Kafka Message Aggregation using Camel and Spring Boot

# Introduction

[Apache Camel](https://camel.apache.org/) is a popular open source integration framework that can work with almost any message brokers like Kafka, ActiveMQ, RabbitMQ etc. It provides [out of the box support](https://camel.apache.org/manual/latest/enterprise-integration-patterns.html) for the most popular EIPs ([Enterprise Integration Patterns](https://www.enterpriseintegrationpatterns.com/patterns/messaging/toc.html)). Camel can also work seamlessly with Spring Boot, and that makes it a killer combination. In this example, we will see how to use the [Aggregate EIP](https://camel.apache.org/manual/latest/aggregate-eip.html) provided by Camel to do message aggregation on Kafka.

# Problem Statement

We are building a microservice. It reads BankDetail messages in JSON format from the Kafka Topic *bank-details*. A BankDetail message has the below attributes:

private int id;

private int age;

private String job;

private String marital;

private String education;

private String defaulted;

private BigDecimal balance;

private String housing;

private String loan;

private String contact;

private int day;

private String month;

private int duration;

private int campaign;

private int pdays;

private int previous;

private String poutcome;

private String y;

BankDetail messages come in batches, each message of the same batch, has the same *kafka.key*. The microservice then aggregates all the messages of the same batch based on the *job* and finds out the count of various job categories. It would then publish the result of the aggregation on the Kafka Topic *bank-details-aggregated*. A typical aggregate message would look like:

{

"adminCount": 478,

"blueCollarCount": 946,

"entrepreneurCount": 168,

"houseMaidCount": 112,

"managementCount": 969,

"retiredCount": 230,

"selfEmployedCount": 183,

"servicesCount": 417,

"studentCount": 84,

"technicianCount": 768,

"unemployedCount": 128,

"unknownCount": 38

}

# Solution

## Project Setup using Maven

Its a standard Spring Boot project. We will define the Camel BOM as below:

<dependencyManagement>

<dependencies>

<!-- Camel BOM -->

<dependency>

<groupId>org.apache.camel</groupId>

<artifactId>camel-spring-boot-dependencies</artifactId>

<version>${spring.camel-version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

Then we will define the dependencies for Camel:

<!-- START :: Camel -->

<dependency>

<groupId>org.apache.camel</groupId>

<artifactId>camel-spring-boot-starter</artifactId>

</dependency>

<dependency>

<groupId>org.apache.camel</groupId>

<artifactId>camel-stream-starter</artifactId>

</dependency>

<dependency>

<groupId>org.apache.camel</groupId>

<artifactId>camel-kafka</artifactId>

</dependency>

<dependency>

<groupId>org.apache.camel</groupId>

<artifactId>camel-kafka-starter</artifactId>

</dependency>

<dependency>

<groupId>org.apache.camel</groupId>

<artifactId>camel-jackson-starter</artifactId>

</dependency>

<!-- END :: Camel -->

## Contract with the message publisher

We have the below contract with the BankDetail message publisher:

1. The messages will be published in JSON format on the Kafka Topic *bank-details*.
2. The *kafka.key* of all messages of the same group or batch would be identical.
3. The message will have a header named *\_\_TypeId\_\_* that will have its fully qualified Java class name. This feature comes out of the box with Spring.
4. There is no expectation as to the type of the key, as long as the keys are fairly unique across different batches. In this example, I am using a random UUID.
5. After all messages in a batch are published, a CompletionSignal is published on Kafka.

A simple BankDetail publisher that respects the above contract can be found [here](https://github.com/paawak/spring-boot-demo/tree/master/kafka-spring/kafka-simple-publisher).

## Implementation details

### Camel Route

* + 1. A *Route* defines a logical message routing. We define a *Route* by extending a *RouteBuilder*. And it can be defined as a Spring Bean:

@Service

public class BankDetailAggregatorByJob extends RouteBuilder {

@Override

public void configure() {

...

}

}

* + 1. Inside the *configure()* method, we define how we will process the incoming messages. In this case, we will read the messages off a Kafka Topic, aggregate those based on some condition and then publish the aggregated message back onto a Kafka Topic.

### Reading messages from Kafka

The below line reads messages off the Kafka Topic *bank-details*:

from("kafka:**bank-details**?brokers=" + kafkaBrokers + "&autoOffsetReset=earliest"

+ "&autoCommitEnable=true" + "&groupId=bank-detail-camel-consumer")

### Define a Route ID

* + 1. Its always a good practice to define a RouteID, its helps debugging.
    2. .routeId(BankDetailAggregatorByJob.class.getSimpleName())

### Logging

* + 1. Camel supports logging out of the box and also takes in a Slf4J Logger. Here *LOG* is a Slf4j *Logger*. *${header[key]}* will print the message header with the *key*. *${headers}* will print all the message headers and *${body}* will print the message body. These are very useful short hands for debugging. [Here](https://camel.apache.org/manual/latest/simple-language.html) is the full list of expressions supported by Camel.
    2. .log(LoggingLevel.TRACE, LOG,
    3. "${header[" + RouteConstants.TYPE\_HEADER + "]}")

### Conditional branching using choice expression

* + 1. In our Route, we can expect the below 2 types of messages:
    2. 1. *BankDetail*: This is the message that has to be aggregated
    3. 2. *CompletionSignal*: This message signifies the end of the batch, essentially, the end of aggregation operation
    4. Based on the type of message, as defined by the *\_\_TypeId\_\_* header, we would have to do different things. If the message is of the type *BankDetail*, we would convert this to a *JobCount*, which is the aggregate message. If the message is of the type *CompletionSignal*, we would send a signal to the aggregation framework to complete the aggregation operation for the current batch. We would do that by setting a *Boolean* flag. The complete choice code is shown below:
    5. .choice()
    6. .when(simple("${header." + RouteConstants.TYPE\_HEADER + "} == '"
    7. + BankDetail.class.getName() + "'"))
    8. .unmarshal().json(JsonLibrary.Jackson, BankDetail.class)
    9. .process(exchange -> {
    10. BankDetail bankDetail = exchange.getIn().getBody(BankDetail.class);
    11. exchange.getIn().setBody(toJobCount(bankDetail), JobCount.class);
    12. }).when(simple("${header." + RouteConstants.TYPE\_HEADER + "} == '"
    13. + CompletionSignal.class.getName() + "'"))
    14. .process(exchange -> {
    15. exchange.getIn().setBody(new JobCount(), JobCount.class);
    16. exchange.getIn().setHeader(
    17. RouteConstants.COMPLETE\_JOB\_AGGREGATION\_COMMAND, Boolean.TRUE);
    18. }).end()

Note that the message arrives as a JSON formatted string. In order to convert that to a Java object, we need to apply the *unmarshal()* transform as shown below:

.unmarshal().json(JsonLibrary.Jackson, BankDetail.class)

### Aggregation Implementation

* + 1. In its simplest form, the aggregate() function takes in 2 params:
    2. 1. The *expression*, based on which the messages are grouped together for aggregation. In the present case, messages are aggregated by the *KafkaKey* header.
    3. 2. The *AggregationStrategy*: Defines how 2 messages can be merged into a single message: more on it later.
    4. .aggregate(header(KafkaConstants.KEY), new BankDetailAggregationStrategy())
    5. We can also specify the *completionTimeout*. This defines the max time this aggregation operation can wait for the next message to arrive, starting from the time the last message was received. In essence, if a message does not arrive within the stipulated *completionTimeout* of the last message in the same aggregation group, the aggregation operation would be deemed to have timed out. We can also specify if we want to discard the partially aggregated message in case of time out.
    6. .completionTimeout(2\_000).discardOnCompletionTimeout()

### Aggregation Strategy

* + 1. The *AggregationStrategy* is the place where incoming messages are combined to form a single message. It defines a single method:
    2. public Exchange aggregate(Exchange oldExchange, Exchange newExchange)
    3. The *oldExchange* holds the partial aggregate results, while the *newExchange* holds the new incoming message. We would need to compute the next partial result and return it. Note that for the first message, the *oldExchange* would be *null*.
    4. @Override
    5. public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
    6. if (oldExchange == null) {
    7. LOG.info("########### First message recieved for the correlationID: {}",
    8. getCorrelationId(newExchange));
    9. return newExchange;
    10. }
    11. JobCount partialResults = oldExchange.getIn().getBody(JobCount.class);
    12. JobCount newMessage = newExchange.getIn().getBody(JobCount.class);
    13. oldExchange.getIn().setBody(doAggregation(newMessage, partialResults), JobCount.class);
    14. // any header that is used by the predicate should be set in the message
    15. // returned from this method
    16. Map<String, Object> headers = new HashMap<>();
    17. headers.putAll(oldExchange.getIn().getHeaders());
    18. headers.putAll(newExchange.getIn().getHeaders());
    19. oldExchange.getIn().setHeaders(headers);
    20. return oldExchange;
    21. }

1. Apart from computing the aggregate result, it can also do some other interesting things as defined below.

### Completing the Aggregation

* + 1. If we implement the *Predicate*, we could indicate when the aggregation operation is deemed complete. In the present case, we wait till we receive a *TRUE* signal in a given header.
    2. @Override
    3. public boolean matches(Exchange oldExchange) {
    4. Boolean completionCommand = oldExchange.getIn()
    5. .getHeader(RouteConstants.COMPLETE\_JOB\_AGGREGATION\_COMMAND, Boolean.class);
    6. if (completionCommand == null) {
    7. return false;
    8. }
    9. LOG.debug("\*\*\*\*\*\*shouldComplete: {}", completionCommand);
    10. return completionCommand;
    11. }

### Aggregation Complete signal

* + 1. If we implement the *CompletionAwareAggregationStrategy*, we would get a callback when the aggregation operation completes successfully.
    2. @Override
    3. public void onCompletion(Exchange exchange) {
    4. String correlationId = getCorrelationId(exchange);
    5. LOG.info("########### Aggregation is complete for {}", correlationId);
    6. }

### Aggregation Timeout signal

* + 1. If we implement the *TimeoutAwareAggregationStrategy*, we would get a callback if the aggregation operation times out.
    2. @Override
    3. public void timeout(Exchange oldExchange, int index, int total, long timeout) {
    4. LOG.info("############# timed out");
    5. }

### Converting message to JSON

* + 1. Before we publish the aggregate message to the bus, we would need to convert the Java POJO into JSON.
    2. .marshal().json(JsonLibrary.Jackson)

### Publishing message to Kafka

* + 1. This is how we would publish the aggregated message to the *bank-details-aggregated* Kafka Topic:
    2. .to("kafka:bank-details-aggregated?brokers=" + kafkaBrokers)

## Putting it together

* 1. Assuming that the [kafka-simple-publisher](https://github.com/paawak/spring-boot-demo/tree/master/kafka-spring/kafka-simple-publisher) is running on *localhost:8090*, to trigger BankDetail messages on Kafka Topic *bank-detail*, do:
  2. curl "<http://localhost:8090/kafka/publish>"
  3. Wait for around 50 seconds, and then you would see the aggregated message published on the *bank-details-aggregated* topic.

## Source Code

* 1. The completely working source code can be found here:
  2. <https://github.com/paawak/spring-boot-demo/tree/master/kafka-spring/kafka-camel-aggregation-simple>