

Kafka Listener Guidance

You said:



kafka listener

ChatGPT said:



Are you looking for a Kafka listener implementation, or do you need guidance on setting up and configuring a Kafka listener in a particular environment or programming language? Let me know how I can assist you!

You said:



org.springframework.kafka.annotations.kafkalistener

Using application.yml

ChatGPT said:



To set up a Kafka listener using Spring's `@KafkaListener` annotation and configure it using `application.yml`, you can follow these steps:

1. Add Dependencies

Make sure you have the necessary dependencies in your `pom.xml` (for Maven) or `build.gradle` (for Gradle).

For Maven:

xml

```
<dependency>
  <groupId>org.springframework.kafka</groupId>
  <artifactId>spring-kafka</artifactId>
</dependency>
```

For Gradle:

```
groovy
```

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

2. Configure Kafka Properties in `application.yml`

You need to configure the Kafka broker details, consumer group ID, topic, etc., in your `application.yml` file:

```
yaml
```

```
spring:
  kafka:
    bootstrap-servers: localhost:9092
    consumer:
      group-id: your-group-id
      auto-offset-reset: earliest
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer: org.apache.kafka.common.serialization.StringDeserializer
    listener:
      concurrency: 3 # Optional, to specify the number of threads to use for this
listener
  topic:
    name: your-topic-name
```

3. Implement the Kafka Listener

Create a Spring component where you can use the `@KafkaListener` annotation to listen to the Kafka topic:

```
java
```

```
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;

@Service
public class KafkaConsumerService {

    @KafkaListener(topics = "${spring.kafka.topic.name}", groupId =
"${spring.kafka.consumer.group-id}")
    public void listen(String message) {
        System.out.println("Received message: " + message);
        // Process the message here
    }
}
```

4. Application Entry Point

Ensure your application has a main entry point that runs your Spring Boot application:

```
java
```

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

```
@SpringBootApplication
public class KafkaApplication {

    public static void main(String[] args) {
        SpringApplication.run(KafkaApplication.class, args);
    }
}
```

5. Run the Application

Once your application is set up, you can run it, and the Kafka listener will start consuming messages from the specified topic.

Additional Configurations

You can further customize the `application.yml` with additional Kafka configurations, such as `max-poll-records`, `heartbeat-interval`, etc., depending on your needs.

Let me know if you need further customization or additional details!

You said:



ConsumeRecord

ChatGPT said:



To consume a Kafka record in Spring using the `@KafkaListener` annotation and handle the record as a `ConsumerRecord`, you can modify the listener method to accept a `ConsumerRecord` object. This provides you with more control and access to additional metadata, such as the partition, offset, headers, etc.

1. Update `@KafkaListener` to Consume `ConsumerRecord`

Here's how you can modify the listener method to consume a `ConsumerRecord`:

java

```
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;

@Service
public class KafkaConsumerService {

    @KafkaListener(topics = "${spring.kafka.topic.name}", groupId =
"${spring.kafka.consumer.group-id}")
    public void listen(ConsumerRecord<String, String> record) {
        String topic = record.topic();
        int partition = record.partition();
        long offset = record.offset();
        String key = record.key();
        String value = record.value();
    }
}
```

```

        System.out.println("Received message: " + value);
        System.out.println("From topic: " + topic + ", partition: " + partition + ",
offset: " + offset);
        System.out.println("Key: " + key);

        // Additional processing logic here
    }
}

```

2. Explanation of `ConsumerRecord`

- **topic:** The topic from which the record was consumed.
- **partition:** The partition number of the topic.
- **offset:** The offset of the record within the partition.
- **key:** The key of the record (if present).
- **value:** The actual message payload.

3. `application.yml` Configuration

The `application.yml` configuration remains the same as previously mentioned, as the `ConsumerRecord` is handled within the `@KafkaListener` method:

yaml

```

spring:
  kafka:
    bootstrap-servers: localhost:9092
    consumer:
      group-id: your-group-id
      auto-offset-reset: earliest
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer: org.apache.kafka.common.serialization.StringDeserializer
    listener:
      concurrency: 3
    topic:
      name: your-topic-name

```

4. Handling More Complex Records

If you're dealing with more complex data, like Avro or JSON, you may need to modify the `ConsumerRecord` type to handle the deserialized data properly. The key and value types can be adapted based on your deserialization logic.

This setup allows you to consume and process Kafka records with detailed control over each record's metadata. Let me know if you need any further customization!

You said:



Difference between listen and consume

ChatGPT said:



In the context of Kafka and Spring, the terms "listen" and "consume" are often used interchangeably, but there are nuances in how they are typically understood:

1. Listen

- **Conceptual Meaning:** "Listening" generally refers to the action of a service or component that waits for messages to arrive on a Kafka topic. It implies a continuous, passive action where the service is always ready to receive messages.
- **In Spring Kafka:** The `@KafkaListener` annotation is used to define a method that "listens" to a Kafka topic. The method is automatically invoked when a new message arrives on the specified topic.

Example:

```
java
```

```
@KafkaListener(topics = "example-topic")
public void listen(String message) {
    System.out.println("Received message: " + message);
}
```

Here, the method annotated with `@KafkaListener` is "listening" to the topic. The term "listen" emphasizes the ongoing nature of this activity.

2. Consume

- **Conceptual Meaning:** "Consuming" refers to the action of reading or processing a message that has been received from a Kafka topic. This is a more active term that focuses on the handling of the message once it has been delivered to the consumer.
- **In Kafka Generally:** Consumers are entities that "consume" messages from Kafka topics. The process of consuming involves fetching messages from Kafka and performing some operation on them, such as storing them in a database or triggering some business logic. In the context of Kafka, "consuming" is the broader term that encompasses the entire process of fetching and processing messages.
- **In Spring Kafka:** The term "consume" might not be explicitly used in the method name, but the method inside a `@KafkaListener` is effectively "consuming" the messages it receives. If you use a `ConsumerRecord` parameter, it suggests that you're consuming not just the message, but also its associated metadata.

Example:

```
java
```

```
@KafkaListener(topics = "example-topic")
public void consume(ConsumerRecord<String, String> record) {
    String message = record.value();
    System.out.println("Consumed message: " + message);
}
```

Here, the method is "consuming" the `ConsumerRecord`, which contains the message and additional information like topic, partition, and offset.

Summary

- **Listen:** Focuses on the continuous action of waiting for and receiving messages. In Spring Kafka, this is set up using the `@KafkaListener` annotation.
- **Consume:** Focuses on the action of processing the messages after they have been received. While listening is about waiting for the messages, consuming is about what you do with them once they arrive.

In practice, these terms are closely related, and in many cases, listening to a Kafka topic is synonymous with consuming its messages. However, "listening" emphasizes the readiness to receive, while "consuming" emphasizes the handling of the received data.

You said:



If you're dealing with more complex data, like Avro or JSON, you may need to modify the `ConsumerRecord` type to handle the deserialized data properly. The key and value types can be adapted based on your deserialization logic.

Example

ChatGPT said:



When dealing with more complex data formats like Avro or JSON in Kafka, you can customize the `ConsumerRecord` to handle the deserialization properly. This involves configuring your Kafka consumer to use the appropriate deserializers and then processing the deserialized data within your `@KafkaListener` method.

1. Using JSON Data

Configuration in `application.yml`

To handle JSON data, you can configure the consumer to use a JSON deserializer.

yaml

```
spring:
  kafka:
    bootstrap-servers: localhost:9092
    consumer:
      group-id: your-group-id
      auto-offset-reset: earliest
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer:
        org.springframework.kafka.support.serializer.ErrorHandlingDeserializer
      properties:
        spring.deserializer.value.delegate.class:
```

```
org.springframework.kafka.support.serializer.JsonDeserializer
listener:
  concurrency: 3
  topic:
    name: your-json-topic-name
```

Define a POJO for JSON Mapping

Let's assume your JSON data maps to a `MyCustomObject` class:

```
java

import com.fasterxml.jackson.annotation.JsonProperty;

public class MyCustomObject {

    @JsonProperty("field1")
    private String field1;

    @JsonProperty("field2")
    private int field2;

    // Getters and Setters
}
```

Kafka Listener with JSON Deserialization

Now, modify your Kafka listener to handle `ConsumerRecord<String, MyCustomObject>`:

```
java

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;

@Service
public class KafkaConsumerService {

    @KafkaListener(topics = "${spring.kafka.topic.name}", groupId =
"${spring.kafka.consumer.group-id}")
    public void consumeJson(ConsumerRecord<String, MyCustomObject> record) {
        MyCustomObject data = record.value();

        System.out.println("Received JSON message: " + data);
        // Process the deserialized JSON data here
    }
}
```

2. Using Avro Data

Configuration in `application.yml`

To handle Avro data, you'll need to use a Kafka Avro deserializer, typically provided by Confluent's Kafka schema registry.

yaml

```
spring:
  kafka:
    bootstrap-servers: localhost:9092
    consumer:
      group-id: your-group-id
      auto-offset-reset: earliest
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer: io.confluent.kafka.serializers.KafkaAvroDeserializer
      properties:
        schema.registry.url: http://localhost:8081 # URL to your schema registry
    listener:
      concurrency: 3
    topic:
      name: your-avro-topic-name
```

Define Avro Schema and Generate Java Classes

Assume you have an Avro schema (`my_custom_object.avsc`):

json

```
{
  "type": "record",
  "name": "MyCustomObject",
  "namespace": "com.example",
  "fields": [
    {"name": "field1", "type": "string"},
    {"name": "field2", "type": "int"}
  ]
}
```

Generate Java classes using Avro tools, or define the class manually if needed.

Kafka Listener with Avro Deserialization

Modify your Kafka listener to handle `ConsumerRecord<String, MyCustomObject>`:

java

```
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;
import com.example.MyCustomObject; // Avro generated class

@Service
public class KafkaConsumerService {

    @KafkaListener(topics = "${spring.kafka.topic.name}", groupId =
"${spring.kafka.consumer.group-id}")
    public void consumeAvro(ConsumerRecord<String, MyCustomObject> record) {
        MyCustomObject data = record.value();

        System.out.println("Received Avro message: " + data);
        // Process the deserialized Avro data here
    }
}
```


Summary

- **JSON Handling:** Use the `JsonDeserializer` provided by Spring Kafka to automatically convert JSON strings to your Java POJOs.
- **Avro Handling:** Use the `KafkaAvroDeserializer` and a schema registry to deserialize Avro-encoded messages to generated Avro classes.

By modifying the `ConsumerRecord` type and configuring the appropriate deserializer, you can effectively handle more complex data types like JSON or Avro within your Spring Kafka application.

You said:



spring Multiple kafka

ChatGPT said:



If you need to set up multiple Kafka listeners in a Spring application, consuming from different Kafka topics or using different configurations, you can achieve this by either defining multiple `@KafkaListener` methods or configuring multiple Kafka consumer factories and containers.

1. Multiple Kafka Listeners with Different Topics

The simplest scenario is having multiple `@KafkaListener` methods, each listening to a different Kafka topic. All these listeners can share the same configuration.

Example:

java

```
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;

@Service
public class MultiKafkaConsumerService {

    @KafkaListener(topics = "topic1", groupId = "group1")
    public void listenToTopic1(ConsumerRecord<String, String> record) {
        System.out.println("Received message from topic1: " + record.value());
    }

    @KafkaListener(topics = "topic2", groupId = "group2")
    public void listenToTopic2(ConsumerRecord<String, String> record) {
        System.out.println("Received message from topic2: " + record.value());
    }

    @KafkaListener(topics = "topic3", groupId = "group3")
    public void listenToTopic3(ConsumerRecord<String, String> record) {
        System.out.println("Received message from topic3: " + record.value());
    }
}
```

In this example:

- ``listenToTopic1`` listens to ``topic1`` with ``group1``.
- ``listenToTopic2`` listens to ``topic2`` with ``group2``.
- ``listenToTopic3`` listens to ``topic3`` with ``group3``.

2. Multiple Kafka Listeners with Different Configurations

If you need different configurations (e.g., different deserializers, bootstrap servers, or group IDs), you'll need to define multiple ``KafkaListenerContainerFactory`` beans and then use those in your listeners.

Configuration in ``application.yml``:

yaml

```
spring:
  kafka:
    consumer:
      group-id: default-group
      auto-offset-reset: earliest
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      bootstrap-servers: localhost:9092

custom-kafka:
  consumer1:
    group-id: group1
    bootstrap-servers: localhost:9092

  consumer2:
    group-id: group2
    bootstrap-servers: localhost:9093
```

Custom Kafka Configuration:

java

```
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.core.DefaultKafkaConsumerFactory;
import org.springframework.kafka.listener.config.ContainerProperties;

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

@Configuration
public class CustomKafkaConfig {

    @Bean
    @ConfigurationProperties(prefix = "custom-kafka.consumer1")
    public Map<String, Object> consumer1Configs() {
        return new HashMap<>();
    }

    @Bean
```

```

    public ConsumerFactory<String, String> consumer1Factory() {
        return new DefaultKafkaConsumerFactory<>(consumer1Configs());
    }

    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, String>
    kafkaListenerContainerFactory1() {
        ConcurrentKafkaListenerContainerFactory<String, String> factory =
            new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumer1Factory());

        factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL_IMMEDIATE);
        return factory;
    }

    @Bean
    @ConfigurationProperties(prefix = "custom-kafka.consumer2")
    public Map<String, Object> consumer2Configs() {
        return new HashMap<>();
    }

    @Bean
    public ConsumerFactory<String, String> consumer2Factory() {
        return new DefaultKafkaConsumerFactory<>(consumer2Configs());
    }

    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, String>
    kafkaListenerContainerFactory2() {
        ConcurrentKafkaListenerContainerFactory<String, String> factory =
            new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumer2Factory());

        factory.getContainerProperties().setAckMode(ContainerProperties.AckMode.MANUAL_IMMEDIATE);
        return factory;
    }
}

```

Kafka Listeners Using Different Factories:

java

```

import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;

@Service
public class MultiKafkaConsumerService {

    @KafkaListener(topics = "topic1", containerFactory = "kafkaListenerContainerFactory1")
    public void listenToTopic1(String message) {
        System.out.println("Received message from topic1: " + message);
    }

    @KafkaListener(topics = "topic2", containerFactory = "kafkaListenerContainerFactory2")
    public void listenToTopic2(String message) {
        System.out.println("Received message from topic2: " + message);
    }
}

```

Summary:

- **Simple Multiple Listeners:** Just use multiple `@KafkaListener` methods, each with a different topic.

- **Multiple Listeners with Different Configurations:** Define custom `KafkaListenerContainerFactory` beans and specify different configurations for each listener using containerFactory`.`

This approach gives you flexibility in handling different Kafka topics with different settings within the same Spring Boot application.

You said:



multiple cluster

ChatGPT said:



Memory updated

To work with multiple Kafka clusters in a Spring Boot application, you can configure each cluster separately and create multiple `KafkaListenerContainerFactory` beans, each associated with a different cluster. Here's how you can set it up:`

1. Configuration in `application.yml``

You can define separate configurations for each Kafka cluster:

yaml

```
spring:
  kafka:
    default:
      consumer:
        group-id: default-group
        auto-offset-reset: earliest
        key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
        value-deserializer: org.apache.kafka.common.serialization.StringDeserializer
        bootstrap-servers: kafka-cluster-1:9092

    kafka-cluster1:
      consumer:
        group-id: group1
        auto-offset-reset: earliest
        key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
        value-deserializer: org.apache.kafka.common.serialization.StringDeserializer
        bootstrap-servers: kafka-cluster-1:9092

    kafka-cluster2:
      consumer:
        group-id: group2
        auto-offset-reset: earliest
        key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
        value-deserializer: org.apache.kafka.common.serialization.StringDeserializer
        bootstrap-servers: kafka-cluster-2:9092
```

2. Custom Kafka Configuration

You'll need to create custom configuration classes to handle each Kafka cluster.

```
java
```

```
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.core.DefaultKafkaConsumerFactory;

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

@Configuration
public class KafkaConfig {

    @Bean
    @ConfigurationProperties(prefix = "kafka-cluster1")
    public Map<String, Object> cluster1ConsumerConfigs() {
        return new HashMap<>();
    }

    @Bean
    public ConsumerFactory<String, String> cluster1ConsumerFactory() {
        return new DefaultKafkaConsumerFactory<>(cluster1ConsumerConfigs());
    }

    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, String>
cluster1KafkaListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, String> factory =
            new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(cluster1ConsumerFactory());
        return factory;
    }

    @Bean
    @ConfigurationProperties(prefix = "kafka-cluster2")
    public Map<String, Object> cluster2ConsumerConfigs() {
        return new HashMap<>();
    }

    @Bean
    public ConsumerFactory<String, String> cluster2ConsumerFactory() {
        return new DefaultKafkaConsumerFactory<>(cluster2ConsumerConfigs());
    }

    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, String>
cluster2KafkaListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, String> factory =
            new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(cluster2ConsumerFactory());
        return factory;
    }
}
```

3. Kafka Listeners Using Different Clusters

You can now create listeners that use different clusters by specifying the appropriate `containerFactory` in the `@KafkaListener` annotation.

```
java
```

```
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;

@Service
public class MultiClusterKafkaConsumerService {

    @KafkaListener(topics = "topic1", containerFactory =
"cluster1KafkaListenerContainerFactory")
    public void listenToCluster1(String message) {
        System.out.println("Received message from cluster 1: " + message);
    }

    @KafkaListener(topics = "topic2", containerFactory =
"cluster2KafkaListenerContainerFactory")
    public void listenToCluster2(String message) {
        System.out.println("Received message from cluster 2: " + message);
    }
}
```

Summary:

- **Configuration:** Separate configurations are defined for each Kafka cluster in `application.yml`.
- **Custom Factories:** Define custom `KafkaListenerContainerFactory` beans for each cluster.
- **Listeners:** Use different `containerFactory` attributes in the `@KafkaListener` annotations to specify which cluster's configuration to use.

This setup allows your Spring Boot application to interact with multiple Kafka clusters, each with its own configuration and set of topics.

ChatGPT can make mistakes. Check important info.