(/)

Intro to Apache Kafka with Spring

Last modified: August 29, 2020

by bael

X

Persister

Spring (https://www.baeldung.com/category/spring/) +

Kafka (https://www.baeldung.com/tag/kafka/) **Messaging (https://www.baeldung.com/tag/messaging/)**

I just announced the new *Learn Spring* course, focused on the fundamentals of Spring 5 and Spring Boot 2:

>> CHECK OUT THE COURSE (/ls-course-start)

1. Overview

Apache Kafka (https://kafka.apache.org/) is a distributed and fault-tolerant stream processing system.

In this article, we'll cover Spring support for Kafka and the level of abstractions it provides over native Kafka Java client APIs.

Spring Kafka brings the simple and typical Spring template programming model with a *KafkaTemplate* and Message-driven POJOs via *@KafkaListener* annotation.

Further reading:

Building a Data Pipeline with Flink and Kafka (/kafka-flink-data-pipeline)

Learn how to process stream data with Flink and Kafka

Read more (/kafka-flink-data-pipeline) →

Kafka Connect Example with MQTT and MongoDB (/kafka-connect-mqtt-mongodb)

Have a look at a practical example using Kafka connectors.

Read more (/kafka-connect-mqtt-mongodb) →

2. Installation and Setup

To downl ckstart).

We also n

```
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
    <version>2.3.7.RELEASE</version>
</dependency>
```

The latest version of this artifact can be found here (https://search.maven.org/classic/#search%7Cga%7C1%7Cg%3A%22org.springframework.kafka%22%20AND%20a%3A%22spring-kafka%22).

Our example application will be a Spring Boot application.

This article assumes that the server is started using the default configuration and no server ports are changed.

3. Configuring Topics

Previously we used to run command line tools to create topics in Kafka such as:

```
$ bin/kafka-topics.sh --create \
  --zookeeper localhost:2181 \
  --replication-factor 1 --partitions 1 \
  --topic mytopic
```

But with the introduction of *AdminClient* in Kafka, we can now create topics programmatically.

We need to add the *KafkaAdmin* Spring bean, which will automatically add topics for all beans of type *NewTopic:*

```
@Configuration
public class KafkaTopicConfig {

    @Value(value = "${kafka.bootstrapAddress}")
    private String bootstrapAddress;

    @Bean
    public KafkaAdmin kafkaAdmin() {
        Map<String, Object> configs = new HashMap<>();
        configs.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapAddress);
        return new KafkaAdmin(configs);
    }

    @Bean
    public NewTopic topic1() {
         return new NewTopic("baeldung", 1, (short) 1);
    }
}
```

4. Pro

To create messages, first, we need to configure a *ProducerFactory* (https://docs.spring.io/spring-kafka/api/org/springframework/kafka/core/ProducerFactory.html) which sets the strategy for creating Kafka *Producer* (https://kafka.apache.org/0100/javadoc/org/apache/kafka/clients/producer/Producer.html) instances.

Then we need a *KafkaTemplate* (https://docs.spring.io/spring-kafka/api/org/springframework/kafka/core/KafkaTemplate.html) which wraps a *Producer* instance and provides convenience methods for sending messages to Kafka topics.

Producer instances are thread-safe and hence using a single instance throughout an application context will give higher performance. Consequently, *KakfaTemplate* instances are also thread-safe and use of one instance is recommended.

4.1. Producer Configuration

```
@Configuration
public class KafkaProducerConfig {

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

    @Bean
    publ

    }
}
```

4.2. Publishing Messages

We can send messages using the *KafkaTemplate* class:

```
@Autowired
private KafkaTemplate<String, String> kafkaTemplate;

public void sendMessage(String msg) {
    kafkaTemplate.send(topicName, msg);
}
```

The *send* API returns a *ListenableFuture* object. If we want to block the sending thread and get the result about the sent message, we can call the *get* API of the *ListenableFuture* object. The thread will wait for the result, but it will slow down the producer.

Kafka is a fast stream processing platform. So it's a better idea to handle the results asynchronously so that the subsequent messages do not wait for the result of the previous message. We can do this through a callback:

```
public void sendMessage(String message) {

    ListenableFuture<SendResult<String, String>> future =
      kafkaTemplate.send(topicName, message);

    future.addCallback(new ListenableFutureCallback<SendResult<String, String>>() {

        @Override
        public void onSuccess(SendResult<String, String> result) {
            System.out.println("Sent message=[" + message +
              "] with offset=[" + result.getRecordMetadata().offset() + "]");
        }
        @Override
        public void onFailure(Throwable ex) {
            System.out.println("Unable to send message=["
              + message + "] due to : " + ex.getMessage());
        }
    });
}
```

5. Consuming Messages

5.1. Consumer Configuration

X

For consuming messages, we need to configure a *ConsumerFactory (https://docs.spring.io/autorepo/docs/spring-kafka-dist/1.1.3.RELEASE/api/org/springframework/kafka/core/ConsumerFactory.html)* and a *KafkaListenerContainerFactory* (https://docs.spring.io/autorepo/docs/spring-kafka-dist/1.1.3.RELEASE/api/org/springframework/kafka/config/KafkaListenerContainerFactory.html). Once these beans are available in the Spring bean factory, POJO based consumers can be configured using *@KafkaListener* (https://docs.spring.io/autorepo/docs/spring-kafka-dist/1.1.3.RELEASE/api/org/springframework/kafka/annotation/KafkaListener.html) annotation.

***@EnableKafka* (https://docs.spring.io/autorepo/docs/spring-kafka-dist/1.1.3.RELEASE/api/org/springframework/kafka/annotation/EnableKafka.html) annotation is required on the configuration class to enable detection of *@KafkaListener* annotation on spring managed beans:**

```
@EnableKafka
@Configuration
public class KafkaConsumerConfig {

    @Bean
    public ConsumerFactory<String, String> consumerFactory() {
        Map<String, Object> props = new HashMap<>();
        props.put(
          ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
          bootstrapAddress);
        props.put(
          ConsumerConfig.GROUP_ID_CONFIG,
          groupId);
        props.put(
          ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
          StringDeserializer.class);
        props.put(
          ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
          StringDeserializer.class);
        return new DefaultKafkaConsumerFactory<>(props);
    }

    @Bean
    public ConcurrentKafkaListenerContainerFactory<String, String>
      kafkaListenerContainerFactory() {

        ConcurrentKafkaListenerContainerFactory<String, String> factory =
          new ConcurrentKafkaListenerContainerFactory<>();
        factory.setConsumerFactory(consumerFactory());
        return factory;
    }
}
```

5.2. Consuming Messages

```
@KafkaListener(topics = "topicName", groupId = "foo")
public void listenGroupFoo(String message) {
    System.out.println("Received Message in group foo: " + message);
}
```

Multiple listeners can be implemented for a topic, each with a different group Id. Furthermore, one consumer can listen for messages from various topics:

```
@KafkaListener(topics = "topic1, topic2", groupId = "foo")
```

Spring also supports retrieval of one or more message headers using the *@Header* (https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/messaging/handler/annotation/Header.html) annotation in the listener:

```
@KafkaListener(topics = "topicName")
public void listenWithHeaders(
  @Payload String message,
  @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition) {
      System.out.println(
        "Received Message: " + message"
        + "from partition: " + partition);
}
```

5.3. Consuming Messages from a Specific Partition

As you may have noticed, we had created the topic *baeldung* with only one partition. However, for a topic with multiple partitions, a *@KafkaListener* can explicitly subscribe to a particular partition of a topic with an initial offset:

```
@KafkaListener(
  topicPartitions = @TopicPartition(topic = "topicName",
  partitionOffsets = {
    @PartitionOffset(partition = "0", initialOffset = "0"),
    @PartitionOffset(partition = "3", initialOffset = "0")}),
  containerFactory = "partitionsKafkaListenerContainerFactory")
public void listenToPartition(
  @Payload String message,
  @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition) {
      System.out.println(
        "Received Message: " + message"
        + "from partition: " + partition);
}
```

Since the *initialOffset* has been sent to 0 in this listener, all the previously consumed messages from partitions 0 and three will be re-consumed every time this listener is initialized. If setting the offset is not required, we can use the *partitions* property of *@TopicPartition* annotation to set only the partitions without the offset:

```
@KafkaListener(topicPartitions
  = @TopicPartition(topic = "topicName", partitions = { "0", "1" }))
```

5.4. Adding Message Filter for Listeners

Listeners can be configured to consume specific types of messages by adding a custom filter. This can be done by setting a *RecordFilterStrategy (https://docs.spring.io/spring-kafka/api/org/springframework/kafka/listener/adapter/RecordFilterStrategy.html)* to the *KafkaListenerContainerFactory*.

```
@Bean
public ConcurrentKafkaListenerContainerFactory<String, String>
  filterKafkaListenerContainerFactory() {

    ConcurrentKafkaListenerContainerFactory<String, String> factory =
      new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory());
    factory.setRecordFilterStrategy(
      record -> record.value().contains("World"));
    return factory;
}
```

A listener can then be configured to use this container factory:

```
@KafkaListener(
  topics = "topicName",
  containerFactory = "filterKafkaListenerContainerFactory")
public void listenWithFilter(String message) {
    System.out.println("Received Message in filtered listener: " + message);
}
```

In this listener, all the **messages matching the filter will be discarded**.

6. Custom Message Converters

So far we have only covered sending and receiving Strings as messages. However, we can also send and receive custom Java objects. This requires configuring appropriate serializer in *ProducerFactory* and deserializer in *ConsumerFactory*.

Let's look at a simple bean class, which we will send as messages:

```
public class Greeting {

    private String msg;
    private String name;

    // standard getters, setters and constructor
}
```

6.1. Producing Custom Messages

In this example, we will use *JsonSerializer (https://docs.spring.io/spring-kafka/api/org/springframework/kafka/support/serializer/JsonSerializer.html)*. Let's look at the code for *ProducerFactory* and *KafkaTemplate*:

```
@Bean
public ProducerFactory<String, Greeting> greetingProducerFactory() {
    // ...
    configProps.put(
      ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
      JsonSerializer.class);
    return new DefaultKafkaProducerFactory<>(configProps);
}

@Bean
public KafkaTemplate<String, Greeting> greetingKafkaTemplate() {
    return new KafkaTemplate<>(greetingProducerFactory());
}
```

This new *KafkaTemplate* can be used to send the *Greeting* message:

```
kafkaTemplate.send(topicName, new Greeting("Hello", "World"));
```

6.2. Consuming Custom Messages

Similarly, let's modify the *ConsumerFactory* and *KafkaListenerContainerFactory* to deserialize the Greeting message correctly:

```
@Bean
public ConsumerFactory<String, Greeting> greetingConsumerFactory() {
    // ...
    return new DefaultKafkaConsumerFactory<>(
      props,
      new StringDeserializer(),
      new JsonDeserializer<>(Greeting.class));
}

@Bean
public ConcurrentKafkaListenerContainerFactory<String, Greeting>
  greetingKafkaListenerContainerFactory() {

    ConcurrentKafkaListenerContainerFactory<String, Greeting> factory =
      new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(greetingConsumerFactory());
    return factory;
}
```

The spring-kafka JSON serializer and deserializer uses the Jackson (/jackson) library which is also an optional maven dependency for the spring-kafka project. So let's add it to our *pom.xml*:

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.9.7</version>
</dependency>
```

Instead of using the latest version of Jackson, it's recommended to use the version which is added to the *pom.xml* of spring-kafka.

Finally, we need to write a listener to consume *Greeting* messages:

```
@KafkaListener(
  topics = "topicName",
  containerFactory = "greetingKafkaListenerContainerFactory")
public void greetingListener(Greeting greeting) {
    // process greeting message
}
```

7. Conclusion

In this article, we covered the basics of Spring support for Apache Kafka. We had a brief look at the classes which are used for sending and receiving messages.

Complete source code for this article can be found over on GitHub (https://github.com/eugenp/tutorials/tree/master/spring-kafka). Before executing the code, please make sure that Kafka server is running and the topics are created manually.

X

An intro SPRING data, JPA and
Transaction Semantics Details with JPA

Get Persistence right with Spring

Enter your email address

Download Now

Comments are closed on this article!

CATEGORIES

SPRING (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/)
REST (HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/)
JAVA (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/)
SECURITY (HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/)
PERSISTENCE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/)
JACKSON (HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/)
HTTP CLIENT-SIDE (HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/)

SERIES

JAVA “BACK TO BASICS” TUTORIAL (/JAVA-TUTORIAL)
JACKSON JSON TUTORIAL (/JACKSON)
HTTPCLIENT 4 TUTORIAL (/HTTPCLIENT-GUIDE)
REST WITH SPRING TUTORIAL (/REST-WITH-SPRING-SERIES)
SPRING PERSISTENCE TUTORIAL (/PERSISTENCE-WITH-SPRING-SERIES)
SECURITY WITH SPRING (/SECURITY-SPRING)

ABOUT

ABOUT BAELDUNG (/ABOUT)
THE COURSES (HTTPS://COURSES.BAELDUNG.COM)
JOBS (/TAG/ACTIVE-JOB/)
THE FULL ARCHIVE (/FULL_ARCHIVE)
EDITORS (/EDITORS)
OUR PARTNERS (/PARTNERS)
ADVERTISE ON BAELDUNG (/ADVERTISE)

TERMS OF SERVICE (/TERMS-OF-SERVICE)
PRIVACY POLICY (/PRIVACY-POLICY)
COMPANY INFO (/BAELDUNG-COMPANY-INFO)
CONTACT (/CONTACT)