

(/)

Introducción a Apache Kafka con Spring

Última modificación: 16 de mayo de 2020

por baeldung (<https://www.baeldung.com/author/baeldung/>)
(<https://www.baeldung.com/author/baeldung/>)

Persistencia (<https://www.baeldung.com/category/persistence/>)

Primavera (<https://www.baeldung.com/category/spring/>) +

Kafka (<https://www.baeldung.com/tag/kafka/>) **Mensajería** (<https://www.baeldung.com/tag/messaging/>)

Acabo de anunciar el nuevo curso *Learn Spring*, centrado en los fundamentos de Spring 5 y Spring Boot 2:

>> VER EL CURSO (/ls-course-start)

1. Información general

Apache Kafka (<https://kafka.apache.org/>) es un sistema de procesamiento de flujo distribuido y tolerante a fallas.

En este artículo, cubriremos el soporte de Spring para Kafka y el nivel de abstracciones que proporciona sobre las API de cliente Java Kafka nativas.

Spring Kafka trae el modelo de programación de plantillas Spring simple y típico con *KafkaTemplate* y POJOs basados en *mensajes* a través de la anotación *@KafkaListener*.

Otras lecturas:

Construyendo una tubería de datos con Flink y Kafka (<https://www.baeldung.com/kafka-flink-data-pipeline>)

Aprenda a procesar datos de flujo con Flink y Kafka

Leer más (<https://www.baeldung.com/kafka-flink-data-pipeline>) →

Ejemplo de Kafka Connect con MQTT y MongoDB (<https://www.baeldung.com/kafka-connect-mqtt-mongodb>)

Eche un vistazo a un ejemplo práctico con conectores Kafka.

Leer más (<https://www.baeldung.com/kafka-connect-mqtt-mongodb>) →

2. Instalación y configuración

Para descargar e instalar Kafka, consulte la guía oficial aquí (<https://kafka.apache.org/quickstart>) .

También necesitamos agregar la dependencia *spring-kafka* a nuestro *pom.xml*:

```
1 <dependency>
2     <groupId>org.springframework.kafka</groupId>
3     <artifactId>spring-kafka</artifactId>
4     <version>2.3.7.RELEASE</version>
5 </dependency>
```

La última versión de este artefacto se puede encontrar aquí (<https://search.maven.org/classic/#search%7Cga%7C1%7Cg%3A%22org.springframework.kafka%22%20AND%20a%3A%22spring-kafka%22>) .

Nuestra aplicación de ejemplo será una aplicación Spring Boot.

Este artículo asume que el servidor se inicia utilizando la configuración predeterminada y que no se cambian los puertos del servidor.

3. Configuración de temas

Anteriormente solíamos ejecutar herramientas de línea de comandos para crear temas en Kafka como:

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

Pero con la introducción de *AdminClient* en Kafka, ahora podemos crear temas mediante programación.

Necesitamos agregar el bean *KafkaAdmin* Spring, que agregará automáticamente temas para todos los beans de tipo *NewTopic*:

```
1  @Configuration
2  public class KafkaTopicConfig {
3
4      @Value(value = "${kafka.bootstrapAddress}")
5      private String bootstrapAddress;
6
7      @Bean
8      public KafkaAdmin kafkaAdmin() {
9          Map<String, Object> configs = new HashMap<>();
10         configs.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapAddress);
11         return new KafkaAdmin(configs);
12     }
13
14     @Bean
15     public NewTopic topic1() {
16         return new NewTopic("baeldung", 1, (short) 1);
17     }
18 }
```

4. Produciendo mensajes

Para crear mensajes, primero, necesitamos configurar una *ProducerFactory* (<https://docs.spring.io/spring-kafka/api/org/springframework/kafka/core/ProducerFactory.html>) que establece la estrategia para crear instancias de Kafka *Producer* (<https://kafka.apache.org/0100/javadoc/org/apache/kafka/clients/producer/Producer.html>) .

Luego, necesitamos un *KafkaTemplate* (<https://docs.spring.io/spring-kafka/api/org/springframework/kafka/core/KafkaTemplate.html>) que envuelva una instancia de *Productor* y proporcione métodos convenientes para enviar mensajes a los temas de Kafka.

Las instancias de *producer* son seguras para subprocesos y, por lo tanto, el uso de una sola instancia en todo el contexto de una aplicación proporcionará un mayor rendimiento. En consecuencia, *las* instancias de *KafkaTemplate* también son seguras para subprocesos y se recomienda el uso de una instancia.

4.1. Configuración del productor

```
1  @Configuration
2  public class KafkaProducerConfig {
3
4      @Bean
5      public ProducerFactory<String, String> producerFactory() {
6          Map<String, Object> configProps = new HashMap<>();
7          configProps.put(
8              ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
9              bootstrapAddress);
10         configProps.put(
11             ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
12             StringSerializer.class);
13         configProps.put(
14             ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
15             StringSerializer.class);
16         return new DefaultKafkaProducerFactory<>(configProps);
17     }
18
19     @Bean
20     public KafkaTemplate<String, String> kafkaTemplate() {
21         return new KafkaTemplate<>(producerFactory());
22     }
23 }
```

4.2. Publicando mensajes

Podemos enviar mensajes usando la clase *KafkaTemplate*:

```
1  @Autowired
2  private KafkaTemplate<String, String> kafkaTemplate;
3
4  public void sendMessage(String msg) {
5      kafkaTemplate.send(topicName, msg);
6  }
```

La API de envío devuelve un objeto *ListenableFuture* . Si queremos bloquear el hilo de envío y obtener el resultado sobre el mensaje enviado, podemos llamar a la API *get* del objeto *ListenableFuture* . El hilo esperará el resultado, pero ralentizará al productor.

Kafka es una plataforma de procesamiento de flujo rápido. Por lo tanto, es una mejor idea manejar los resultados de forma asíncrona para que los mensajes posteriores no esperen el resultado del mensaje anterior. Podemos hacer esto a través de una devolución de llamada:

```
1  public void sendMessage(String message) {
2
3      ListenableFuture<SendResult<String, String>> future =
4          kafkaTemplate.send(topicName, message);
5
6      future.addCallback(new ListenableFutureCallback<SendResult<String, String>>() {
7
8          @Override
9          public void onSuccess(SendResult<String, String> result) {
10              System.out.println("Sent message=[" + message +
11                  "] with offset=[" + result.getRecordMetadata().offset() + "]);
12          }
13          @Override
14          public void onFailure(Throwable ex) {
15              System.out.println("Unable to send message=["
16                  + message + "] due to : " + ex.getMessage());
17          }
18      });
19  }
```


5. Mensajes de consumo

5.1. Configuración del consumidor

Para consumir mensajes, necesitamos configurar una *ConsumerFactory* (<https://docs.spring.io/autorepo/docs/spring-kafka-dist/1.1.3.RELEASE/api/org/springframework/kafka/core/ConsumerFactory.html>) y una *KafkaListenerContainerFactory* (<https://docs.spring.io/autorepo/docs/spring-kafka-dist/1.1.3.RELEASE/api/org/springframework/kafka/config/KafkaListenerContainerFactory.html>) . Una vez que estos beans están disponibles en la fábrica de Spring bean, los consumidores basados en POJO pueden configurarse utilizando la anotación *@KafkaListener* (<https://docs.spring.io/autorepo/docs/spring-kafka-dist/1.1.3.RELEASE/api/org/springframework/kafka/annotation/KafkaListener.html>) .

La (<https://docs.spring.io/autorepo/docs/spring-kafka-dist/1.1.3.RELEASE/api/org/springframework/kafka/annotation/EnableKafka.html>) anotación *@EnableKafka* (<https://docs.spring.io/autorepo/docs/spring-kafka-dist/1.1.3.RELEASE/api/org/springframework/kafka/annotation/EnableKafka.html>) es necesaria en la clase de configuración para permitir la detección de la anotación *@KafkaListener* en beans gestionados por resorte:

```
1  @EnableKafka
2  @Configuration
3  public class KafkaConsumerConfig {
4
5      @Bean
6      public ConsumerFactory<String, String> consumerFactory() {
7          Map<String, Object> props = new HashMap<>();
8          props.put(
9              ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
10             bootstrapAddress);
11         props.put(
12             ConsumerConfig.GROUP_ID_CONFIG,
13             groupId);
14         props.put(
15             ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
16             StringDeserializer.class);
17         props.put(
18             ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
19             StringDeserializer.class);
20         return new DefaultKafkaConsumerFactory<>(props);
21     }
22
23     @Bean
24     public ConcurrentKafkaListenerContainerFactory<String, String>
25         kafkaListenerContainerFactory() {
26
27         ConcurrentKafkaListenerContainerFactory<String, String> factory =
28             new ConcurrentKafkaListenerContainerFactory<>();
29         factory.setConsumerFactory(consumerFactory());
30         return factory;
31     }
32 }
```

5.2. Mensajes de consumo

```
1 | @KafkaListener(topics = "topicName", groupId = "foo")
2 | public void listen(String message) {
3 |     System.out.println("Received Message in group foo: " + message);
4 | }
```

Se pueden implementar varios oyentes para un tema, cada uno con un ID de grupo diferente. Además, un consumidor puede escuchar mensajes de varios temas:

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

Spring también admite la recuperación de uno o más encabezados de mensaje utilizando la anotación `@Header` (<https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/messaging/handler/annotation/Header.html>) en el oyente:

```
1 | @KafkaListener(topics = "topicName")
2 | public void listenWithHeaders(
3 |     @Payload String message,
4 |     @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition) {
5 |     System.out.println(
6 |         "Received Message: " + message
7 |         + "from partition: " + partition);
8 | }
```

5.3. Consumir mensajes de una partición específica

Como habrás notado, creamos el tema *baeldung* con solo una partición. Sin embargo, para un tema con múltiples particiones, un *@KafkaListener* puede suscribirse explícitamente a una partición particular de un tema con un desplazamiento inicial:

```
1  @KafkaListener(  
2      topicPartitions = @TopicPartition(topic = "topicName",  
3      partitionOffsets = {  
4          @PartitionOffset(partition = "0", initialOffset = "0"),  
5          @PartitionOffset(partition = "3", initialOffset = "0")  
6      })  
7      public void listenToParition(  
8          @Payload String message,  
9          @Header(KafkaHeaders.RECEIVED_PARTITION_ID) int partition) {  
10         System.out.println(  
11             "Received Message: " + message"  
12             + "from partition: " + partition);  
13     }
```

Dado que *initialOffset* se ha enviado a 0 en esta escucha, todos los mensajes consumidos previamente de las particiones 0 y tres se volverán a consumir cada vez que se inicialice esta escucha. Si el ajuste no se requiere el desplazamiento, podemos utilizar la *particiones* propiedad de *@TopicPartition* anotación a conjunto sólo las particiones sin la compensación:

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

5.4. Agregar filtro de mensajes para oyentes

Los oyentes pueden configurarse para consumir tipos específicos de mensajes agregando un filtro personalizado. Esto se puede hacer estableciendo una *RecordFilterStrategy* (<https://docs.spring.io/spring-kafka/api/org/springframework/kafka/listener/adapter/RecordFilterStrategy.html>) en *KafkaListenerContainerFactory*:

```
1  @Bean
2  public ConcurrentKafkaListenerContainerFactory<String, String>
3      filterKafkaListenerContainerFactory() {
4
5      ConcurrentKafkaListenerContainerFactory<String, String> factory =
6          new ConcurrentKafkaListenerContainerFactory<>();
7      factory.setConsumerFactory(consumerFactory());
8      factory.setRecordFilterStrategy(
9          record -> record.value().contains("World"));
10     return factory;
11 }
```

Un oyente puede configurarse para usar esta fábrica de contenedores:

```
1  @KafkaListener(
2      topics = "topicName",
3      containerFactory = "filterKafkaListenerContainerFactory")
4  public void listen(String message) {
5      // handle message
6  }
```

En esta escucha, **se descartarán** todos los **mensajes que coincidan con el filtro** .

6. Convertidores de mensajes personalizados

Hasta ahora solo hemos cubierto el envío y la recepción de cadenas como mensajes. Sin embargo, también podemos enviar y recibir objetos Java personalizados. Esto requiere configurar el serializador apropiado en *ProducerFactory* y el deserializador en *ConsumerFactory*.

Veamos una clase de bean simple , que enviaremos como mensajes:

```
1 public class Greeting {  
2  
3     private String msg;  
4     private String name;  
5  
6     // standard getters, setters and constructor  
7 }
```

6.1. Produciendo mensajes personalizados

En este ejemplo, usaremos *JsonSerializer* (<https://docs.spring.io/spring-kafka/api/org/springframework/kafka/support/serializer/JsonSerializer.html>). Veamos el código de *ProducerFactory* y *KafkaTemplate*:

```
1  @Bean
2  public ProducerFactory<String, Greeting> greetingProducerFactory() {
3      // ...
4      configProps.put(
5          ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
6          JsonSerializer.class);
7      return new DefaultKafkaProducerFactory<>(configProps);
8  }
9
10 @Bean
11 public KafkaTemplate<String, Greeting> greetingKafkaTemplate() {
12     return new KafkaTemplate<>(greetingProducerFactory());
13 }
```

Este nuevo *KafkaTemplate* se puede utilizar para enviar el mensaje de *saludo*:

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

6.2. Consumir mensajes personalizados

Del mismo modo, modifiquemos *ConsumerFactory* y *KafkaListenerContainerFactory* para deserializar el mensaje de saludo correctamente:

```
1  @Bean
2  public ConsumerFactory<String, Greeting> greetingConsumerFactory() {
3      // ...
4      return new DefaultKafkaConsumerFactory<>(
5          props,
6          new StringDeserializer(),
7          new JsonSerializer<>(Greeting.class));
8  }
9
10 @Bean
11 public ConcurrentKafkaListenerContainerFactory<String, Greeting>
12     greetingKafkaListenerContainerFactory() {
13
14     ConcurrentKafkaListenerContainerFactory<String, Greeting> factory =
15         new ConcurrentKafkaListenerContainerFactory<>();
16     factory.setConsumerFactory(greetingConsumerFactory());
17     return factory;
18 }
```

El serializador y deserializador spring-kafka JSON utiliza la biblioteca Jackson (/jackson) , que también es una dependencia opcional de Maven para el proyecto spring-kafka. Así que vamos a agregarlo a nuestro *pom.xml* :

```
1  <dependency>
2      <groupId>com.fasterxml.jackson.core</groupId>
3      <artifactId>jackson-databind</artifactId>
4      <version>2.9.7</version>
5  </dependency>
```

En lugar de usar la última versión de Jackson, se recomienda usar la versión que se agrega al *pom.xml* de spring-kafka.

Finalmente, necesitamos escribir un oyente para consumir mensajes de *saludo* :


```
1  @KafkaListener(  
2      topics = "topicName",  
3      containerFactory = "greetingKafkaListenerContainerFactory")  
4  public void greetingListener(Greeting greeting) {  
5      // process greeting message  
6  }
```

7. Conclusión

En este artículo, cubrimos los conceptos básicos del soporte de Spring para Apache Kafka. Tuvimos un breve vistazo a las clases que se utilizan para enviar y recibir mensajes.

El código fuente completo de este artículo se puede encontrar en GitHub (<https://github.com/eugenp/tutorials/tree/master/spring-kafka>). Antes de ejecutar el código, asegúrese de que el servidor Kafka se esté ejecutando y que los temas se creen manualmente.

Acabo de anunciar el nuevo curso *Learn Spring*, centrado en los fundamentos de Spring 5 y Spring Boot 2:

>> VER EL CURSO (/ls-course-end)




Una introducción de datos **SPRING**, JPA y
detalles de semántica de transacción con JPA

Haz persistencia correctamente
con Spring

Ingrese su dirección de correo electrónico

Descargar ahora

¡Los comentarios están cerrados en este artículo!

 **ezoic** (<https://www.ezoic.com/what-is-ezoic/>)

[reportar este anuncio](#)

CATEGORIAS

PRIMAVERA ([HTTPS://WWW.BAELDUNG.COM/CATEGORY/SPRING/](https://www.baeldung.com/category/spring/))

DESCANSO ([HTTPS://WWW.BAELDUNG.COM/CATEGORY/REST/](https://www.baeldung.com/category/rest/))

[JAVA \(HTTPS://WWW.BAELDUNG.COM/CATEGORY/JAVA/\)](https://www.baeldung.com/category/java/)

[SEGURIDAD \(HTTPS://WWW.BAELDUNG.COM/CATEGORY/SECURITY-2/\)](https://www.baeldung.com/category/security-2/)

[PERSISTENCIA \(HTTPS://WWW.BAELDUNG.COM/CATEGORY/PERSISTENCE/\)](https://www.baeldung.com/category/persistence/)

[JACKSON \(HTTPS://WWW.BAELDUNG.COM/CATEGORY/JSON/JACKSON/\)](https://www.baeldung.com/category/json/jackson/)

[HTTP DEL LADO DEL CLIENTE \(HTTPS://WWW.BAELDUNG.COM/CATEGORY/HTTP/\)](https://www.baeldung.com/category/http/)

[KOTLIN \(HTTPS://WWW.BAELDUNG.COM/CATEGORY/KOTLIN/\)](https://www.baeldung.com/category/kotlin/)

SERIE

[TUTORIAL DE JAVA "VOLVER A LO BÁSICO" \(/JAVA-TUTORIAL\)](#)

[JACKSON JSON TUTORIAL \(/JACKSON\)](#)

[HTTPCLIENT 4 TUTORIAL \(/HTTPCLIENT-GUIDE\)](#)

[RESTO CON SPRING TUTORIAL \(/REST-WITH-SPRING-SERIES\)](#)

[TUTORIAL SPRING PERSISTENCE \(/PERSISTENCE-WITH-SPRING-SERIES\)](#)

[SEGURIDAD CON PRIMAVERA \(/SECURITY-SPRING\)](#)

ACERCA DE

[SOBRE BAELDUNG \(/ABOUT\)](#)

[LOS CURSOS \(HTTPS://COURSES.BAELDUNG.COM\)](https://courses.baeldung.com)

[TRABAJOS \(/TAG/ACTIVE-JOB/\)](#)

[EL ARCHIVO COMPLETO \(/FULL_ARCHIVE\)](#)

[ESCRIBIR PARA BAELDUNG \(/CONTRIBUTION-GUIDELINES\)](#)

[EDITORES \(/EDITORS\)](#)

[NUESTROS COMPAÑEROS \(/PARTNERS\)](#)

[ANUNCIE EN BAELDUNG \(/ADVERTISE\)](#)

TÉRMINOS DE SERVICIO (/TERMS-OF-SERVICE)

POLÍTICA DE PRIVACIDAD (/PRIVACY-POLICY)

INFORMACIÓN DE LA COMPAÑÍA (/BAELDUNG-COMPANY-INFO)

CONTACTO (/CONTACT)