

Determining Kafka Deserializer Type



how to know type of kafka message deserializer



To determine the type of Kafka message deserializer, you need to check the Kafka consumer configuration. Kafka uses deserializer classes to convert the byte stream from Kafka into the desired object types for both the key and the value.

Here are some ways to know or specify the type of Kafka message deserializer:

1. Checking the Consumer Configuration

In your Kafka consumer configuration, you'll typically define the deserializer for the key and the value. These configurations can be found in your `application.properties`, `application.yml`, or directly in the Java code.

- For a properties file:

properties

```
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer
value.deserializer=org.apache.kafka.common.serialization.StringDeserializer
```

- For YAML:

yaml

```
kafka:
  consumer:
    key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
    value-deserializer: org.apache.kafka.common.serialization.StringDeserializer
```

- For Java configuration:

java

```
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
```

The deserializer class will tell you what type of message is expected:

- `StringDeserializer`: for string data

- ``ByteArrayDeserializer``: for raw byte arrays
- ``IntegerDeserializer``: for integer data
- Custom deserializer: if you've implemented a custom deserializer, you can check the class.

2. Checking the Message Schema

If you are using Confluent Schema Registry, it might be using Avro or Protobuf serialized messages. In this case, the deserializer would be something like:

- ``io.confluent.kafka.serializers.KafkaAvroDeserializer``
- ``io.confluent.kafka.serializers.protobuf.KafkaProtobufDeserializer``

3. Inspecting the Message Payload

If you have access to the messages and don't know the deserializer type, you can inspect the payload manually. Tools like Kafka Console Consumer or Kafka UI tools can help:

```
bash

kafka-console-consumer --bootstrap-server <broker> --topic <topic> --from-beginning
```

This will give you raw data, and based on the structure, you can infer whether it's a string, Avro, JSON, or some custom format.

Let me know if you need help with any specific deserializer setup.



Whether header have information



Kafka headers can carry additional metadata along with the message but typically do not contain information about the deserializer used for the key or value. However, **custom implementations** can include deserializer details in the headers if configured that way.

Here's what you need to know about Kafka headers and their potential role in deserialization:

1. Standard Kafka Headers

Kafka headers are key-value pairs attached to messages. You can add arbitrary metadata, such as content type, message version, or schema details, which could include deserialization information **if you set it up to do so**. Kafka itself does not automatically populate headers with deserializer information.

2. Custom Headers with Deserializer Information

If you are working with multiple serialization formats or deserializers, you could design your producer to include deserialization information in the message headers. For example, you could add a header specifying the data format (like Avro, JSON, etc.) or the deserializer class name.

Example of adding a custom header to Kafka:

```
java
```

```
ProducerRecord<String, String> record = new ProducerRecord<>(
    "topic_name",
    "key",
    "value"
);

record.headers().add("deserializer",
    "io.confluent.kafka.serializers.KafkaAvroDeserializer".getBytes());
producer.send(record);
```

In this example, the producer adds the deserializer information to the headers. The consumer can then inspect the headers to decide which deserializer to use:

```
java
```

```
Headers headers = record.headers();
Header deserializerHeader = headers.lastHeader("deserializer");
String deserializerName = new String(deserializerHeader.value(), StandardCharsets.UTF_8);
```

3. Confluent Schema Registry

If you are using Confluent Schema Registry, some metadata related to the schema may be present in the message itself, particularly when using formats like Avro, Protobuf, or JSON Schema. This metadata can be helpful for determining the deserializer but is **not present in the headers directly**. Instead, the first few bytes of the message payload contain schema identifiers that can be resolved via the registry.

4. Accessing Kafka Headers in Consumer

You can access the headers in a Kafka consumer to examine any custom or existing header information:

```
java
```

```
ConsumerRecord<String, String> record = ...;
Headers headers = record.headers();
for (Header header : headers) {
    System.out.printf("Header Key: %s, Value: %s\n", header.key(), new
String(header.value(), StandardCharsets.UTF_8));
}
```

5. Using Headers for Schema or Deserializer Information

If your system is complex and supports multiple data formats, you can enforce a practice where the producer adds headers with information about the content type, version, or deserializer. This could be a useful pattern when consuming data from topics that can carry different types of messages. Let me know if you're interested in setting up custom headers for deserialization or if you're using a specific framework like Spring Kafka.



Confluent Schema Registry



Confluent Schema Registry is a service that manages schemas for data serialized in formats such as Avro, Protobuf, or JSON Schema, which are widely used in Kafka. It allows producers and consumers to share schemas, ensuring compatibility between them without including the schema with every message.

Here's an overview of how it works and how deserialization fits in when using the Schema Registry.

1. Schema Registry Overview

- **Centralized Schema Management:** It stores and provides access to schemas used in Kafka messages. Producers register schemas when sending messages, and consumers retrieve schemas to deserialize messages correctly.
- **Schema Evolution:** It supports schema versioning and ensures compatibility (backward, forward, or full compatibility) between different versions of a schema.
- **Format Support:** It supports Avro, Protobuf, and JSON Schema formats.

2. How it Works

When using Confluent Schema Registry, the Kafka producer serializes data into a binary format using Avro, Protobuf, or JSON Schema and registers the schema with the Schema Registry. The message payload consists of the serialized data, but the first few bytes contain the schema identifier (schema ID) that can be used to fetch the correct schema from the registry.

- **Producers:** Serialize data using Avro/Protobuf/JSON and send it to Kafka along with a schema ID in the message.
- **Consumers:** Deserialize data by fetching the schema using the schema ID embedded in the message.

3. Message Format with Schema Registry

When a Kafka producer sends a message with the Confluent Schema Registry, the serialized message has the following format:

```
java
```

```
Magic Byte (1 byte) | Schema ID (4 bytes) | Serialized Data
```

- **Magic Byte:** A fixed byte (`0x00`) that indicates the message is serialized using a schema.
- **Schema ID:** A 4-byte integer that corresponds to the schema in the Schema Registry.
- **Serialized Data:** The actual serialized data based on the schema.

The **Schema ID** helps the consumer retrieve the correct schema from the registry to deserialize the data.

4. Producer Configuration

To use the Confluent Schema Registry with a Kafka producer, you need to configure the producer to serialize data using a specific schema format, such as Avro, Protobuf, or JSON Schema.

Example of configuring a producer with Avro serialization:

```
java
```

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "io.confluent.kafka.serializers.KafkaAvroSerializer");
props.put("schema.registry.url", "http://localhost:8081");

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

In this example:

- `value.serializer` is set to `KafkaAvroSerializer`, which is responsible for serializing the message using Avro.
- `schema.registry.url` is the URL of the Schema Registry service.

5. Consumer Configuration

The Kafka consumer needs to be configured to use the appropriate deserializer (e.g., Avro, Protobuf) and connect to the Schema Registry to retrieve the schema.

Example of configuring a consumer to use Avro deserialization:

```
java
```

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "consumer-group");
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", "io.confluent.kafka.serializers.KafkaAvroDeserializer");
props.put("schema.registry.url", "http://localhost:8081");

KafkaConsumer<String, GenericRecord> consumer = new KafkaConsumer<>(props);
```

In this case:

- `value.deserializer` is set to `KafkaAvroDeserializer`, which fetches the schema from the Schema Registry using the schema ID from the message and deserializes the data accordingly.

6. Handling Schema Evolution

Confluent Schema Registry supports schema evolution. You can define compatibility rules, such as:

- **Backward compatibility:** New consumers can read data produced with older schemas.
- **Forward compatibility:** Older consumers can read data produced with newer schemas.
- **Full compatibility:** Both backward and forward compatibility are enforced.

These compatibility rules prevent breaking changes and ensure smooth schema updates.

7. Schema Registry REST API

Schema Registry provides a REST API for operations such as registering, retrieving, and validating schemas. Common endpoints include:

- **Registering a Schema:** ``/subjects/{subject}/versions``
- **Retrieving a Schema by ID:** ``/schemas/ids/{id}``
- **Checking Compatibility:** ``/compatibility/subjects/{subject}/versions/{version}``

8. Deserializing Data in Consumers

When a Kafka message is consumed, the consumer fetches the schema ID from the message's header and retrieves the corresponding schema from the Schema Registry. This schema is used to deserialize the binary data into an Avro/Protobuf/JSON object.

Here's a simple deserialization flow:

- The consumer reads the Kafka message.
- It extracts the schema ID from the message payload.
- The consumer fetches the schema using the Schema Registry client.
- It uses the schema to deserialize the data.

Let me know if you'd like more details on configuring the Schema Registry, handling schema evolution, or working with specific formats like Avro, Protobuf, or JSON Schema.



Could you please write the code

Here's a simple deserialization flow:

The consumer reads the Kafka message.
It extracts the schema ID from the message payload.
The consumer fetches the schema using the Schema Registry client.
It uses the schema to deserialize the data.

Assuming schema registry URL is hosted in another location



Here's a step-by-step implementation of a Kafka consumer that reads a message, extracts the schema ID from the payload, fetches the schema from an externally hosted Confluent Schema Registry, and deserializes the data using Avro.

We will assume that the messages are serialized using Avro and the schema is stored in the Confluent Schema Registry. We will use ``KafkaAvroDeserializer`` for deserialization.

Steps:

1. Read the Kafka message.
2. Extract the schema ID from the message payload.
3. Use the Schema Registry client to fetch the schema.
4. Deserialize the Avro data using the schema.

Maven Dependencies

First, add the necessary dependencies for Kafka and Avro deserialization, along with the Confluent Schema Registry client:

xml

```
<dependencies>
  <!-- Kafka clients -->
  <dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>3.0.0</version>
  </dependency>

  <!-- Confluent Avro deserializer -->
  <dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-avro-serializer</artifactId>
    <version>7.3.0</version>
  </dependency>

  <!-- Avro -->
  <dependency>
    <groupId>org.apache.avro</groupId>
    <artifactId>avro</artifactId>
    <version>1.10.2</version>
  </dependency>
</dependencies>
```

Kafka Consumer Code with Avro Deserialization

java

```
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import io.confluent.kafka.serializers.KafkaAvroDeserializer;
import org.apache.kafka.common.serialization.StringDeserializer;
import io.confluent.kafka.schemaregistry.client.CachedSchemaRegistryClient;
import io.confluent.kafka.schemaregistry.client.SchemaRegistryClient;
import io.confluent.kafka.schemaregistry.client.rest.exceptions.RestClientException;
import org.apache.avro.generic.GenericRecord;
import org.apache.avro.io.DatumReader;
import org.apache.avro.io.Decoder;
import org.apache.avro.io.DecoderFactory;
import org.apache.avro.generic.GenericDatumReader;

import java.io.IOException;
import java.util.Collections;
import java.util.Properties;

public class KafkaAvroConsumer {

    public static void main(String[] args) throws IOException, RestClientException {
        // Define Kafka consumer properties
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "my-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
KafkaAvroDeserializer.class.getName());

        // Connect to an externally hosted Schema Registry
        String schemaRegistryUrl = "http://<external-schema-registry-url>:8081";
```

```

props.put("schema.registry.url", schemaRegistryUrl);

// Create Kafka consumer
KafkaConsumer<String, GenericRecord> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Collections.singletonList("my-avro-topic"));

// Schema Registry client for fetching schema by ID
SchemaRegistryClient schemaRegistryClient = new
CachedSchemaRegistryClient(schemaRegistryUrl, 100);

try {
    while (true) {
        for (ConsumerRecord<String, GenericRecord> record : consumer.poll(1000)) {
            // Deserialize Avro data using the fetched schema

            // Get the raw message payload
            byte[] messagePayload = (byte[]) record.value();

            // Extract the schema ID from the message (bytes 1-4 after the magic
byte)

            int schemaId = getSchemaIdFromMessage(messagePayload);

            // Fetch schema from Schema Registry
            org.apache.avro.Schema schema =
schemaRegistryClient.getById(schemaId);

            // Deserialize message using the schema
            DatumReader<GenericRecord> reader = new GenericDatumReader<>(schema);
            Decoder decoder = DecoderFactory.get().binaryDecoder(messagePayload,
null);

            GenericRecord avroRecord = reader.read(null, decoder);

            // Process the deserialized message
            System.out.println("Deserialized message: " + avroRecord);
        }
    }
} finally {
    consumer.close();
}

// Utility method to extract the schema ID from the message
private static int getSchemaIdFromMessage(byte[] messagePayload) {
    // The schema ID is stored in bytes 1-4 (after the magic byte 0x00)
    return ((messagePayload[1] & 0xFF) << 24)
        | ((messagePayload[2] & 0xFF) << 16)
        | ((messagePayload[3] & 0xFF) << 8)
        | (messagePayload[4] & 0xFF);
}
}

```

Key Points:

1. **Schema Registry Client:** The `CachedSchemaRegistryClient` is used to fetch the schema from the externally hosted Schema Registry. It caches schemas to reduce network calls.
2. **Deserialization:** We are using the `GenericDatumReader` to deserialize the Avro message using the schema fetched from the Schema Registry.
3. **Schema ID Extraction:** The schema ID is extracted from the first few bytes of the message, where:
 - Byte 0 is the **magic byte** (indicating the use of Schema Registry).
 - Bytes 1-4 represent the **schema ID**.
4. **Polling Kafka:** The consumer polls messages from Kafka, extracts the schema ID, fetches the schema, and deserializes the message.

Assumptions:

- The messages are Avro serialized.
- The Schema Registry is hosted externally and is accessible via ``http://<external-schema-registry-url>:8081``.

You can modify the schema deserialization part if you're working with Protobuf or JSON Schema. Let me know if you need any adjustments!



Whether it is true for String, JSON



No, the described approach for Avro (using schema IDs and Schema Registry) is not directly applicable for **String** and **JSON** messages. The process for **String** and **JSON** deserialization does not involve the Confluent Schema Registry in the same way that Avro, Protobuf, or JSON Schema messages do. Here's how deserialization works for different formats:

1. String Deserialization

For string messages, Kafka simply treats the message payload as a sequence of bytes and directly converts them into a string. There's no need for a schema or schema ID.

- **No Schema Registry Involvement:** Since there's no schema involved, you don't need a Schema Registry to serialize or deserialize the data.
- **Deserialization Example:**
 - **Producer** sends a string.
 - **Consumer** reads the message and uses ``StringDeserializer`` to convert the byte array into a string.

Example Code for String Deserialization:

java

```
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;

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

public class KafkaStringConsumer {

    public static void main(String[] args) {
        // Kafka consumer configuration settings
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "test-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());

        // Create a new Kafka consumer
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
```

```
// Subscribe to the topic
consumer.subscribe(Collections.singletonList("my-string-topic"));

// Poll for new messages
while (true) {
    for (ConsumerRecord<String, String> record : consumer.poll(1000)) {
        // Process the message
        System.out.println("Received message: " + record.value());
    }
}
}
```

2. JSON Deserialization

For JSON messages, deserialization is typically done by converting the byte array (message payload) into a JSON string, and then parsing it into a JSON object or a Java class.

- **No Schema Registry Involvement:** Unless you're using **Confluent's JSON Schema** support (which is different from plain JSON), there's no schema registry involved in plain JSON serialization or deserialization.
- **JSON Deserialization Example:**
 - **Producer** sends a JSON string.
 - **Consumer** reads the message, deserializes it into a string, and then parses the JSON string using a JSON library like Jackson or Gson.

Example Code for JSON Deserialization:

java

```
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;

import java.io.IOException;
import java.util.Collections;
import java.util.Properties;

public class KafkaJsonConsumer {

    public static void main(String[] args) throws IOException {
        // Kafka consumer configuration settings
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "test-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());

        // Create a new Kafka consumer
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

        // Subscribe to the topic
        consumer.subscribe(Collections.singletonList("my-json-topic"));

        ObjectMapper objectMapper = new ObjectMapper();

        // Poll for new messages
        while (true) {
            for (ConsumerRecord<String, String> record : consumer.poll(1000)) {
```

```
// Deserialize the JSON string
String jsonString = record.value();
JsonNode jsonNode = objectMapper.readTree(jsonString);

// Process the deserialized JSON
System.out.println("Received JSON: " + jsonNode.toString());
}
}
}
```

Summary of Differences:

- **String Deserialization:**
 - Directly converts the byte array to a string.
 - No schema or Schema Registry is involved.
- **JSON Deserialization:**
 - Converts the byte array to a string, and then parses it into a JSON object.
 - No schema or Schema Registry is involved, unless you use JSON Schema (Confluent JSON Schema).
- **Avro/Protobuf/JSON Schema Deserialization:**
 - Requires the Schema Registry to fetch the schema using the schema ID stored in the message.
 - Deserializes the binary payload based on the fetched schema.

For **String** and **plain JSON** messages, the Schema Registry is not needed unless you are explicitly using **Confluent's JSON Schema** (a formal schema format).

Let me know if you need more information on JSON Schema or further clarification!



Can you bundle everything



Sure! Here's a bundled version that demonstrates deserialization for three types of Kafka message formats: **String**, **plain JSON**, and **Avro with Schema Registry**. The key difference is how each format is handled and whether the Schema Registry is involved.

Project Setup

Make sure to include the required dependencies for Kafka clients, Avro deserialization, and JSON handling.

Maven Dependencies:

xml

```
<dependencies>
  <!-- Kafka clients -->
  <dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
```

```

        <version>3.0.0</version>
    </dependency>

    <!-- Confluent Avro deserializer -->
    <dependency>
        <groupId>io.confluent</groupId>
        <artifactId>kafka-avro-serializer</artifactId>
        <version>7.3.0</version>
    </dependency>

    <!-- Jackson JSON library -->
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.13.0</version>
    </dependency>

    <!-- Avro -->
    <dependency>
        <groupId>org.apache.avro</groupId>
        <artifactId>avro</artifactId>
        <version>1.10.2</version>
    </dependency>
</dependencies>

```

Kafka Consumers for String, JSON, and Avro

This example provides consumers for three types of messages: **String**, **JSON**, and **Avro (with Schema Registry)**.

1. String Message Consumer:

This consumer simply reads messages and deserializes them as strings without any schema management.

```
java
```

```

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;

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

public class KafkaStringConsumer {

    public static void main(String[] args) {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "string-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());

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

        while (true) {
            for (ConsumerRecord<String, String> record : consumer.poll(1000)) {
                System.out.println("Received String message: " + record.value());
            }
        }
    }
}

```

```
}
}
```

2. JSON Message Consumer:

This consumer reads messages as strings and then parses them into a JSON object using Jackson.

```
java
```

```
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;

import java.io.IOException;
import java.util.Collections;
import java.util.Properties;

public class KafkaJsonConsumer {

    public static void main(String[] args) throws IOException {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "json-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());

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

        ObjectMapper objectMapper = new ObjectMapper();

        while (true) {
            for (ConsumerRecord<String, String> record : consumer.poll(1000)) {
                String jsonString = record.value();
                JsonNode jsonNode = objectMapper.readTree(jsonString);
                System.out.println("Received JSON message: " + jsonNode.toPrettyString());
            }
        }
    }
}
```

3. Avro with Schema Registry Consumer:

This consumer reads messages that were serialized using Avro and fetches the schema from an externally hosted Confluent Schema Registry. It uses the schema to deserialize the Avro binary payload.

```
java
```

```
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import io.confluent.kafka.serializers.KafkaAvroDeserializer;
import org.apache.kafka.common.serialization.StringDeserializer;
import io.confluent.kafka.schemaregistry.client.CachedSchemaRegistryClient;
import io.confluent.kafka.schemaregistry.client.SchemaRegistryClient;
import org.apache.avro.generic.GenericRecord;
```

```
import org.apache.avro.io.DatumReader;
import org.apache.avro.io.Decoder;
import org.apache.avro.io.DecoderFactory;
import org.apache.avro.generic.GenericDatumReader;

import java.io.IOException;
import java.util.Collections;
import java.util.Properties;

public class KafkaAvroConsumer {

    public static void main(String[] args) throws IOException {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "avro-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://<external-schema-registry-url>:8081");

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

        SchemaRegistryClient schemaRegistryClient = new
CachedSchemaRegistryClient("http://<external-schema-registry-url>:8081", 100);

        while (true) {
            for (ConsumerRecord<String, GenericRecord> record : consumer.poll(1000)) {
                byte[] messagePayload = (byte[]) record.value();
                int schemaId = getSchemaIdFromMessage(messagePayload);
                org.apache.avro.Schema schema = schemaRegistryClient.getById(schemaId);

                DatumReader<GenericRecord> reader = new GenericDatumReader<>(schema);
                Decoder decoder = DecoderFactory.get().binaryDecoder(messagePayload,
null);

                GenericRecord avroRecord = reader.read(null, decoder);

                System.out.println("Received Avro message: " + avroRecord);
            }
        }

        // Extract schema ID from the message payload
        private static int getSchemaIdFromMessage(byte[] messagePayload) {
            return ((messagePayload[1] & 0xFF) << 24)
                | ((messagePayload[2] & 0xFF) << 16)
                | ((messagePayload[3] & 0xFF) << 8)
                | (messagePayload[4] & 0xFF);
        }
    }
}
```

Summary of Key Differences:

1. String Messages:

- Deserialized directly as strings.
- No Schema Registry or additional parsing needed.

2. JSON Messages:

- Deserialized as strings, then parsed into JSON objects using Jackson.
- No Schema Registry required unless you're using **JSON Schema** from Confluent.

3. Avro Messages (with Schema Registry):

- Messages contain a schema ID.
- The schema ID is used to fetch the schema from the Schema Registry.
- The Avro binary data is deserialized using the fetched schema.

Conclusion

- **String and JSON** deserialization are simple and don't involve Schema Registry unless you're using a formal schema like **JSON Schema**.
- **Avro** messages rely on the Schema Registry to dynamically fetch schemas for deserialization.

Make sure to configure your Kafka brokers and Schema Registry URLs accordingly. Let me know if you need further clarification or more details on any part!



Bundle together that one class handle all



Certainly! Here's a combined Kafka consumer class that can handle **String**, **JSON**, and **Avro (with Schema Registry)** messages in one implementation. The consumer will check the topic and use the appropriate deserialization method based on the topic name or message content type. This solution assumes you are consuming from multiple topics, and each topic has a known message format (e.g., ``string-topic``, ``json-topic``, ``avro-topic``). The class will decide the deserialization mechanism based on topic names. You can easily modify the condition for a more flexible approach, such as using message headers to determine the content type.

Unified Kafka Consumer Class

java

```
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import io.confluent.kafka.schemaregistry.client.CachedSchemaRegistryClient;
import io.confluent.kafka.schemaregistry.client.SchemaRegistryClient;
import org.apache.avro.generic.GenericDatumReader;
import org.apache.avro.generic.GenericRecord;
import org.apache.avro.io.Decoder;
import org.apache.avro.io.DecoderFactory;
import org.apache.avro.io.DatumReader;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;
import io.confluent.kafka.serializers.KafkaAvroDeserializer;

import java.io.IOException;
import java.util.Collections;
import java.util.Properties;

public class UnifiedKafkaConsumer {

    private static final ObjectMapper objectMapper = new ObjectMapper();
    private static final String SCHEMA_REGISTRY_URL = "http://<external-schema-registry-url>:8081";

    public static void main(String[] args) throws IOException {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "unified-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());

        KafkaConsumer<String, Object> consumer = new KafkaConsumer<>(props);
```



```
// Subscribe to multiple topics
consumer.subscribe(Collections.singletonList("string-topic"));
consumer.subscribe(Collections.singletonList("json-topic"));
consumer.subscribe(Collections.singletonList("avro-topic"));

SchemaRegistryClient schemaRegistryClient = new
CachedSchemaRegistryClient(SCHEMA_REGISTRY_URL, 100);

while (true) {
    for (ConsumerRecord<String, Object> record : consumer.poll(1000)) {
        String topic = record.topic();

        if (topic.equals("string-topic")) {
            handleStringMessage(record);
        } else if (topic.equals("json-topic")) {
            handleJsonMessage(record);
        } else if (topic.equals("avro-topic")) {
            handleAvroMessage(record, schemaRegistryClient);
        }
    }
}

private static void handleStringMessage(ConsumerRecord<String, Object> record) {
    String message = (String) record.value();
    System.out.println("Received String message: " + message);
}

private static void handleJsonMessage(ConsumerRecord<String, Object> record) throws
IOException {
    String jsonString = (String) record.value();
    JsonNode jsonNode = objectMapper.readTree(jsonString);
    System.out.println("Received JSON message: " + jsonNode.toPrettyString());
}

private static void handleAvroMessage(ConsumerRecord<String, Object> record,
SchemaRegistryClient schemaRegistryClient) throws IOException {
    byte[] messagePayload = (byte[]) record.value();

    // Extract schema ID from the Avro message (stored in the payload)
    int schemaId = getSchemaIdFromMessage(messagePayload);
    org.apache.avro.Schema schema = schemaRegistryClient.getById(schemaId);

    DatumReader<GenericRecord> reader = new GenericDatumReader<>(schema);
    Decoder decoder = DecoderFactory.get().binaryDecoder(messagePayload, null);
    GenericRecord avroRecord = reader.read(null, decoder);

    System.out.println("Received Avro message: " + avroRecord);
}

// Extract schema ID from the message payload
private static int getSchemaIdFromMessage(byte[] messagePayload) {
    return ((messagePayload[1] & 0xFF) << 24)
        | ((messagePayload[2] & 0xFF) << 16)
        | ((messagePayload[3] & 0xFF) << 8)
        | (messagePayload[4] & 0xFF);
}
}
```

Key Points of the Unified Consumer:

1. Topic-based Deserialization:

- The `main` method subscribes to multiple topics (`string-topic`, `json-topic`, `avro-topic`) and uses topic names to determine which deserialization process to use.
- Based on the topic, it routes the message to the appropriate handler method (`handleStringMessage`, `handleJsonMessage`, `handleAvroMessage`).

2. String Messages:

- The `handleStringMessage` method simply casts the message payload to a string and prints it.

3. JSON Messages:

- The `handleJsonMessage` method converts the message payload to a string and then parses it into a `JsonNode` using Jackson's `ObjectMapper`.

4. Avro Messages with Schema Registry:

- The `handleAvroMessage` method extracts the schema ID from the message payload and fetches the schema from the Schema Registry using the `CachedSchemaRegistryClient`.
- The message payload is deserialized using the fetched schema and the Avro deserialization process.

Dependencies:

Make sure you have these dependencies in your **Maven** `pom.xml` or **Gradle** `build.gradle` file:

```
xml

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

  <!-- Confluent Avro deserializer and Schema Registry -->
  <dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-avro-serializer</artifactId>
    <version>7.3.0</version>
  </dependency>

  <!-- Jackson JSON library -->
  <dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.13.0</version>
  </dependency>

  <!-- Avro -->
  <dependency>
    <groupId>org.apache.avro</groupId>
    <artifactId>avro</artifactId>
    <version>1.10.2</version>
  </dependency>
</dependencies>
```

How It Works:

1. The consumer subscribes to multiple Kafka topics (each containing different message types).
2. Based on the topic name, it directs the message to the appropriate handler for deserialization.
3. **String** messages are handled by simple casting, **JSON** messages are deserialized using Jackson, and **Avro** messages are deserialized using the Schema Registry.

Next Steps:

You can easily extend this class to handle other message formats, or improve the deserialization selection mechanism by inspecting message headers or other metadata.

Let me know if you need further assistance!



include non-schema for AVRO



To handle both **Avro with Schema Registry** and **Avro without Schema Registry** in the same unified consumer class, you need to differentiate between messages that are serialized using a **schema stored in the Schema Registry** and those that include their schema directly in the payload (non-schema Avro).

Modifications to Handle Non-Schema Avro Messages

For non-schema Avro messages, the Avro message itself must include its schema. This means that the deserialization process should use the embedded schema, or a predefined schema, to deserialize the message.

Here's the updated class:

Updated Unified Kafka Consumer Class

java

```
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import io.confluent.kafka.schemaregistry.client.CachedSchemaRegistryClient;
import io.confluent.kafka.schemaregistry.client.SchemaRegistryClient;
import org.apache.avro.Schema;
import org.apache.avro.generic.GenericDatumReader;
import org.apache.avro.generic.GenericRecord;
import org.apache.avro.io.DatumReader;
import org.apache.avro.io.Decoder;
import org.apache.avro.io.DecoderFactory;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;

import java.io.IOException;
import java.util.Collections;
import java.util.Properties;

public class UnifiedKafkaConsumer {

    private static final ObjectMapper objectMapper = new ObjectMapper();
    private static final String SCHEMA_REGISTRY_URL = "http://<external-schema-registry-url>:8081";

    public static void main(String[] args) throws IOException {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "unified-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());

        KafkaConsumer<String, Object> consumer = new KafkaConsumer<>(props);

        // Subscribe to multiple topics
```

```

consumer.subscribe(Collections.singletonList("string-topic"));
consumer.subscribe(Collections.singletonList("json-topic"));
consumer.subscribe(Collections.singletonList("avro-schema-topic"));
consumer.subscribe(Collections.singletonList("avro-non-schema-topic"));

SchemaRegistryClient schemaRegistryClient = new
CachedSchemaRegistryClient(SCHEMA_REGISTRY_URL, 100);

while (true) {
    for (ConsumerRecord<String, Object> record : consumer.poll(1000)) {
        String topic = record.topic();

        if (topic.equals("string-topic")) {
            handleStringMessage(record);
        } else if (topic.equals("json-topic")) {
            handleJsonMessage(record);
        } else if (topic.equals("avro-schema-topic")) {
            handleAvroWithSchemaRegistryMessage(record, schemaRegistryClient);
        } else if (topic.equals("avro-non-schema-topic")) {
            handleNonSchemaAvroMessage(record);
        }
    }
}

private static void handleStringMessage(ConsumerRecord<String, Object> record) {
    String message = (String) record.value();
    System.out.println("Received String message: " + message);
}

private static void handleJsonMessage(ConsumerRecord<String, Object> record) throws
IOException {
    String jsonString = (String) record.value();
    JsonNode jsonNode = objectMapper.readTree(jsonString);
    System.out.println("Received JSON message: " + jsonNode.toPrettyString());
}

private static void handleAvroWithSchemaRegistryMessage(ConsumerRecord<String, Object>
record, SchemaRegistryClient schemaRegistryClient) throws IOException {
    byte[] messagePayload = (byte[]) record.value();

    // Extract schema ID from the Avro message (stored in the payload)
    int schemaId = getSchemaIdFromMessage(messagePayload);
    Schema schema = schemaRegistryClient.getById(schemaId);

    DatumReader<GenericRecord> reader = new GenericDatumReader<>(schema);
    Decoder decoder = DecoderFactory.get().binaryDecoder(messagePayload, null);
    GenericRecord avroRecord = reader.read(null, decoder);

    System.out.println("Received Avro message (with Schema Registry): " + avroRecord);
}

// Handling Non-Schema Avro Message (Schema embedded in payload or pre-known schema)
private static void handleNonSchemaAvroMessage(ConsumerRecord<String, Object> record)
throws IOException {
    byte[] messagePayload = (byte[]) record.value();

    // If schema is embedded in the Avro message or you have a predefined schema
    Schema schema = getPredefinedSchema(); // Use predefined or derived schema

    DatumReader<GenericRecord> reader = new GenericDatumReader<>(schema);
    Decoder decoder = DecoderFactory.get().binaryDecoder(messagePayload, null);
    GenericRecord avroRecord = reader.read(null, decoder);

    System.out.println("Received Avro message (Non-Schema): " + avroRecord);
}

// Mock method to provide a predefined Avro schema
private static Schema getPredefinedSchema() {
    // Define your schema here if not coming from Schema Registry
    String schemaStr = "{
        + \"type\": \"record\",
        + \"name\": \"User\",
        + \"fields\": [
        + {\"name\": \"name\", \"type\": \"string\"},

```

```

        + "{ \"name\": \"age\", \"type\": \"int\" }"
        + "]"
        + "}";

        return new Schema.Parser().parse(schemaStr);
    }

    // Extract schema ID from the message payload (for Schema Registry messages)
    private static int getSchemaIdFromMessage(byte[] messagePayload) {
        return ((messagePayload[1] & 0xFF) << 24)
            | ((messagePayload[2] & 0xFF) << 16)
            | ((messagePayload[3] & 0xFF) << 8)
            | (messagePayload[4] & 0xFF);
    }
}

```

Key Changes:

1. Avro with Schema Registry:

- This method (`handleAvroWithSchemaRegistryMessage`) uses the Schema Registry to fetch the schema via the schema ID embedded in the message.

2. Non-Schema Avro:

- The `handleNonSchemaAvroMessage` method handles messages where the Avro schema is either embedded in the message or predefined. This scenario assumes that you either know the schema beforehand (via a predefined schema) or the schema is carried in the payload.
- In this example, I've included a `getPredefinedSchema()` method that returns a basic schema for demonstration purposes. You can modify this to dynamically extract the schema from the payload or use a predefined one.

How It Works:

- The class subscribes to multiple topics.
- Based on the topic name:
 - **String** and **JSON** messages are deserialized using basic string and Jackson parsing.
 - **Avro with Schema Registry** messages use the schema fetched from the external Schema Registry.
 - **Non-Schema Avro** messages use a predefined schema to deserialize the message.

Example Avro Schemas:

For **non-schema Avro**, you can define your own Avro schema as needed. In this example, I've used a simple `User` schema.

json

```

{
  "type": "record",
  "name": "User",
  "fields": [
    { "name": "name", "type": "string" },
    { "name": "age", "type": "int" }
  ]
}

```

Dependencies:

Same as before, ensure you have the following dependencies for **Avro**, **Kafka**, and **Schema Registry**:

xml

```
<dependencies>
  <!-- Kafka clients -->
  <dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>3.0.0</version>
  </dependency>

  <!-- Confluent Avro deserializer and Schema Registry -->
  <dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-avro-serializer</artifactId>
    <version>7.3.0</version>
  </dependency>

  <!-- Jackson JSON library -->
  <dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.13.0</version>
  </dependency>

  <!-- Avro -->
  <dependency>
    <groupId>org.apache.avro</groupId>
    <artifactId>avro</artifactId>
    <version>1.10.2</version>
  </dependency>
</dependencies>
```

Conclusion:

This combined class can handle:

- **String** messages
- **JSON** messages
- **Avro with Schema Registry** (messages stored with a schema reference)
- **Avro without Schema Registry** (predefined or schema embedded in the message)

Let me know if you need more details or clarification!



Include Error Topic also



Certainly! If you want to include an **error handling** mechanism where any deserialization issues are sent to an **error topic**, you can add this functionality to the unified Kafka consumer class.

Unified Kafka Consumer with Error Handling

This implementation includes:

1. **Error Topic:** A separate Kafka topic where error messages will be sent if there is an issue with deserialization.
2. **Error Handling:** Exception handling during deserialization, which will publish error messages to the specified error topic.

Here's how you can enhance the unified Kafka consumer class to include error handling:

java

```
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import io.confluent.kafka.schemaregistry.client.CachedSchemaRegistryClient;
import io.confluent.kafka.schemaregistry.client.SchemaRegistryClient;
import org.apache.avro.Schema;
import org.apache.avro.generic.GenericDatumReader;
import org.apache.avro.generic.GenericRecord;
import org.apache.avro.io.DatumReader;
import org.apache.avro.io.Decoder;
import org.apache.avro.io.DecoderFactory;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.apache.kafka.common.serialization.StringSerializer;

import java.io.IOException;
import java.util.Collections;
import java.util.Properties;
import java.util.concurrent.Future;

public class UnifiedKafkaConsumer {

    private static final ObjectMapper objectMapper = new ObjectMapper();
    private static final String SCHEMA_REGISTRY_URL = "http://<external-schema-registry-url>:8081";
    private static final String ERROR_TOPIC = "error-topic";

    public static void main(String[] args) throws IOException {
        Properties consumerProps = new Properties();
        consumerProps.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        consumerProps.put(ConsumerConfig.GROUP_ID_CONFIG, "unified-group");
        consumerProps.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        consumerProps.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());

        KafkaConsumer<String, Object> consumer = new KafkaConsumer<>(consumerProps);

        // Subscribe to multiple topics
        consumer.subscribe(Collections.singletonList("string-topic"));
        consumer.subscribe(Collections.singletonList("json-topic"));
        consumer.subscribe(Collections.singletonList("avro-schema-topic"));
        consumer.subscribe(Collections.singletonList("avro-non-schema-topic"));

        SchemaRegistryClient schemaRegistryClient = new
CachedSchemaRegistryClient(SCHEMA_REGISTRY_URL, 100);

        // Initialize Kafka producer for error messages
        Properties producerProps = new Properties();
        producerProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        producerProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
        producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
```

```

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

while (true) {
    for (ConsumerRecord<String, Object> record : consumer.poll(1000)) {
        String topic = record.topic();

        try {
            if (topic.equals("string-topic")) {
                handleStringMessage(record);
            } else if (topic.equals("json-topic")) {
                handleJsonMessage(record);
            } else if (topic.equals("avro-schema-topic")) {
                handleAvroWithSchemaRegistryMessage(record, schemaRegistryClient);
            } else if (topic.equals("avro-non-schema-topic")) {
                handleNonSchemaAvroMessage(record);
            }
        } catch (Exception e) {
            // Send error details to error topic
            String errorMessage = "Error processing message from topic " + topic +
": " + e.getMessage();
            producer.send(new ProducerRecord<>(ERROR_TOPIC, errorMessage));
        }
    }
}

private static void handleStringMessage(ConsumerRecord<String, Object> record) {
    String message = (String) record.value();
    System.out.println("Received String message: " + message);
}

private static void handleJsonMessage(ConsumerRecord<String, Object> record) throws
IOException {
    String jsonString = (String) record.value();
    JsonNode jsonNode = objectMapper.readTree(jsonString);
    System.out.println("Received JSON message: " + jsonNode.toPrettyString());
}

private static void handleAvroWithSchemaRegistryMessage(ConsumerRecord<String, Object>
record, SchemaRegistryClient schemaRegistryClient) throws IOException {
    byte[] messagePayload = (byte[]) record.value();

    // Extract schema ID from the Avro message (stored in the payload)
    int schemaId = getSchemaIdFromMessage(messagePayload);
    Schema schema = schemaRegistryClient.getById(schemaId);

    DatumReader<GenericRecord> reader = new GenericDatumReader<>(schema);
    Decoder decoder = DecoderFactory.get().binaryDecoder(messagePayload, null);
    GenericRecord avroRecord = reader.read(null, decoder);

    System.out.println("Received Avro message (with Schema Registry): " + avroRecord);
}

// Handling Non-Schema Avro Message (Schema embedded in payload or pre-known schema)
private static void handleNonSchemaAvroMessage(ConsumerRecord<String, Object> record)
throws IOException {
    byte[] messagePayload = (byte[]) record.value();

    // If schema is embedded in the Avro message or you have a predefined schema
    Schema schema = getPredefinedSchema(); // Use predefined or derived schema

    DatumReader<GenericRecord> reader = new GenericDatumReader<>(schema);
    Decoder decoder = DecoderFactory.get().binaryDecoder(messagePayload, null);
    GenericRecord avroRecord = reader.read(null, decoder);

    System.out.println("Received Avro message (Non-Schema): " + avroRecord);
}

// Mock method to provide a predefined Avro schema
private static Schema getPredefinedSchema() {
    // Define your schema here if not coming from Schema Registry
    String schemaStr = "{"
        + "\"type\": \"record\", \"
        + "\"name\": \"User\", \"

```



```

+ "\"fields\":["
+ "{ \"name\": \"name\", \"type\": \"string\" }, "
+ "{ \"name\": \"age\", \"type\": \"int\" }"
+ "]"
+ "}";

return new Schema.Parser().parse(schemaStr);
}

// Extract schema ID from the message payload (for Schema Registry messages)
private static int getSchemaIdFromMessage(byte[] messagePayload) {
    return ((messagePayload[1] & 0xFF) << 24)
        | ((messagePayload[2] & 0xFF) << 16)
        | ((messagePayload[3] & 0xFF) << 8)
        | (messagePayload[4] & 0xFF);
}
}

```

Key Additions:

1. Error Topic:

- An error topic (`error-topic`) is defined to handle any deserialization errors.

2. Kafka Producer for Errors:

- Added a `KafkaProducer` to send error messages to the error topic.
- The producer properties (`producerProps`) are configured to connect to Kafka and serialize error messages as strings.

3. Exception Handling:

- Wrapped the deserialization process in a `try-catch` block.
- If an exception occurs, an error message is sent to the error topic with details about the failure.

Summary:

- The consumer reads from multiple topics and handles different message formats.
- Deserialization errors are captured and sent to a designated error topic.
- The `handleNonSchemaAvroMessage` method assumes a predefined schema for non-schema Avro messages or another schema extraction method.

Dependencies:

Ensure you have the necessary dependencies in your **Maven** or **Gradle** configuration, including Kafka client libraries and Avro.

Let me know if you need any additional features or adjustments!



Dead letter queue... am i correct?



Yes, you're right! A **Dead Letter Queue (DLQ)** is a type of error handling mechanism often used to manage messages that cannot be processed successfully.

Dead Letter Queue (DLQ) Concept

- **Purpose:** A DLQ is used to handle messages that fail processing after several retries or that have other issues that prevent them from being successfully processed.
- **Usage:** It allows you to isolate problematic messages and take corrective action without interrupting the processing of other messages.

Implementing a DLQ in Kafka

1. Setup DLQ Topic:

- Create a separate Kafka topic to serve as the DLQ. This topic will store messages that failed processing.

2. Consumer Error Handling:

- Modify your Kafka consumer to send messages that cannot be processed to the DLQ topic after a certain number of retries or upon encountering a specific error.

3. Configuration:

- You need to configure both the main topic and the DLQ topic in your Kafka setup.

Updated Kafka Consumer with DLQ

Here's how you can update the unified Kafka consumer class to include a DLQ mechanism:

java

```
import com.fasterxml.jackson.databind.JsonNode;
import com.fasterxml.jackson.databind.ObjectMapper;
import io.confluent.kafka.schemaregistry.client.CachedSchemaRegistryClient;
import io.confluent.kafka.schemaregistry.client.SchemaRegistryClient;
import org.apache.avro.Schema;
import org.apache.avro.generic.GenericDatumReader;
import org.apache.avro.generic.GenericRecord;
import org.apache.avro.io.DatumReader;
import org.apache.avro.io.Decoder;
import org.apache.avro.io.DecoderFactory;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.apache.kafka.common.serialization.StringSerializer;

import java.io.IOException;
import java.util.Collections;
import java.util.Properties;
import java.util.concurrent.Future;

public class UnifiedKafkaConsumer {

    private static final ObjectMapper objectMapper = new ObjectMapper();
    private static final String SCHEMA_REGISTRY_URL = "http://<external-schema-registry-url>:8081";
    private static final String ERROR_TOPIC = "error-topic";
    private static final String DLQ_TOPIC = "dlq-topic";

    public static void main(String[] args) throws IOException {
        Properties consumerProps = new Properties();
        consumerProps.put(ConsumerConfig.BootstrapServersConfig, "localhost:9092");
        consumerProps.put(ConsumerConfig.GroupIdConfig, "unified-group");
        consumerProps.put(ConsumerConfig.KeyDeserializerClassConfig,
StringDeserializer.class.getName());
        consumerProps.put(ConsumerConfig.ValueDeserializerClassConfig,
StringDeserializer.class.getName());
```

```

KafkaConsumer<String, Object> consumer = new KafkaConsumer<>(consumerProps);

// Subscribe to multiple topics
consumer.subscribe(Collections.singletonList("string-topic"));
consumer.subscribe(Collections.singletonList("json-topic"));
consumer.subscribe(Collections.singletonList("avro-schema-topic"));
consumer.subscribe(Collections.singletonList("avro-non-schema-topic"));

SchemaRegistryClient schemaRegistryClient = new
CachedSchemaRegistryClient(SCHEMA_REGISTRY_URL, 100);

// Initialize Kafka producer for DLQ messages
Properties producerProps = new Properties();
producerProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
producerProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
producerProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());

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

while (true) {
    for (ConsumerRecord<String, Object> record : consumer.poll(1000)) {
        String topic = record.topic();

        try {
            if (topic.equals("string-topic")) {
                handleStringMessage(record);
            } else if (topic.equals("json-topic")) {
                handleJsonMessage(record);
            } else if (topic.equals("avro-schema-topic")) {
                handleAvroWithSchemaRegistryMessage(record, schemaRegistryClient);
            } else if (topic.equals("avro-non-schema-topic")) {
                handleNonSchemaAvroMessage(record);
            }
        } catch (Exception e) {
            // Send error details to DLQ topic
            String errorMessage = "Error processing message from topic " + topic +
": " + e.getMessage();
            producer.send(new ProducerRecord<>(DLQ_TOPIC, errorMessage));
        }
    }
}

private static void handleStringMessage(ConsumerRecord<String, Object> record) {
    String message = (String) record.value();
    System.out.println("Received String message: " + message);
}

private static void handleJsonMessage(ConsumerRecord<String, Object> record) throws
IOException {
    String jsonString = (String) record.value();
    JsonNode jsonNode = objectMapper.readTree(jsonString);
    System.out.println("Received JSON message: " + jsonNode.toPrettyString());
}

private static void handleAvroWithSchemaRegistryMessage(ConsumerRecord<String, Object>
record, SchemaRegistryClient schemaRegistryClient) throws IOException {
    byte[] messagePayload = (byte[]) record.value();

    // Extract schema ID from the Avro message (stored in the payload)
    int schemaId = getSchemaIdFromMessage(messagePayload);
    Schema schema = schemaRegistryClient.getById(schemaId);

    DatumReader<GenericRecord> reader = new GenericDatumReader<>(schema);
    Decoder decoder = DecoderFactory.get().binaryDecoder(messagePayload, null);
    GenericRecord avroRecord = reader.read(null, decoder);

    System.out.println("Received Avro message (with Schema Registry): " + avroRecord);
}

// Handling Non-Schema Avro Message (Schema embedded in payload or pre-known schema)
private static void handleNonSchemaAvroMessage(ConsumerRecord<String, Object> record)
throws IOException {

```

```

byte[] messagePayload = (byte[]) record.value();

// If schema is embedded in the Avro message or you have a predefined schema
Schema schema = getPredefinedSchema(); // Use predefined or derived schema

DatumReader<GenericRecord> reader = new GenericDatumReader<>(schema);
Decoder decoder = DecoderFactory.get().binaryDecoder(messagePayload, null);
GenericRecord avroRecord = reader.read(null, decoder);

System.out.println("Received Avro message (Non-Schema): " + avroRecord);
}

// Mock method to provide a predefined Avro schema
private static Schema getPredefinedSchema() {
    // Define your schema here if not coming from Schema Registry
    String schemaStr = "{
        + \"type\": \"record\",
        + \"name\": \"User\",
        + \"fields\": [
        + {\"name\": \"name\", \"type\": \"string\"},
        + {\"name\": \"age\", \"type\": \"int\"}
        + ]
        + }";

    return new Schema.Parser().parse(schemaStr);
}

// Extract schema ID from the message payload (for Schema Registry messages)
private static int getSchemaIdFromMessage(byte[] messagePayload) {
    return ((messagePayload[1] & 0xFF) << 24)
        | ((messagePayload[2] & 0xFF) << 16)
        | ((messagePayload[3] & 0xFF) << 8)
        | (messagePayload[4] & 0xFF);
}
}

```

Key Additions:

1. DLQ Topic:

- Defined a `DLQ_TOPIC` constant to represent the topic where failed messages will be sent.

2. Kafka Producer for DLQ:

- Initialized a `KafkaProducer` (`producer`) to send messages to the DLQ topic.
- Configured the producer properties (`producerProps`) to connect to Kafka and serialize messages as strings.

3. Error Handling:

- In the `catch` block, any exceptions encountered during message processing are logged, and the error details are sent to the DLQ topic.

Summary:

- DLQ** is used to manage failed message processing.
- The consumer class now includes functionality to send messages to a DLQ if they encounter errors during processing.
- Adjust configurations and topics as per your specific Kafka setup and requirements.

This setup ensures that problematic messages are not lost and can be investigated separately. Let me know if you need further customization or have other questions!



Use ConcurrentKafkaListenerContainerFactory
 Create Bean Programmatically
 No use of annotation because I have to create resource based on dynamic configuration
 Error Handling
 And Improve efficiency
 Best Practice
 And Above context



Memory updated

Given your requirements to use `ConcurrentKafkaListenerContainerFactory` for programmatic bean creation and dynamic configuration, let's put together a solution that includes:

1. **Dynamic Kafka Consumer Bean Creation:** Programmatically create Kafka consumers and configure them based on dynamic properties.
2. **Error Handling:** Implement robust error handling and potentially use a Dead Letter Queue (DLQ).
3. **Efficiency:** Ensure the setup is efficient and follows best practices.

Kafka Consumer Configuration with `ConcurrentKafkaListenerContainerFactory`

Here's how you can set up a `ConcurrentKafkaListenerContainerFactory` programmatically, including error handling and best practices.

Step 1: Define Kafka Configuration

Create a configuration class that sets up the Kafka consumer factory and listener container factory programmatically.

```
java
```

```
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
import org.apache.kafka.common.serialization.StringSerializer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.config.KafkaListenerContainerFactory;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.core.DefaultKafkaConsumerFactory;
import org.springframework.kafka.core.DefaultKafkaProducerFactory;
import org.springframework.kafka.core.ProducerFactory;
import org.springframework.kafka.listener.ConcurrentMessageListenerContainer;
import org.springframework.kafka.listener.MessageListener;
import org.springframework.kafka.listener.config.ContainerProperties;
import org.springframework.kafka.support.KafkaHeaders;
import org.springframework.kafka.support.serializer.ErrorHandlingDeserializer;
import org.springframework.kafka.support.serializer.JsonDeserializer;
import org.springframework.kafka.support.serializer.StringDeserializer;
import org.springframework.kafka.support.serializer.StringSerializer;
import org.apache.kafka.common.serialization.Deserializer;
```

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

@Configuration
public class KafkaConfig {

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

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

    @Bean
    public Map<String, Object> consumerProps() {
        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "dynamic-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
ErrorHandlingDeserializer.class);
        props.put(ErrorHandlingDeserializer.VALUE_DESERIALIZER_CLASS,
JsonDeserializer.class);
        props.put(JsonDeserializer.VALUE_DEFAULT_TYPE, "java.lang.String"); // Adjust type
accordingly
        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        return props;
    }

    @Bean
    public KafkaProducer<String, String> errorProducer() {
        Properties producerProps = new Properties();
        producerProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        producerProps.put(ProducerConfig.KEY_SERIAL
```



Use ConcurrentKafkaListenerContainerFactory
 Create Bean Programmatically
 No use of annotation because I have to created resource based on dynamic configuration
 Error Handling
 And Improve efficiency
 Best Practice
 And Above context
 and Deserilaization and Serialization



Certainly! Given your requirements, I'll guide you through creating a Kafka consumer setup programmatically using `ConcurrentKafkaListenerContainerFactory`, including dynamic configuration, error handling, and best practices for deserialization and serialization.

Overview

1. **Dynamic Kafka Consumer Configuration:** Programmatically create Kafka consumers based on dynamic properties.
2. **Error Handling:** Implement a robust error handling mechanism.
3. **Efficiency:** Ensure that the setup is efficient.
4. **Serialization/Deserialization:** Handle various message formats including Avro and JSON.

Step 1: Define Kafka Configuration Programmatically

1.1 Kafka Consumer Configuration

Create a configuration class to set up Kafka properties, `ConcurrentKafkaListenerContainerFactory`, and error handling.

```
java
```

```
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.apache.kafka.clients.consumer.ConsumerFactory;
import org.apache.kafka.clients.consumer.DefaultKafkaConsumerFactory;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerFactory;
import org.apache.kafka.clients.producer.DefaultKafkaProducerFactory;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.StringSerializer;
import org.springframework.context.annotation.Bean;
import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.config.KafkaListenerContainerFactory;
import org.springframework.kafka.core.ConsumerFactory;
import org.springframework.kafka.core.DefaultKafkaConsumerFactory;
import org.springframework.kafka.listener.ConcurrentMessageListenerContainer;
import org.springframework.kafka.listener.MessageListener;
import org.springframework.kafka.listener.config.ContainerProperties;
import org.springframework.kafka.listener.error.SeekToCurrentErrorHandler;
import org.springframework.kafka.support.serializer.JsonDeserializer;
import org.springframework.kafka.support.serializer.ErrorHandlingDeserializer;
import org.apache.avro.Schema;
import org.apache.avro.generic.GenericRecord;
import org.apache.avro.io.DatumReader;
import org.apache.avro.io.Decoder;
import org.apache.avro.io.DecoderFactory;
import org.apache.avro.generic.GenericDatumReader;

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

@Configuration
public class KafkaConfig {

    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, String>
kafkaListenerContainerFactory() {
        ConcurrentKafkaListenerContainerFactory<String, String> factory = new
ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory());
        factory.setErrorHandler(new SeekToCurrentErrorHandler()); // Configurable error
handler
        return factory;
    }

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

    @Bean
    public Map<String, Object> consumerProps() {
```



```

        Map<String, Object> props = new HashMap<>();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "dynamic-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
ErrorHandlingDeserializer.class);
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
ErrorHandlingDeserializer.class);
        props.put(ErrorHandlingDeserializer.VALUE_DESERIALIZER_CLASS,
CustomDeserializer.class.getName());
        return props;
    }

    @Bean
    public ProducerFactory<String, String> producerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        return new DefaultKafkaProducerFactory<>(props);
    }

    @Bean
    public KafkaProducer<String, String> kafkaProducer() {
        return new KafkaProducer<>(producerFactory().getConfigurationProperties());
    }
}

```

1.2 Custom Deserializer

Implement a custom deserializer to handle various formats like Avro and JSON dynamically.

java

```

import org.apache.kafka.common.serialization.Deserializer;
import org.apache.avro.generic.GenericRecord;
import org.apache.avro.io.DatumReader;
import org.apache.avro.io.Decoder;
import org.apache.avro.io.DecoderFactory;
import org.apache.avro.generic.GenericDatumReader;

import java.util.Map;

public class CustomDeserializer implements Deserializer<Object> {

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

    @Override
    public Object deserialize(String topic, byte[] data) {
        try {
            if (data == null || data.length == 0) {
                return null;
            }

            // Example: Check for schema ID or other indicators
            // Adjust this logic based on your specific data format
            if (isAvroFormat(data)) {
                return deserializeAvro(data);
            } else if (isJsonFormat(data)) {
                return new String(data);
            } else {
                throw new IllegalArgumentException("Unsupported data format");
            }
        } catch (Exception e) {
            // Handle deserialization errors
            throw new RuntimeException("Deserialization error", e);
        }
    }
}

```

```

}

private boolean isAvroFormat(byte[] data) {
    // Implement logic to determine if data is in Avro format
    return true; // Placeholder
}

private boolean isJsonFormat(byte[] data) {
    // Implement logic to determine if data is in JSON format
    return false; // Placeholder
}

private Object deserializeAvro(byte[] data) throws IOException {
    // Use Avro schema to deserialize
    Schema schema = getAvroSchema(); // Obtain schema dynamically if necessary
    DatumReader<GenericRecord> reader = new GenericDatumReader<>(schema);
    Decoder decoder = DecoderFactory.get().binaryDecoder(data, null);
    return reader.read(null, decoder);
}

private Schema getAvroSchema() {
    // Return a predefined Avro schema or obtain dynamically
    String schemaStr = "{"
        + "\"type\": \"record\", \"
        + "\"name\": \"User\", \"
        + "\"fields\": [\"
        + \"{ \"name\": \"name\", \"type\": \"string\" }, \"
        + \"{ \"name\": \"age\", \"type\": \"int\" } \"
        + \"] \"
        + \"; \"
    return new Schema.Parser().parse(schemaStr);
}

@Override
public void close() {
    // Cleanup if needed
}
}

```

Step 2: Setting Up Error Handling

SeekToCurrentErrorHandler: This is a built-in error handler that seeks to the current offset on errors. You can configure it to move failed messages to a DLQ if needed.

Step 3: Creating Consumer Beans Dynamically

To dynamically create Kafka consumers based on configuration:

java

```

import org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory;
import org.springframework.kafka.listener.ConcurrentMessageListenerContainer;
import org.springframework.kafka.core.ConsumerFactory;

public class DynamicKafkaConsumer {

    private final ConcurrentKafkaListenerContainerFactory<String, String> factory;
    private final KafkaProducer<String, String> kafkaProducer;

    public DynamicKafkaConsumer(ConcurrentKafkaListenerContainerFactory<String, String>
factory, KafkaProducer<String, String> kafkaProducer) {
        this.factory = factory;
        this.kafkaProducer = kafkaProducer;
    }

    public void createConsumer(String topic) {
        ConcurrentMessageListenerContainer<String, String> container =

```



```
factory.createContainer(topic);
    container.setMessageListener(new MessageListener<String, String>() {
        @Override
        public void onMessage(ConsumerRecord<String, String> record) {
            try {
                // Process the message
                System.out.println("Processing message: " + record.value());
            } catch (Exception e) {
                // Send failed messages to DLQ
                kafkaProducer.send(new ProducerRecord<>("dlq-topic", record.value()));
            }
        }
    });
    container.start();
}
```

Summary

- **Dynamic Configuration:** The Kafka consumer is set up programmatically to adapt to dynamic configurations.
- **Error Handling:** Uses `SeekToCurrentErrorHandler` and can send failed messages to a DLQ.
- **Deserialization:** Custom deserializer handles multiple formats including Avro and JSON.
- **Efficiency:** The setup is designed to be efficient and robust for real-world applications.

This approach ensures that you can create Kafka consumers dynamically based on configuration, handle errors gracefully, and manage different message formats effectively. Let me know if you need further customization or additional details!

ChatGPT can make mistakes. Check important info.