server:

port: 8080

spring:

kafka:

bootstrap-servers: localhost:9092 # Kafka broker

producer:

key-serializer: org.apache.kafka.common.serialization.StringSerializer

value-serializer: org.apache.kafka.common.serialization.StringSerializer

retries: 3 # Retries for producer in case of failure

acks: all # Ensure all replicas acknowledge the message

consumer:

group-id: order-consumers # Consumer group for load balancing

auto-offset-reset: earliest # Read from the beginning if no offset is found

key-deserializer: org.apache.kafka.common.serialization.StringDeserializer

value-deserializer: org.apache.kafka.common.serialization.StringDeserializer

enable-auto-commit: false # Manual offset commit

max-poll-records: 10 # Max number of records consumed in a single poll

properties:

isolation.level: read\_committed # Ensures exactly-once processing in transactions

listener:

missing-topics-fatal: false # Allow the application to start without existing topics

**Explanation of Key Configurations:**

* **Producer Settings**:
  + **retries**: Specifies the number of retry attempts for a producer in case of a failure to send a message.
  + **acks**: Controls the acknowledgment level for message delivery. all ensures that all replicas acknowledge the message.
* **Consumer Settings**:
  + **auto-offset-reset**: Configured as earliest to start consuming from the earliest available message if there is no committed offset.
  + **enable-auto-commit**: Disabled to manually commit offsets after the message is processed.
  + **isolation.level**: This ensures that the consumer reads only committed data, which is important for exactly-once semantics in Kafka.
* **Listener Settings**:
  + **missing-topics-fatal**: By default, Spring Boot requires that all topics be created beforehand. Setting this to false allows the application to start even if the Kafka topic doesn't yet exist.

**Spring Boot Kafka Auto-Configuration**

With these configurations in application.yml, Spring Boot will automatically set up the producer and consumer beans, as well as handle the Kafka listener. You don't need a separate KafkaConfig class anymore because Spring Boot will create all the necessary beans and configurations.

**Updated Classes (Without KafkaConfig)**

1. **Producer**: No change is needed in the producer class.

package com.example.kafkademo.producer;

import org.springframework.kafka.core.KafkaTemplate;

import org.springframework.stereotype.Service;

@Service

public class OrderProducer {

private static final String TOPIC = "order\_topic";

private final KafkaTemplate<String, String> kafkaTemplate;

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

this.kafkaTemplate = kafkaTemplate;

}

public void sendOrderMessage(String message) {

kafkaTemplate.send(TOPIC, message);

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

}

}

package com.example.kafkademo.consumer;

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

import org.springframework.kafka.annotation.KafkaListener;

import org.springframework.kafka.support.Acknowledgment;

import org.springframework.stereotype.Service;

@Service

public class OrderConsumer {

@KafkaListener(topics = "order\_topic", groupId = "order-consumers")

public void consumeOrderMessage(ConsumerRecord<String, String> record, Acknowledgment ack) {

String message = record.value();

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

// Process the order...

// Manually acknowledge the message

ack.acknowledge();

}

}

package com.example.kafkademo.controller;

import com.example.kafkademo.producer.OrderProducer;

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

@RestController

@RequestMapping("/api/v1/orders")

public class OrderController {

private final OrderProducer orderProducer;

public OrderController(OrderProducer orderProducer) {

this.orderProducer = orderProducer;

}

@PostMapping

public String sendOrder(@RequestBody String order) {

orderProducer.sendOrderMessage(order);

return "Order sent: " + order;

}

}

**Benefits of Using application.yml for Configuration:**

* **Cleaner code**: All Kafka configurations are consolidated in application.yml, eliminating the need for a separate Java configuration class.
* **Easier to maintain**: As configurations are stored in a single place, updates or environment-specific configurations can be easily modified without touching the codebase.
* **Environment-specific settings**: You can maintain different Kafka configurations for different environments (dev, staging, prod) by using Spring Boot profiles (e.g., application-dev.yml, application-prod.yml).