

Posts

Sending and consuming messages with Spring and Kafka

=====

🕒 5 minute read ✎ Published: 13 Apr, 2018


> This demonstration explains how to craft classical (not reactive) consumer/producer components within you
> Spring apps

► Table of Contents


Spring Kafka Writer and Readers

=====

What is Kafka

Apache Kafka is an open-source stream-processing software platform developed by the Apache Software Foundation written in Scala and Java. You'll find more information about kafka at [it's Homepage](#) .

Intro to Spring For Apache Kafka

The Spring for Apache Kafka (spring-kafka) project applies core Spring concepts to the development of kafka-based messaging solutions. It provides a “template” as a high-level abstraction for sending messages. It also provides support for Message-driven POJOs with [@KafkaListener](#)  annotations and a ‘listener container’. You will see similarities to the JMS support in the Spring Framework and RabbitMQ support in Spring AMQP.

See [HomePage](#)  for more details on this project.




Kafka Producers

For connecting to kafka brokers, you will need to specify a host:port property value for ``spring.kafka.producer.bootstrap-servers``. This tells Spring to configure any of the Producer/Consumer Factories with that host as it's target. By default, Spring will autoconfigure this to connect to ‘localhost:9092’.

A producer factory creates ``org.apache.kafka.clients.producer.Producer``'s. There is a lot of code involved when using the low level Producer API. Prefer instead to use the ``org.springframework.kafka.core.KafkaTemplate``.


Configuring a Kafka Template

Spring provides the [org.springframework.boot.autoconfigure.kafka.KafkaAutoConfiguration](#)  bean that gets consumed on app start. This bean will do all of the component creation and setup once we have configuration properties ready.


Lets take a look at a fairly simple config for setting up a working producer called a [KafkaTemplate](#) . The class [KafkaProperties](#)  is where we can find any of the config items we'll need to make our producer work. This bean relays configuration specifics such as `acks`, and `retries` to the [KafkaProducer](#)  that is used within `KafkaTemplate`.

application.properties:

```
spring.kafka.producer.client-id=producer-1
spring.kafka.producer.retries=3
spring.kafka.producer.acks=1
spring.kafka.producer.value-serializer=org.springframework.kafka.support.serializer.JsonSerializer
```

This is a producer configuration that Kafka brokers will see as “producer-1”. It will attempt sending messages up to 3 times before erring, and will wait for leader acknowledgement (ack) before considering messages as sent. By default, Spring chooses a [StringSerializer](#) , but since we want to send our custom domain objects, we will opt for Json.

Sending Messages to Kafka

KafkaTemplate will be used to get messages into Kafka topics. we should use the [send\(\)](#)  method which gets overloaded for topic, partition, key and then data(message body). Our situation requires us to get data into a topic, then later read it out. We'll send data into our topic given a topic named 'foobar', and a message containing the contents of a 'Customer' POJO.

KafkaProducerApplication:

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

```

    }

    @Bean
    ApplicationRunner applicationRunner(KafkaTemplate<String, Customer> kafkaTemplate) {
        return args -> {
            while (true) {
                Thread.sleep(1000);
                kafkaTemplate.send("foobar", new Customer(System.currentTimeMillis(), "mario"));
            }
        };
    }
}

```

What this application does, is connect to Kafka, then sends a Customer POJO every second. The KafkaTemplate bean is automatically populated because we included `org.springframework.kafka:spring-kafka` dependency in our pom. Autoscanning found our application properties we declared earlier, and made sure the right serializers were active. So we neatly complete this scenario by injecting the KafkaTemplate instance into our application class and start pushing messages into our topic.

Executing this application should show standard output of the spring-bot application, and nothing else. Next step here is to consume the messages we just sent in:

Log Output:


```

2018-03-25 22:24:30.845 INFO 25704 --- [           main] o.a.kafka.common.utils.AppInfoParser : Kafka version
2018-03-25 22:24:30.845 INFO 25704 --- [           main] o.a.kafka.common.utils.AppInfoParser : Kafka commitI

```


A Kafka Consumer

Kafka library supports the [KafkaConsumer](#) class to bind client logic to Kafka topic events - messages received. The event process can be programmed imperatively but is complex in that your message handling logic will have to find out how to deserialize, and ultimately route your messages to the right method. Spring Kafka provides the implementation for [MessageListenerContainer](#) to perform this message forwarding, manage concurrency, and re-balance topic-partitions consumed by individual `KafkaConsumer`s.

Getting your messages out of Kafka is done by annotating a method with [KafkaListener](#) . This method accept both the `java.util.Collection<T>` and bare type `<T>` of the types you expect from the topic. For example, lets visit how we wire this listening method together:


MessageProcessor.java:

```
@KafkaListener(topics = {"foobar"})
public void processMessage(Collection<Customer> customer) {
    customer.forEach(r -> {
        System.out.println("gid: " + myGroupId + ", record = " + r);
    });
}
```

What the annotation does is hoist this `processMessage` method into the [KafkaListenerContainerRegistry](#) . It gets invoked whenever a message arrives on topic 'foobar' that satisfies the method arguments - in this case, a collection of Customers. Finally the body of our method prints out each customer.

application.properties:

```
spring.kafka.consumer.value-deserializer=org.springframework.kafka.support.serializer.JsonDeserializer
spring.kafka.consumer.properties.spring.json.trusted.packages=*
spring.kafka.consumer.group-id=${my.group.id:default}
```

As usual, you must specify the value-deserializer. Specifying [JsonDeserializer](#)  allows our application to read JSON encoded message bodies, and return our domain objects as payload for processing. Setting `trusted.packages` to the lowest level of the package your POJO's (e.g. `com.example.pojos`) are located in. Alternately set this property to `*` so that any POJO can become eligible for deserialization. Each consumer should have a unique ID if you're running in offset mode. To avoid offset commit conflicts, you should usually ensure that the `groupId` is unique for each application context.

Running the consumer, we should expect to see the messages we put into the "foobar" topic earlier:

```
2018-03-26 00:47:54.464 INFO 30872 --- [ntainer#0-0-C-1] o.s.k.l.KafkaMessageListenerContainer : partitions as
GROUPID default, record = Customer(id=1522050390932, name=mario)
GROUPID default, record = Customer(id=1522050391934, name=mario)
```

Wrap-up

Indeed, we have scratched the surface just a little. Your next logical step to discover more about spring-kafka is to visit my recommended reading list:

Published by Mario Gray 13 Apr, 2018 in [kafka](#), [messaging](#), [mq](#) and [spring](#) and tagged [demo](#), [functional](#), [java](#), [spring](#) and [web](#) using 900 words.

Related Content

- [Intro to RIFF Is For Functions](#) - 5 minutes
- [Spring Test Slices](#) - 5 minutes

This page was generated using [After Dark](#)  for [Hugo](#) .