public class MyMessageSchema implements DeserializationSchema<MyMessage>, SerializationSchema<MyMessage> {

@Override

public MyMessage deserialize(byte[] bytes) throws IOException {

return MyMessage.fromString(new String(bytes));

}

@Override

public byte[] serialize(MyMessage myMessage) {

return myMessage.toString().getBytes();

}

@Override

public TypeInformation<MyMessage> getProducedType() {

return TypeExtractor.getForClass(MyMessage.class);

}

// Method to decide whether the element signals the end of the stream.

// If true is returned the element won't be emitted.

@Override

public boolean isEndOfStream(MyMessage myMessage) {

return false;

}

}

SimpleStringSchema: SimpleStringSchema deserializes the message as a string. In case your messages have keys, the latter will be ignored.

new FlinkKafkaConsumer09<>(kafkaInputTopic, new SimpleStringSchema(), prop);

JSONDeserializationSchema

JSONDeserializationSchema deserializes json-formatted messages using jackson and returns a stream of com.fasterxml.jackson.databind.node.ObjectNode objects. You can then use the .get("property") method to access fields. Once again, keys are ignored.

new FlinkKafkaConsumer09<>(kafkaInputTopic, new JSONDeserializationSchema(), prop);

JSONKeyValueDeserializationSchema

JSONKeyValueDeserializationSchema is very similar to the previous one, but deals with messages with json-encoded keys AND values.

boolean fetchMetadata = true;

new FlinkKafkaConsumer09<>(kafkaInputTopic, new JSONKeyValueDeserializationSchema(fetchMetadata), properties);

The ObjectNode returned contains the following fields:

key: all the fields present in the key

value: all the message fields

(optional) metadata: exposes the offset, partition and topic of the message (pass true to the constructor in order to fetch metadata as well).

For example:

kafka-console-producer --broker-list localhost:9092 --topic json-topic \

--property parse.key=true \

--property key.separator=|

{"keyField1": 1, "keyField2": 2} | {"valueField1": 1, "valueField2" : {"foo": "bar"}}

^C

Will be decoded as:

{

"key":{"keyField1":1,"keyField2":2},

"value":{"valueField1":1,"valueField2":{"foo":"bar"}},

"metadata":{

"offset":43,

"topic":"json-topic",

"partition":0

}

}

In kafka, each consumer from the same consumer group gets assigned one or more partitions. Note that it is not possible for two consumers to consume from the same partition. The number of flink consumers depends on the flink parallelism (defaults to 1).

There are three possible cases:

kafka partitions == flink parallelism: this case is ideal, since each consumer takes care of one partition. If your messages are balanced between partitions, the work will be evenly spread across flink operators;

kafka partitions < flink parallelism: some flink instances won't receive any messages. To avoid that, you need to call rebalance on your input stream before any operation, which causes data to be re-partitioned:

inputStream = env.addSource(new FlinkKafkaConsumer10("topic", new SimpleStringSchema(), properties));

inputStream

.rebalance()

.map(s -> "message" + s)

.print();

kafka partitions > flink parallelism: in this case, some instances will handle multiple partitions. Once again, you can use rebalance to spread messages evenly accross workers.