently, making it possible to have multiple consumers process the same topic differently.
3. **Load Balancing:** The partition rebalancing mechanism ensures load balancing among consumers within a group.

Example:

For a topic with 4 partitions:

- A consumer group with 2 consumers will process the messages from 4 partitions, with each consumer handling 2 partitions.
 - Adding more consumers (up to 4) increases parallelism but cannot exceed the number of partitions.
-

2. Scenario: Multiple Consumer Groups for a Single Topic

Use Case:

Imagine a **log-processing system** where a Kafka topic receives application logs. Different teams need to process these logs for different purposes:

1. Real-Time Monitoring Team:

- Consumes logs for real-time alert generation.
- Uses a consumer group (``monitoring-group``) to ensure that every log is processed quickly.

2. Analytics Team:

- Consumes logs to perform batch processing and generate insights.
- Uses a separate consumer group (``analytics-group``) to ensure independent processing without affecting real-time monitoring.

In this setup:

- **Consumer Group 1** (``monitoring-group``) ensures real-time processing.
- **Consumer Group 2** (``analytics-group``) ensures data is reprocessed without affecting the first group.

Diagram:

vbnet

```
Topic: app-logs
Partitions: [P1, P2, P3, P4]

Consumer Group 1 (monitoring-group):
- Consumer 1 processes P1 and P2.
- Consumer 2 processes P3 and P4.

Consumer Group 2 (analytics-group):
- Consumer 3 processes P1 and P2.
- Consumer 4 processes P3 and P4.
```

Each group consumes the topic independently.

Java Example for Multiple Consumer Groups

Step 1: Producer Code (Sending Logs)

java

```
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;

import java.util.Properties;

public class LogProducer {
    public static void main(String[] args) {
        String topic = "app-logs";

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

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

        // Simulate log messages
        for (int i = 0; i < 10; i++) {
            producer.send(new ProducerRecord<>(topic, "key-" + i, "Log message " + i));
            System.out.println("Sent: Log message " + i);
        }
    }
}
```

```
        producer.close();  
    }  
}
```

Step 2: Consumer Code for Monitoring Group

java

```
import org.apache.kafka.clients.consumer.ConsumerRecords;  
import org.apache.kafka.clients.consumer.KafkaConsumer;  
  
import java.time.Duration;  
import java.util.Collections;  
import java.util.Properties;  
  
public class MonitoringConsumer {  
    public static void main(String[] args) {  
        String topic = "app-logs";  
  
        Properties props = new Properties();  
        props.put("bootstrap.servers", "localhost:9092");  
        props.put("group.id", "monitoring-group");  
        props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");  
        props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");  
  
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);  
        consumer.subscribe(Collections.singletonList(topic));  
  
        while (true) {  
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(1000));  
            records.forEach(record -> {  
                System.out.println("Monitoring Consumer: " + record.value());  
            });  
        }  
    }  
}
```

Step 3: Consumer Code for Analytics Group

```
java
```

```
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

import java.time.Duration;
import java.util.Collections;
import java.util.Properties;

public class AnalyticsConsumer {
    public static void main(String[] args) {
        String topic = "app-logs";

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

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

        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(1000));
            records.forEach(record -> {
                System.out.println("Analytics Consumer: " + record.value());
            });
        }
    }
}
```

Summary:

1. Consumer Group:

- Ensures distributed and fault-tolerant consumption.
- Important for scaling and ensuring parallelism.

2. Multiple Consumer Groups:

- Used when different consumers need to process the same topic independently.
- Example: Real-time monitoring and batch analytics for logs.

3. Java Implementation:

- Producers send messages to the topic.
- Different consumer groups independently process the messages based on their needs.



- How does Kafka ensure load balancing among consumers in a group?
- How do you send JSON data to a Kafka topic and ensure it is properly serialized? kindly try to answer given questions in details along with example in Java



1. How Does Kafka Ensure Load Balancing Among Consumers in a Group?

Key Mechanism: Partition Assignment

Kafka achieves load balancing among consumers in a group through the **partition assignment** process. Here's how it works:

1. Partitions in a Topic:

- A Kafka topic is divided into partitions. Each partition can be consumed by only one consumer within the same group at a time.

2. Consumer Group:

- Consumers in a group share the responsibility of consuming messages from the topic's partitions. Kafka ensures that the partitions are distributed evenly among the available consumers.

3. Partition Assignment Strategy: Kafka uses partition assignment strategies to allocate partitions to consumers:

- **RangeAssignor** (default): Assigns contiguous partitions to consumers.
- **RoundRobinAssignor**: Distributes partitions evenly among consumers.
- **StickyAssignor**: Similar to `RoundRobin`, but tries to minimize partition reassignments during rebalancing.

4. Rebalancing:

- When a new consumer joins the group or an existing consumer leaves, Kafka triggers a **rebalance**.
- During rebalancing, partitions are reassigned to ensure all partitions are consumed and workload is distributed.

Example:

Consider a topic with 6 partitions (`P0`` to `P5``) and a consumer group with 3 consumers:

- **Initial Assignment:**

- Consumer 1: `P0`, `P1`
- Consumer 2: `P2`, `P3`
- Consumer 3: `P4`, `P5`

- **Adding a New Consumer (Consumer 4):**

- Kafka triggers a rebalance:
 - Consumer 1: `P0`
 - Consumer 2: `P1`
 - Consumer 3: `P2`, `P3`
 - Consumer 4: `P4`, `P5`

2. How Do You Send JSON Data to a Kafka Topic and Ensure Proper Serialization?

To send JSON data to a Kafka topic, you need to:

1. Serialize the JSON data properly.
2. Configure the Kafka producer with the appropriate serializers.

Steps:

Step 1: Add Maven Dependencies

```
xml<dependency>  
  <groupId>org.apache.kafka</groupId>  
  <artifactId>kafka-clients</artifactId>  
  <version>3.6.0</version>  
</dependency>  
<dependency>  
  <groupId>com.fasterxml.jackson.core</groupId>  
  <artifactId>jackson-databind</artifactId>  
  <version>2.15.2</version>  
</dependency>
```

Step 2: Define a Custom JSON Serializer

```
java

import com.fasterxml.jackson.databind.ObjectMapper;
import org.apache.kafka.common.serialization.Serializer;

import java.util.Map;

public class JsonSerializer<T> implements Serializer<T> {
    private final ObjectMapper objectMapper = new ObjectMapper();

    @Override
    public void configure(Map<String, ?> configs, boolean isKey) {}

    @Override
    public byte[] serialize(String topic, T data) {
        try {
            return objectMapper.writeValueAsBytes(data);
        } catch (Exception e) {
            throw new RuntimeException("Error serializing JSON data", e);
        }
    }

    @Override
    public void close() {}
}
```

Step 3: Define a POJO for JSON Data

```
java

public class User {
    private int id;
    private String name;
    private String email;

    // Constructors
    public User() {}
    public User(int id, String name, String email) {
        this.id = id;
        this.name = name;
        this.email = email;
    }
}
```

```
// Getters and Setters
public int getId() { return id; }
public void setId(int id) { this.id = id; }
public String getName() { return name; }
public void setName(String name) { this.name = name; }
public String getEmail() { return email; }
public void setEmail(String email) { this.email = email; }

@Override
public String toString() {
    return "User{id=" + id + ", name='" + name + "', email='" + email + "'}";
}
}
```

Step 4: Kafka Producer Code

java

```
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;

import java.util.Properties;

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

        // Kafka producer configuration
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer", JsonSerializer.class.getName()); // Use custom JSON serializer

        KafkaProducer<String, User> producer = new KafkaProducer<>(props);

        // Create a User object
        User user = new User(1, "John Doe", "johndoe@example.com");

        // Send JSON data to Kafka
        producer.send(new ProducerRecord<>(topic, "user1", user), (metadata, exception) -> {
            if (exception != null) {
                System.err.println("Error sending message: " + exception.getMessage());
            } else {
                System.out.println("Message sent to topic: " + metadata.topic() + ", partition: " + metadata.partition());
            }
        });
    }
}
```

```
});
    producer.close();
}
```

Key Concepts:

1. Load Balancing in Consumer Groups:

- Achieved via partition assignment strategies and rebalancing.
- Ensures efficient message consumption and scalability.

2. Sending JSON to Kafka:

- Use a custom serializer to convert objects to JSON.
- Configure the producer with the custom serializer for seamless serialization.

By using the custom `JsonSerializer`, you can send any object (converted to JSON) to a Kafka topic, ensuring type safety and flexibility.



- Describe the process of consuming JSON data from a Kafka topic and converting it to a usable format.
- Explain how you can work with CSV data in Kafka, including serialization and deserialization. kindly try to answer given questions in details along with example in Java



1. Consuming JSON Data from a Kafka Topic and Converting it to a Usable Format

To consume JSON data from a Kafka topic, the following steps are involved:

1. **Deserialize the JSON Data:** Use a JSON deserializer to convert the JSON string into a Java object.
2. **Configure the Kafka Consumer:** Set up the consumer to read messages from the topic where JSON data is published.

3. **Process the Data:** Parse and map the JSON data into a usable Java object format (e.g., POJOs).

Example: Java Consumer for JSON Data

Step 1: Add Maven Dependencies

```
xml

<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>3.6.0</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version>2.15.2</version>
</dependency>
```

Step 2: Define a POJO for JSON Data

```
java

public class User {
  private int id;
  private String name;
  private String email;

  // Constructors
  public User() {}
  public User(int id, String name, String email) {
    this.id = id;
    this.name = name;
    this.email = email;
  }

  // Getters and Setters
  public int getId() { return id; }
  public void setId(int id) { this.id = id; }
  public String getName() { return name; }
  public void setName(String name) { this.name = name; }
```

```
public String getEmail() { return email; }
public void setEmail(String email) { this.email = email; }

@Override
public String toString() {
    return "User{id=" + id + ", name='" + name + "', email='" + email + "'}";
}
}
```

Step 3: Consumer Code for JSON

java

```
import com.fasterxml.jackson.databind.ObjectMapper;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

import java.time.Duration;
import java.util.Collections;
import java.util.Properties;

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

        // Kafka consumer configuration
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.put("group.id", "json-group");
        props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
        props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

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

        ObjectMapper objectMapper = new ObjectMapper();

        try {
            while (true) {
                ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(1000));
                records.forEach(record -> {
                    try {
                        User user = objectMapper.readValue(record.value(), User.class);
                        System.out.println("Consumed: " + user);
                    } catch (Exception e) {
                        System.err.println("Error deserializing JSON: " + e.getMessage());
                    }
                });
            }
        }
```

```

    });
  }
} finally {
  consumer.close();
}
}
}

```

2. Working with CSV Data in Kafka

Kafka can handle **CSV (Comma-Separated Values)** data similarly to JSON by creating custom serializers and deserializers. Here's how:

Steps:

1. **Serialize CSV Data:** Convert CSV strings into byte arrays when producing messages.
2. **Deserialize CSV Data:** Parse CSV strings into Java objects when consuming messages.
3. **Configure Producer and Consumer:** Use custom serializers/deserializers for handling CSV data.

Example: Handling CSV Data in Kafka

Step 1: Define a Custom Serializer for CSV

```

java

import org.apache.kafka.common.serialization.Serializer;
import java.nio.charset.StandardCharsets;

public class CsvSerializer<T> implements Serializer<T> {
    @Override
    public byte[] serialize(String topic, T data) {
        if (data == null) return null;
        return data.toString().getBytes(StandardCharsets.UTF_8);
    }
}

```


Step 2: Define a Custom Deserializer for CSV

```
java

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

import java.nio.charset.StandardCharsets;

public class CsvDeserializer<T> implements Deserializer<T> {
    @Override
    public T deserialize(String topic, byte[] data) {
        if (data == null) return null;
        String csv = new String(data, StandardCharsets.UTF_8);
        // Custom logic to parse CSV data and map to a Java object
        String[] fields = csv.split(",");
        return (T) new User(Integer.parseInt(fields[0]), fields[1], fields[2]);
    }
}
```

Step 3: Producer Code for CSV Data

```
java

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;

import java.util.Properties;

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

        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer", CsvSerializer.class.getName());

        KafkaProducer<String, User> producer = new KafkaProducer<>(props);

        User user = new User(1, "Jane Doe", "jane.doe@example.com");
        producer.send(new ProducerRecord<>(topic, "user1", user));

        producer.close();
    }
}
```

```
}
}
```

Step 4: Consumer Code for CSV Data

java

```
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

import java.time.Duration;
import java.util.Collections;
import java.util.Properties;

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

        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.put("group.id", "csv-group");
        props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
        props.put("value.deserializer", CsvDeserializer.class.getName());

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

        try {
            while (true) {
                ConsumerRecords<String, User> records = consumer.poll(Duration.ofMillis(1000));
                records.forEach(record -> {
                    System.out.println("Consumed CSV: " + record.value());
                });
            }
        } finally {
            consumer.close();
        }
    }
}
```

Summary:

JSON Data:

1. Use `ObjectMapper` for serialization/deserialization.
2. Configure Kafka with standard serializers and deserializers.
3. Parse JSON into POJOs for easy use.

CSV Data:

1. Implement custom serializers/deserializers for handling CSV.
2. Convert CSV strings into Java objects during deserialization.
3. Use the custom serializers in Kafka producer/consumer configurations.

Both formats can be effectively handled in Kafka by tailoring the serialization/deserialization processes to suit the specific data structure.