Using Apache Kafka in Java

Using Apache Kafka in Java involves setting up a Kafka client in your Java application to produce and consume messages. Kafka is a distributed streaming platform, and its Java client libraries make it straightforward to integrate. Below, I'll walk you through the basic steps to get started.

First, you'll need to set up your environment. Ensure you have Kafka installed and running on your system or a server. You can download it from the official Apache Kafka website and start the ZooKeeper and Kafka server using the provided scripts. For simplicity, I'll assume you're running Kafka locally with default settings (e.g., `localhost:9092` as the bootstrap server).

Next, add the Kafka client dependency to your Java project. If you're using Maven, include this in your `pom.xml`:

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>3.6.0</version> <!-- Use the latest version -->
</dependency>
```

Now, let's write some code. I'll show you how to create a simple producer and consumer.

Kafka Producer Example

The producer sends messages to a Kafka topic. Here's a basic example:

```
import org.apache.kafka.clients.producer.*;
import java.util.Properties;

public class SimpleProducer {
    public static void main(String[] args) {
        // Configure producer properties
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092"); // Kafka server address
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

        // Create a producer instance
        try (Producer<String, String> producer = new KafkaProducer<>(props)) {
            // Send a message to a topic called "test-topic"
            String topic = "test-topic";
            for (int i = 0; i < 10; i++) {
```

```
                String key = "key" + i;
                String value = "Hello, Kafka " + i;
                ProducerRecord<String, String> record = new ProducerRecord<>(topic, key, value);

                producer.send(record, (metadata, exception) -> {
                    if (exception == null) {
                        System.out.println("Sent message: " + value + " to partition " + metadata.partition());
                    } else {
                        exception.printStackTrace();
                    }
                });
            }
        }
    }
}
```

In this code: - `bootstrap.servers` specifies where Kafka is running. - The serializers define how keys and values (both strings here) are converted to bytes. - `ProducerRecord` represents the message, and `send()` dispatches it asynchronously with a callback to handle success or failure.

Kafka Consumer Example

The consumer subscribes to a topic and reads messages. Here's an example:

```
import org.apache.kafka.clients.consumer.*;
import java.util.Collections;
import java.util.Properties;

public class SimpleConsumer {
    public static void main(String[] args) {
        // Configure consumer properties
        Properties props = new Properties();
        props.put("bootstrap.servers", "localhost:9092");
        props.put("group.id", "test-group"); // Consumer group ID
        props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
        props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
        props.put("auto.offset.reset", "earliest"); // Start from the beginning of the topic

        // Create a consumer instance
        try (KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props)) {
            // Subscribe to the topic
```

```
            consumer.subscribe(Collections.singletonList("test-topic"));

            // Poll for messages
            while (true) {
                ConsumerRecords<String, String> records = consumer.poll(1000); // Timeout in ms
                for (ConsumerRecord<String, String> record : records) {
                    System.out.printf("Received: key=%s, value=%s, partition=%d, offset=%d%n",
                            record.key(), record.value(), record.partition(), record.offset());
                }
            }
        }
    }
}
```

Here: - `group.id` identifies the consumer group for coordination. - Deserializers convert the byte data back to strings. - `poll()` fetches messages in a loop, and `auto.offset.reset` ensures you start from the earliest message if no offset is committed.

Running the Code

1. **Start Kafka**: Ensure ZooKeeper and Kafka are running (e.g., `bin/zookeeper-server-start.sh config/zookeeper.properties` and `bin/kafka-server-start.sh config/server.properties` on Unix-like systems).
2. **Create a Topic**: Use the Kafka command-line tool to create `test-topic` if it doesn't exist: `bin/kafka-topics.sh --create --topic test-topic --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1`.
3. **Run the Producer**: Execute the `SimpleProducer` class to send messages.
4. **Run the Consumer**: Run the `SimpleConsumer` class to see the messages being consumed.

Tips

- **Error Handling**: Add try-catch blocks or logging for production code.
- **Configuration**: Adjust properties like `acks` (for producer reliability) or `max.poll.records` (for consumer batch size) based on your needs.
- **Libraries**: For more complex applications, consider using Spring Kafka, which simplifies configuration and usage.

This is a basic setup to get you going. Let me know if you want to dive deeper into specific features like partitioning, error handling, or integrating with a larger application!