

Kafka Avro Deserialization Schema



Kafka AVRO DESERIALIZATION using SCHEMA REGISTRY



To handle Kafka Avro deserialization using the Confluent Schema Registry, you can follow these steps:

Prerequisites

- **Kafka cluster:** Ensure your Kafka cluster is running.
- **Schema Registry:** You must have a Schema Registry instance running. This will store your Avro schemas.

Dependencies

For a Maven project, add the following dependencies to your `pom.xml`:

```
xml

<dependencies>
  <!-- Kafka Clients -->
  <dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>3.0.0</version> <!-- Use the appropriate version -->
  </dependency>

  <!-- Confluent Schema Registry client -->
  <dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-avro-serializer</artifactId>
    <version>7.0.0</version> <!-- Use the appropriate version -->
  </dependency>

  <!-- Avro -->
  <dependency>
    <groupId>org.apache.avro</groupId>
    <artifactId>avro</artifactId>
    <version>1.11.0</version> <!-- Use the appropriate version -->
  </dependency>
</dependencies>
```

Avro Deserialization Configuration

1. **Set up Kafka Consumer Configuration with Schema Registry:** In your Kafka consumer configuration, you'll need to point to your Schema Registry and configure the consumer to use Avro for deserialization.

```
java
```

```
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import io.confluent.kafka.serializers.KafkaAvroDeserializer;
import java.util.Properties;

public class KafkaAvroConsumer {

    public static KafkaConsumer<String, Object> createConsumer() {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "avro-consumer-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
KafkaAvroDeserializer.class.getName());
        props.put("schema.registry.url", "http://localhost:8081");
        props.put("specific.avro.reader", "true"); // This ensures you deserialize to
specific classes

        return new KafkaConsumer<>(props);
    }
}
```

2. **Define the Avro Schema:** Your Avro schema will be stored in the Schema Registry. You can use the Confluent CLI or API to upload the schema.

Here's an example schema for a user:

```
json
```

```
{
  "type": "record",
  "name": "User",
  "namespace": "com.example",
  "fields": [
    { "name": "id", "type": "int" },
    { "name": "name", "type": "string" },
    { "name": "email", "type": "string" }
  ]
}
```

This schema will be stored in the Schema Registry and versioned.

3. **Consume Avro Messages:** Once the consumer is set up, you can start consuming Avro messages and deserialize them using the schema from the Schema Registry.

```
java
```

```
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import com.example.User; // Your Avro class generated from the schema

public class KafkaAvroConsumerExample {
    public static void main(String[] args) {
        KafkaConsumer<String, User> consumer = KafkaAvroConsumer.createConsumer();
        consumer.subscribe(List.of("avro-topic"));
    }
}
```

```
while (true) {
    ConsumerRecords<String, User> records =
    consumer.poll(Duration.ofMillis(100));

    for (ConsumerRecord<String, User> record : records) {
        User user = record.value();
        System.out.printf("Consumed record with ID %d, Name %s, and Email
        %s%n",
                                user.getId(), user.getName(), user.getEmail());
    }
}
```

Key Points:

- **KafkaAvroDeserializer:** This deserializer fetches the schema from the Schema Registry and applies it to deserialize Avro-encoded messages.
- ``schema.registry.url``: This property points to the Schema Registry instance.
- ``specific.avro.reader``: This flag, when set to ``true``, ensures that Avro messages are deserialized into specific Avro classes rather than generic ones.

Schema Registry Setup

If you haven't set up your schema registry, you can start it like this (assuming you're using Docker):

bash

```
docker run -d --name schema-registry \
-p 8081:8081 \
-e SCHEMA_REGISTRY_KAFKASTORE_BOOTSTRAP_SERVERS=PLAINTEXT://localhost:9092 \
-e SCHEMA_REGISTRY_HOST_NAME=schema-registry \
-e SCHEMA_REGISTRY_LISTENERS=http://0.0.0.0:8081 \
confluentinc/cp-schema-registry:latest
```

This setup will allow you to consume Kafka messages serialized with Avro using the Confluent Schema Registry for schema management.



Can we generalize or generic



Yes, you can generalize or make the Kafka Avro deserialization process more generic. This is useful when you need to handle multiple Avro schemas or topics, and you don't want to create a specific deserializer or consumer for each schema type.

Generic Avro Deserialization

Instead of using the specific Avro class like ``User``, you can use a more general approach, which involves deserializing into ``GenericRecord``. This will allow you to work with any Avro schema

dynamically.

Here's how you can achieve it:

1. Update Dependencies

Make sure you have the same dependencies for Avro and the Confluent Schema Registry.

2. Modify the Consumer Configuration

Instead of setting the deserializer to a specific Avro class, you will use `GenericRecord` to deserialize any Avro-encoded message.

java

```
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import io.confluent.kafka.serializers.KafkaAvroDeserializer;
import org.apache.avro.generic.GenericRecord;
import java.util.Properties;

public class GenericKafkaAvroConsumer {

    public static KafkaConsumer<String, GenericRecord> createConsumer() {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "generic-avro-consumer-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
KafkaAvroDeserializer.class.getName());
        props.put("schema.registry.url", "http://localhost:8081");
        // Do not set `specific.avro.reader` so it defaults to GenericRecord
        return new KafkaConsumer<>(props);
    }
}
```

3. Consume and Handle Generic Avro Records

You can now consume any Avro-encoded message as a `GenericRecord`. This approach allows flexibility, as you can handle different schema types dynamically without relying on specific classes.

java

```
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.avro.generic.GenericRecord;
import java.time.Duration;
import java.util.Collections;

public class GenericAvroConsumerExample {
    public static void main(String[] args) {
        KafkaConsumer<String, GenericRecord> consumer =
GenericKafkaAvroConsumer.createConsumer();
        consumer.subscribe(Collections.singletonList("avro-topic"));

        while (true) {
            ConsumerRecords<String, GenericRecord> records =
consumer.poll(Duration.ofMillis(100));
        }
    }
}
```

```

for (ConsumerRecord<String, GenericRecord> record : records) {
    GenericRecord genericRecord = record.value();
    System.out.println("Consumed record: " + genericRecord);

    // You can access the fields of the record by name, e.g.:
    if (genericRecord.getSchema().getField("id") != null) {
        System.out.println("ID: " + genericRecord.get("id"));
    }
    if (genericRecord.getSchema().getField("name") != null) {
        System.out.println("Name: " + genericRecord.get("name"));
    }
    if (genericRecord.getSchema().getField("email") != null) {
        System.out.println("Email: " + genericRecord.get("email"));
    }
}
}
}
}
}

```

4. Explanation of Key Points:

- **GenericRecord:** This is a generic representation of Avro records. It doesn't bind to a specific Avro schema class, which means it can handle any schema, as long as it's available in the Schema Registry.
- **Dynamic Schema Handling:** The `GenericRecord` allows you to access the fields dynamically by their names, which makes this approach flexible for different schemas.
- **Schema Registry:** The Schema Registry ensures that the consumer is aware of the correct schema version when deserializing messages.

Advantages:

- **Schema Flexibility:** You can handle multiple schemas without creating specific Avro classes for each one.
- **Dynamic Schema Evolution:** The deserialization is done based on the schema fetched from the Schema Registry, making it easy to evolve schemas.
- **Generic Handling:** You can process messages dynamically based on the fields present in the schema.

This setup provides a generalized approach to consuming Avro-encoded Kafka messages using Schema Registry without requiring specific schema classes.



```

if (genericRecord.getSchema().getField("id") != null) {
    System.out.println("ID: " + genericRecord.get("id"));
}
if (genericRecord.getSchema().getField("name") != null) {
    System.out.println("Name: " + genericRecord.get("name"));
}
if (genericRecord.getSchema().getField("email") != null) {
    System.out.println("Email: " + genericRecord.get("email"));
}
}

```

I don't know what are the field



If you don't know the fields in advance, you can make the processing of the `GenericRecord` more dynamic by iterating over the fields of the schema. This way, you don't need to know the field names beforehand, and you can print or process all the fields in the Avro record.

Here's how you can handle unknown fields dynamically:

Dynamic Field Handling

java

```
import org.apache.avro.generic.GenericRecord;
import org.apache.avro.Schema;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import java.time.Duration;
import java.util.List;

public class DynamicGenericAvroConsumerExample {
    public static void main(String[] args) {
        KafkaConsumer<String, GenericRecord> consumer =
            GenericKafkaAvroConsumer.createConsumer();
        consumer.subscribe(List.of("avro-topic"));

        while (true) {
            ConsumerRecords<String, GenericRecord> records =
                consumer.poll(Duration.ofMillis(100));

            for (ConsumerRecord<String, GenericRecord> record : records) {
                GenericRecord genericRecord = record.value();
                System.out.println("Consumed record:");

                // Iterate over all fields dynamically
                Schema schema = genericRecord.getSchema();
                for (Schema.Field field : schema.getFields()) {
                    String fieldName = field.name();
                    Object fieldValue = genericRecord.get(fieldName);
                    System.out.printf("Field name: %s, Field value: %s\n", fieldName,
                        fieldValue);
                }
            }
        }
    }
}
```

Explanation:

- `genericRecord.getSchema().getFields()`: This retrieves all the fields of the Avro schema dynamically.
- **Iterating over fields:** By looping through the fields, you can get the name of each field (`field.name()`) and its corresponding value from the `GenericRecord`.
- **Field type handling:** You could extend this by handling the field types (e.g., string, integer, etc.) dynamically if needed.

Benefits:

- **No Need to Know Fields in Advance:** This approach lets you process any Avro message regardless of the schema.
- **Extensible:** You can extend this to handle complex or nested fields as well.

This makes your consumer truly generic, as it works for any Avro message with any schema.



How can we transform GenericRecord to Json



To transform a `GenericRecord` into a JSON object, you can use Avro's built-in `GenericDatumWriter` along with the `JsonEncoder`. This will allow you to convert the `GenericRecord` into a JSON string.

Here's how you can achieve the transformation:

Steps to Convert `GenericRecord` to JSON:

1. **Set up Avro classes for JSON transformation:** You will need `GenericDatumWriter` to write the `GenericRecord` and `JsonEncoder` to encode it as JSON.
2. **Convert the `GenericRecord` to JSON:** Use the `JsonEncoder` to serialize the Avro data into a JSON string.

Code Example

Here's an example of how you can transform a `GenericRecord` to JSON:

java

```
import org.apache.avro.generic.GenericRecord;
import org.apache.avro.generic.GenericDatumWriter;
import org.apache.avro.io.Encoder;
import org.apache.avro.io.EncoderFactory;
import org.apache.avro.Schema;
import org.apache.avro.io.DatumWriter;
import java.io.ByteArrayOutputStream;
import java.io.IOException;

public class AvroToJsonConverter {

    public static String convertGenericRecordToJson(GenericRecord genericRecord) {
        ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
        try {
            // Get the schema of the record
            Schema schema = genericRecord.getSchema();

            // Prepare the Avro GenericDatumWriter for the schema
            DatumWriter<GenericRecord> datumWriter = new GenericDatumWriter<>(schema);

            // Create a JSON encoder to write the GenericRecord into JSON format
            Encoder jsonEncoder = EncoderFactory.get().jsonEncoder(schema, outputStream);

            // Write the GenericRecord to the JSON encoder
            datumWriter.write(genericRecord, jsonEncoder);

            // Flush the encoder to finish writing the JSON
            jsonEncoder.flush();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return outputStream.toString();
    }
}
```

```

        // Convert the output stream to a string and return
        return outputStream.toString();
    } catch (IOException e) {
        e.printStackTrace();
        return null;
    }
}

```

Example Usage in Kafka Consumer:

You can integrate this into the Kafka consumer that processes `GenericRecord` messages:

java

```

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.avro.generic.GenericRecord;
import java.time.Duration;
import java.util.List;

public class GenericKafkaAvroConsumerExample {

    public static void main(String[] args) {
        KafkaConsumer<String, GenericRecord> consumer =
            GenericKafkaAvroConsumer.createConsumer();
        consumer.subscribe(List.of("avro-topic"));

        while (true) {
            ConsumerRecords<String, GenericRecord> records =
                consumer.poll(Duration.ofMillis(100));

            for (ConsumerRecord<String, GenericRecord> record : records) {
                GenericRecord genericRecord = record.value();
                System.out.println("Consumed record as GenericRecord:");

                // Convert the GenericRecord to JSON
                String jsonRecord =
                    AvroToJsonConverter.convertGenericRecordToJson(genericRecord);
                System.out.println("Converted record to JSON: " + jsonRecord);
            }
        }
    }
}

```

Explanation:

1. `GenericDatumWriter`: This writes the `GenericRecord` data.
2. `JsonEncoder`: Encodes the data into a JSON format.
3. `outputStream.toString()`: Converts the JSON byte stream into a string.

Example Output:

For an Avro message with a schema like:

json

```

{
  "type": "record",
  "name": "User",

```



```

    "fields": [
      { "name": "id", "type": "int" },
      { "name": "name", "type": "string" },
      { "name": "email", "type": "string" }
    ]
  }

```

The output JSON might look like this:

json

```

{
  "id": 123,
  "name": "John Doe",
  "email": "john.doe@example.com"
}

```

Advantages:

- **Flexible Transformation:** You can now handle any Avro `GenericRecord` and convert it into a JSON string.
- **Schema-Driven:** The transformation process automatically handles the schema, so you don't need to know field names in advance.

This approach can be particularly useful when working with dynamic Avro messages from Kafka and transforming them for use in systems that expect JSON.



Can you make above in Spring Boot



Yes, you can integrate this Avro-to-JSON conversion process into a Spring Boot application. Below is an example of how you could set this up, including the necessary Kafka configuration and a service for processing `GenericRecord` from Kafka and converting it to JSON.

Steps:

1. **Kafka Configuration** in Spring Boot.
2. **Kafka Listener** to consume Avro messages.
3. **Service** to convert `GenericRecord` to JSON.
4. **Controller (optional)** to expose the converted JSON data if needed.

1. Kafka Configuration in Spring Boot

First, define the necessary Kafka configuration for deserializing Avro messages with Schema Registry.

`application.properties`

properties

```
# Kafka properties
spring.kafka.bootstrap-servers=localhost:9092
spring.kafka.consumer.group-id=avro-consumer-group
spring.kafka.consumer.key-
deserializer=org.apache.kafka.common.serialization.StringDeserializer
spring.kafka.consumer.value-
deserializer=io.confluent.kafka.serializers.KafkaAvroDeserializer
spring.kafka.properties.schema.registry.url=http://localhost:8081
spring.kafka.consumer.auto-offset-reset=earliest
```

2. Kafka Configuration Class

Create a configuration class to set up the Kafka consumer.

java

```
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import io.confluent.kafka.serializers.KafkaAvroDeserializer;
import org.apache.avro.generic.GenericRecord;
import org.springframework.context.annotation.Bean;
import org.springframework.kafka.annotation.EnableKafka;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.core.DefaultKafkaConsumerFactory;
import org.springframework.kafka.support.serializer.ErrorHandlingDeserializer;
import org.springframework.kafka.support.serializer.DeserializationException;
import org.springframework.kafka.support.serializer.JsonDeserializer;
import org.springframework.stereotype.Component;

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

@EnableKafka
@Component
public class KafkaConfig {

    @Bean
    public ConsumerFactory<String, GenericRecord> consumerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "generic-avro-consumer-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
KafkaAvroDeserializer.class.getName());
        props.put("schema.registry.url", "http://localhost:8081");
        return new DefaultKafkaConsumerFactory<>(props);
    }

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

3. Avro-to-JSON Service

This service will handle the conversion of `GenericRecord` to JSON.

```
java
```

```
import org.apache.avro.generic.GenericDatumWriter;
import org.apache.avro.generic.GenericRecord;
import org.apache.avro.io.Encoder;
import org.apache.avro.io.EncoderFactory;
import org.apache.avro.io.DatumWriter;
import org.apache.avro.Schema;
import org.springframework.stereotype.Service;

import java.io.ByteArrayOutputStream;
import java.io.IOException;

@Service
public class AvroToJsonConverterService {

    public String convertGenericRecordToJson(GenericRecord genericRecord) {
        ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
        try {
            // Get the schema of the record
            Schema schema = genericRecord.getSchema();

            // Prepare the Avro GenericDatumWriter for the schema
            DatumWriter<GenericRecord> datumWriter = new GenericDatumWriter<>(schema);

            // Create a JSON encoder to write the GenericRecord into JSON format
            Encoder jsonEncoder = EncoderFactory.get().jsonEncoder(schema, outputStream);

            // Write the GenericRecord to the JSON encoder
            datumWriter.write(genericRecord, jsonEncoder);

            // Flush the encoder to finish writing the JSON
            jsonEncoder.flush();

            // Convert the output stream to a string and return
            return outputStream.toString();
        } catch (IOException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

4. Kafka Listener

The listener will receive messages from Kafka, convert the `GenericRecord` to JSON using the service, and print or process it as needed.

```
java
```

```
import org.apache.avro.generic.GenericRecord;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Service;

@Service
public class KafkaAvroConsumerService {

    private final AvroToJsonConverterService avroToJsonConverterService;

    public KafkaAvroConsumerService(AvroToJsonConverterService avroToJsonConverterService) {
        this.avroToJsonConverterService = avroToJsonConverterService;
    }
}
```

```
@KafkaListener(topics = "avro-topic", groupId = "generic-avro-consumer-group")
public void consume(GenericRecord genericRecord) {
    System.out.println("Consumed record as GenericRecord: " + genericRecord);

    // Convert GenericRecord to JSON
    String jsonRecord =
        avroToJsonConverterService.convertGenericRecordToJson(genericRecord);
    System.out.println("Converted record to JSON: " + jsonRecord);

    // Further processing can be done here
}
}
```

5. Optional: Controller to Expose JSON Data

If you want to expose the consumed JSON via a REST endpoint, you can add a simple controller. However, this depends on your application's requirements.

java

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class KafkaJsonController {

    private final AvroToJsonConverterService avroToJsonConverterService;

    public KafkaJsonController(AvroToJsonConverterService avroToJsonConverterService) {
        this.avroToJsonConverterService = avroToJsonConverterService;
    }

    @GetMapping("/consume")
    public String getLatestJsonRecord() {
        // This should ideally return the latest Kafka message converted to JSON
        // You can integrate this with a message store or processing logic
        return "{\"message\": \"This is where the JSON data from Kafka would be returned\"}";
    }
}
```

Explanation:

- `KafkaListener`: Consumes messages from Kafka, dynamically deserializes Avro messages, and converts them to JSON.
- `AvroToJsonConverterService`: Handles the actual conversion from `GenericRecord` to JSON.
- `application.properties`: Configures Kafka to use the Schema Registry and deserializer.

Running the Application:

- Make sure your Kafka and Schema Registry are running.
- Produce Avro messages into the topic `avro-topic`.
- The listener will pick up the messages, convert them to JSON, and print them to the console.

This Spring Boot setup enables you to consume Avro messages dynamically from Kafka and transform them into JSON without needing to know the Avro schema in advance.



Without Annotation



Memory updated

To implement a Spring Boot Kafka consumer without annotations, you can manually configure the Kafka consumer and trigger the consumption of records. Here's how you can achieve this using a more programmatic approach, avoiding the use of `@KafkaListener`.

1. Kafka Consumer Setup without Annotations

You can create a `KafkaConsumer` manually and use it to poll messages in a separate thread. This method avoids the use of annotations.

Code Example:

1.1 Kafka Consumer Configuration Class

Define the Kafka consumer configuration programmatically using a factory method. This method returns a `KafkaConsumer` that can be used directly.

```
java
```

```
import org.apache.avro.generic.GenericRecord;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;
import io.confluent.kafka.serializers.KafkaAvroDeserializer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

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

@Configuration
public class KafkaConsumerConfig {

    @Bean
    public KafkaConsumer<String, GenericRecord> kafkaConsumer() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "generic-avro-consumer-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
KafkaAvroDeserializer.class.getName());
        props.put("schema.registry.url", "http://localhost:8081");

        // Create the KafkaConsumer
        return new KafkaConsumer<>(props);
    }
}
```

1.2 Kafka Consumer Service



Create a service

manually with Spring Boot including avRO JSON, Listener and factory



To configure a **Kafka consumer manually** without annotations in Spring Boot (including Avro deserialization to JSON, Kafka listener, and Kafka consumer factory), you can set up the Kafka consumer programmatically. This setup includes creating a consumer, polling messages, converting Avro to JSON, and running the listener in a custom thread.

Key Components:

1. **Kafka Consumer Configuration:** A manual consumer setup with Avro deserialization.
2. **Consumer Polling Logic:** Implement a service that polls Kafka manually and processes the messages.
3. **Avro to JSON Conversion:** Use a service to convert Avro messages to JSON format.
4. **Consumer Thread:** A dedicated thread to run the consumer manually.

Step-by-Step Code:

1. Kafka Consumer Configuration Class

Manually configure the Kafka consumer to read Avro messages from a topic.

```
java

import org.apache.avro.generic.GenericRecord;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;
import io.confluent.kafka.serializers.KafkaAvroDeserializer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

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

@Configuration
public class KafkaConsumerConfig {

    @Bean
    public KafkaConsumer<String, GenericRecord> kafkaConsumer() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "manual-consumer-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
KafkaAvroDeserializer.class.getName());
        props.put("schema.registry.url", "http://localhost:8081");
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        return new KafkaConsumer<>(props);
    }
}
```

2. Avro to JSON Conversion Service

Spring Boot



To configure a **Spring Boot Kafka consumer manually** (without annotations) to consume Avro messages and convert them to JSON, you can programmatically create a consumer, handle polling, and implement Avro-to-JSON conversion. Here's how you can set it up, including Kafka consumer configuration, a manual listener, and an Avro-to-JSON conversion service.

Components:

1. **Kafka Consumer Configuration:** Set up the Kafka consumer manually in Spring Boot.
2. **Polling Kafka Manually:** Run a listener manually to poll messages.
3. **Avro to JSON Converter:** Convert `GenericRecord` (Avro) to JSON.
4. **Thread Handling:** Poll Kafka in a separate thread to avoid blocking the main thread.

Step-by-Step Implementation:

1. Kafka Consumer Configuration Class

Define the Kafka consumer configuration manually, ensuring that the Avro deserialization is configured properly.

```
java
```

```
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import io.confluent.kafka.serializers.KafkaAvroDeserializer;
import org.apache.avro.generic.GenericRecord;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

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

@Configuration
public class KafkaConsumerConfig {

    @Bean
    public KafkaConsumer<String, GenericRecord> kafkaConsumer() {
        // Kafka consumer properties
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "manual-avro-consumer-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
KafkaAvroDeserializer.class.getName());
        props.put("schema.registry.url", "http://localhost:8081");
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        return new KafkaConsumer<>(props);
    }
}
```

2. Manual Polling Service

This service will manually poll Kafka for messages and process the `GenericRecord` data.

```
java
```

```
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.avro.generic.GenericRecord;
import org.springframework.stereotype.Service;

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

@Service
public class KafkaManualConsumerService {

    private final KafkaConsumer<String, GenericRecord> kafkaConsumer;
    private final AvroToJsonConverterService avroToJsonConverterService;

    public KafkaManualConsumerService(KafkaConsumer<String, GenericRecord> kafkaConsumer,
                                      AvroToJsonConverterService
avroToJsonConverterService) {
        this.kafkaConsumer = kafkaConsumer;
        this.avroToJsonConverterService = avroToJsonConverterService;
    }

    public void startConsuming() {
        // Subscribe to the topic
        kafkaConsumer.subscribe(Collections.singletonList("avro-topic"));

        // Poll messages in an infinite loop
        new Thread(() -> {
            while (true) {
                ConsumerRecords<String, GenericRecord> records =
kafkaConsumer.poll(Duration.ofMillis(1000));

                for (ConsumerRecord<String, GenericRecord> record : records) {
                    GenericRecord avroRecord = record.value();
                    System.out.println("Received Avro record: " + avroRecord);

                    // Convert Avro record to JSON
                    String json =
avroToJsonConverterService.convertGenericRecordToJson(avroRecord);
                    System.out.println("Converted to JSON: " + json);

                    // Additional processing can be added here
                }
            }
        }).start(); // Start the consumer thread
    }
}
```

3. Avro to JSON Converter Service

This service converts the Avro `GenericRecord` to JSON format using Avro's `GenericDatumWriter` and `JsonEncoder`.

```
java
```

```
import org.apache.avro.generic.GenericRecord;
import org.apache.avro.generic.GenericDatumWriter;
import org.apache.avro.io.Encoder;
import org.apache.avro.io.EncoderFactory;
```



```
import org.apache.avro.io.DatumWriter;
import org.apache.avro.Schema;
import org.springframework.stereotype.Service;

import java.io.ByteArrayOutputStream;
import java.io.IOException;

@Service
public class AvroToJsonConverterService {

    public String convertGenericRecordToJson(GenericRecord genericRecord) {
        ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
        try {
            Schema schema = genericRecord.getSchema();
            DatumWriter<GenericRecord> writer = new GenericDatumWriter<>(schema);
            Encoder jsonEncoder = EncoderFactory.get().jsonEncoder(schema, outputStream);

            writer.write(genericRecord, jsonEncoder);
            jsonEncoder.flush();

            return outputStream.toString();
        } catch (IOException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

4. Start the Kafka Consumer on Application Startup

In order to start the manual Kafka consumer when the application starts, you can use a Spring Boot `@PostConstruct` method to initiate the consumer.

```
java
```

```
import org.springframework.stereotype.Component;
import javax.annotation.PostConstruct;

@Component
public class KafkaConsumerInitializer {

    private final KafkaManualConsumerService kafkaManualConsumerService;

    public KafkaConsumerInitializer(KafkaManualConsumerService kafkaManualConsumerService) {
        this.kafkaManualConsumerService = kafkaManualConsumerService;
    }

    @PostConstruct
    public void startKafkaConsumer() {
        // Start consuming Kafka messages when the application starts
        kafkaManual
    }
}
```



Not Consumer , ConcurrentKafkaListenerContainerFactory



To manually configure a `ConcurrentKafkaListenerContainerFactory` in Spring Boot (without annotations) for consuming Avro messages and converting them to JSON, you can achieve this by setting up a Kafka consumer factory programmatically. This way, you avoid using `@KafkaListener` annotations, but you still benefit from Spring's Kafka listener container mechanism.

Steps:

1. **Configure Kafka Consumer Factory:** Define the consumer factory and the `ConcurrentKafkaListenerContainerFactory`.
2. **Manually Start Listener:** Use the factory to create listener containers manually, avoiding annotations.
3. **Avro to JSON Conversion:** Convert Avro `GenericRecord` to JSON as part of the message processing.

Step-by-Step Code:

1. Kafka Consumer Configuration Class

Define a Kafka consumer factory and configure the `ConcurrentKafkaListenerContainerFactory` manually.

```
java
```

```
import org.apache.avro.generic.GenericRecord;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import io.confluent.kafka.serializers.KafkaAvroDeserializer;
import org.apache.kafka.clients.consumer.ConsumerFactory;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.core.DefaultKafkaConsumerFactory;
import org.springframework.kafka.listener.ConcurrentMessageListenerContainer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

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

@Configuration
public class KafkaConsumerConfig {

    // Kafka consumer configuration
    @Bean
    public Map<String, Object> consumerConfigs() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "manual-consumer-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
KafkaAvroDeserializer.class);
        props.put("schema.registry.url", "http://localhost:8081");
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        return props;
    }

    // Kafka consumer factory using Avro deserialization
    @Bean
    public ConsumerFactory<String, GenericRecord> consumerFactory() {
        return new DefaultKafkaConsumerFactory<>(consumerConfigs());
    }
}
```

```
// Kafka Listener Container Factory configuration
@Bean
public ConcurrentKafkaListenerContainerFactory<String, GenericRecord>
kafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<String, GenericRecord> factory =
        new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory());
    return factory;
}
}
```

2. Kafka Listener Service with Custom Logic

Create a service to handle the consumption of Avro messages, converting them to JSON, and processing them manually without annotations.

```
java
```

```
import org.apache.avro.generic.GenericRecord;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.springframework.kafka.listener.MessageListener;
import org.springframework.kafka.listener.ConcurrentMessageListenerContainer;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.stereotype.Service;

import javax.annotation.PostConstruct;

@Service
public class KafkaManualListenerService {

    private final ConcurrentKafkaListenerContainerFactory<String, GenericRecord>
kafkaListenerContainerFactory;
    private final AvroToJsonConverterService avroToJsonConverterService;

    public KafkaManualListenerService(
        ConcurrentKafkaListenerContainerFactory<String, GenericRecord>
kafkaListenerContainerFactory,
        AvroToJsonConverterService avroToJsonConverterService) {
        this.kafkaListenerContainerFactory = kafkaListenerContainerFactory;
        this.avroToJsonConverterService = avroToJsonConverterService;
    }

    @PostConstruct
    public void startKafkaListener() {
        // Create and configure the listener container manually
        ConcurrentMessageListenerContainer<String, GenericRecord> container =
            kafkaListenerContainerFactory.createContainer("avro-topic");

        container.setupMessageListener((MessageListener<String, GenericRecord>) record ->
        {
            System.out.println("Received message: " + record.value());

            // Convert Avro record to JSON
            String json =
avroToJsonConverterService.convertGenericRecordToJson(record.value());
            System.out.println("Converted to JSON: " + json);

            // Further processing of the JSON message can be done here
        });

        // Start the container
        container.start();
    }
}
```

3. Avro to JSON Conversion Service

This service is responsible for converting Avro `GenericRecord` to JSON format.

```
java
```

```
import org.apache.avro.generic.GenericRecord;
import org.apache.avro.generic.GenericDatumWriter;
import org.apache.avro.io.Encoder;
import org.apache.avro.io.EncoderFactory;
import org.apache.avro.io.DatumWriter;
import org.apache.avro.Schema;
import org.springframework.stereotype.Service;

import java.io.ByteArrayOutputStream;
import java.io.IOException;

@Service
public class AvroToJsonConverterService {

    public String convertGenericRecordToJson(GenericRecord genericRecord) {
        ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
        try {
            Schema schema = genericRecord.getSchema();
            DatumWriter<GenericRecord> writer = new GenericDatumWriter<>(schema);
            Encoder jsonEncoder = EncoderFactory.get().jsonEncoder(schema, outputStream);

            writer.write(genericRecord, jsonEncoder);
            jsonEncoder.flush();

            return outputStream.toString();
        } catch (IOException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

4. Application Lifecycle Management

When Spring Boot starts, the `KafkaManualListenerService` will automatically start the Kafka listener in a thread using the `ConcurrentKafkaListenerContainerFactory` setup.

Summary:

- **No Annotations:** This approach manually sets up the `ConcurrentKafkaListenerContainerFactory` without `@KafkaListener` annotations.
- **Avro-to-JSON Conversion:** You are converting Avro `GenericRecord` messages to JSON using a custom service.
- **Manual Listener:** The listener is manually started using a listener container, providing full control over message consumption.



Thank You... Will you add a Health Actuator for kafka



To add a **Kafka health check** in Spring Boot using **Spring Boot Actuator**, you can extend the existing `HealthIndicator` interface provided by Spring Boot. This will allow you to monitor the health of your Kafka consumer or producer.

Steps:

1. **Add Actuator Dependency:** Ensure you have the Spring Boot Actuator dependency.
2. **Implement Kafka Health Indicator:** Manually create a Kafka health check by pinging the Kafka broker or checking the consumer's ability to poll.
3. **Expose Health Endpoint:** Use Actuator to expose a `/health` endpoint that includes the Kafka health status.

Step-by-Step Implementation:

1. Add Spring Boot Actuator Dependency

First, ensure the following dependency is included in your `pom.xml`.

```
xml
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

2. Kafka Health Indicator

You need to implement a custom `HealthIndicator` to check Kafka health. This can be done by checking if the Kafka broker is available, or by testing if a consumer can poll messages.

```
java
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;

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

@Component
public class KafkaHealthIndicator implements HealthIndicator {

    private final KafkaConsumer<String, String> kafkaConsumer;

    public KafkaHealthIndicator(KafkaConsumer<String, String> kafkaConsumer) {
        this.kafkaConsumer = kafkaConsumer;
    }

    @Override
    public Health health() {
        try {
            // Attempt to subscribe to a topic and poll to check health
            kafkaConsumer.subscribe(Collections.singletonList("test-topic"));
            ConsumerRecords<String, String> records =
            kafkaConsumer.poll(Duration.ofMillis(1000));
```

```
// If no exception is thrown, Kafka is considered healthy
if (records != null) {
    return Health.up().withDetail("Kafka", "Kafka is reachable and healthy").build();
} else {
    return Health.down().withDetail("Kafka", "No records found").build();
}
} catch (Exception e) {
    // If there's an exception, Kafka is considered down
    return Health.down(e).withDetail("Kafka", "Kafka is not reachable").build();
}
}
```

This health indicator will attempt to poll a Kafka topic (`test-topic` in this case) to check the health of the Kafka cluster.

3. Expose Actuator Health Endpoints

In your `application.properties`, enable the Actuator health endpoints:

```
properties

management.endpoints.web.exposure.include=health
management.endpoint.health.show-details=always
```

This will expose the `/health` endpoint with detailed information, including the status of your Kafka health indicator.

4. Monitor Kafka Health via Actuator

Now, if you access `/actuator/health`, you'll see a detailed health status of your Kafka connection.

For example:

```
json

{
  "status": "UP",
  "components": {
    "kafka": {
      "status": "UP",
      "details": {
        "Kafka": "Kafka is reachable and healthy"
      }
    }
  }
}
```

If Kafka becomes unreachable, the status will change to `DOWN` with additional details.

Summary:

- You've created a custom Kafka health indicator to check Kafka's availability by polling a topic.

- The Kafka health status is exposed via Spring Boot Actuator under the `/actuator/health` endpoint.



Add support for more than one Kafka



To add support for monitoring multiple Kafka clusters using a custom `HealthIndicator` in Spring Boot, you can modify the implementation to take a list of Kafka consumers (one for each cluster) and check their health individually. Here's how to do it:

Steps:

1. **Create a Configuration for Multiple Kafka Consumers:** Define beans for each Kafka consumer to connect to different clusters.
2. **Modify the Kafka Health Indicator:** Update the `HealthIndicator` to loop through each consumer and check their health.
3. **Expose the Health Endpoint:** The health endpoint will now return the status of each Kafka cluster.

Step-by-Step Implementation:

1. Kafka Consumer Configuration for Multiple Clusters

First, create a configuration class to define multiple Kafka consumers.

```
java
```

```
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

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

@Configuration
public class KafkaMultipleConsumerConfig {

    // Define the first Kafka consumer
    @Bean(name = "kafkaConsumer1")
    public KafkaConsumer<String, String> kafkaConsumer1() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "manual-consumer-group-1");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        return new KafkaConsumer<>(props);
    }

    // Define the second Kafka consumer
    @Bean(name = "kafkaConsumer2")
```

```

public KafkaConsumer<String, String> kafkaConsumer2() {
    Map<String, Object> props = new HashMap<>();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9093");
    props.put(ConsumerConfig.GROUP_ID_CONFIG, "manual-consumer-group-2");
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
    props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
    return new KafkaConsumer<>(props);
}

// Add more consumers as needed...
}

```

This configuration class defines two Kafka consumers, `kafkaConsumer1` and `kafkaConsumer2`, each connecting to a different Kafka cluster.

2. Modify Kafka Health Indicator for Multiple Clusters

Update the `KafkaHealthIndicator` to check the health of each consumer.

java

```

import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;

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

@Component
public class KafkaHealthIndicator implements HealthIndicator {

    private final List<KafkaConsumer<String, String>> kafkaConsumers;

    public KafkaHealthIndicator(List<KafkaConsumer<String, String>> kafkaConsumers) {
        this.kafkaConsumers = kafkaConsumers;
    }

    @Override
    public Health health() {
        Health.Builder healthBuilder = Health.up();

        for (KafkaConsumer<String, String> consumer : kafkaConsumers) {
            try {
                // Attempt to subscribe to a test topic and poll to check health
                consumer.subscribe(Collections.singletonList("test-topic"));
                ConsumerRecords<String, String> records =
consumer.poll(Duration.ofMillis(1000));

                if (records.isEmpty()) {
                    healthBuilder.down().withDetail("Kafka Cluster", "No records found");
                } else {
                    healthBuilder.withDetail("Kafka Cluster", "Kafka is reachable and
healthy");
                }
            } catch (Exception e) {
                // If there's an exception, Kafka is considered down
                healthBuilder.down(e).withDetail("Kafka Cluster", "Kafka is not
reachable");
            } finally {
                consumer.unsubscribe(); // Unsubscribe to avoid leaving the consumer
subscribed
            }
        }
    }
}

```



```

    }
    return healthBuilder.build();
  }
}

```

3. Expose Actuator Health Endpoints

Ensure your `application.properties` includes Actuator settings, as shown previously.

4. Usage Example

With this setup, when you access the `/actuator/health` endpoint, you'll receive the health status for each Kafka cluster being monitored.

json

```

{
  "status": "UP",
  "components": {
    "kafkaCluster1": {
      "status": "UP",
      "details": {
        "Kafka Cluster": "Kafka is reachable and healthy"
      }
    },
    "kafkaCluster2": {
      "status": "DOWN",
      "details": {
        "Kafka Cluster": "Kafka is not reachable"
      }
    }
  }
}

```

Summary:

- **Multiple Kafka Consumers:** You have configured multiple Kafka consumers, each connecting to a different Kafka cluster.
- **Health Check for Each Cluster:** The health indicator checks the status of each cluster and reports their individual health.
- **Exposed Health Endpoint:** The `/actuator/health` endpoint shows detailed health status for each Kafka cluster.

ChatGPT can make mistakes. Check important info.