

(/)

Introducción a Spring Cloud Stream

Última modificación: 27 de febrero de 2019

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

Nube de primavera (<https://www.baeldung.com/category/spring/spring-cloud/>)

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

Spring Cloud Stream es un marco creado sobre Spring Boot y Spring Integration que **ayuda a crear microservicios basados en eventos o mensajes**.

En este artículo, presentaremos conceptos y construcciones de Spring Cloud Stream con algunos ejemplos simples.



2. Dependencias de Maven

Para comenzar, necesitaremos agregar Spring Cloud Starter Stream con la (<https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%20AND%20%3A%22spring-cloud-starter-stream-rabbit%22>) dependencia del intermediario RabbitMQ (<https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%20AND%20%3A%22spring-cloud-starter-stream-rabbit%22>) Maven como middleware de mensajería a nuestro *pom.xml*:

```

1 <dependency>
2   <groupId>org.springframework.cloud</groupId>
3   <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
4   <version>1.3.0.RELEASE</version>
5 </dependency>

```

Y agregaremos la dependencia del módulo de Maven Central

(<https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%20AND%20a%3A%22spring-cloud-stream-test-support%22>) para habilitar el soporte JUnit también:

```

1 <dependency>
2   <groupId>org.springframework.cloud</groupId>
3   <artifactId>spring-cloud-stream-test-support</artifactId>
4   <version>1.3.0.RELEASE</version>
5   <scope>test</scope>
6 </dependency>

```

3. Conceptos principales

La arquitectura de microservicios sigue el principio de " puntos finales inteligentes y tuberías tontas" (<https://martinfowler.com/articles/microservices.html#SmartEndpointsAndDumbPipes>). La comunicación entre puntos finales es impulsada por partes de middleware de mensajería como RabbitMQ o Apache Kafka.

Los servicios se comunican publicando eventos de dominio a través de estos puntos finales o canales .

Veamos los conceptos que componen el marco de Spring Cloud Stream, junto con los paradigmas esenciales que debemos tener en cuenta para crear servicios basados en mensajes.

3.1. Construcciones

Veamos un servicio simple en Spring Cloud Stream que escucha *el* enlace de *entrada* y envía una respuesta al enlace de *salida* :

```

1 @SpringBootApplication
2 @EnableBinding(Processor.class)
3 public class MyLoggerServiceApplication {
4     public static void main(String[] args) {
5         SpringApplication.run(MyLoggerServiceApplication.class, args);
6     }
7
8     @StreamListener(Processor.INPUT)
9     @SendTo(Processor.OUTPUT)
10    public LogMessage enrichLogMessage(LogMessage log) {
11        return new LogMessage(String.format("[1]: %s", log.getMessage()));
12    }
13 }

```

La anotación `@EnableBinding` configura la aplicación para enlazar los canales `INPUT` y `OUTPUT` definidos dentro del *procesador de interfaz*. **Ambos canales son enlaces que se pueden configurar para usar un mensaje intermedio-middleware o aglutinante.**

Echemos un vistazo a la definición de todos estos conceptos:

- *Enlaces* : una colección de interfaces que identifican los canales de entrada y salida de forma declarativa
- *Binder* : implementación de middleware de mensajería como Kafka o RabbitMQ
- *Canal* : representa el conducto de comunicación entre messaging-middleware y la aplicación
- *StreamListeners* : métodos de manejo de mensajes en beans que se invocarán automáticamente en un mensaje del canal después de que *MessageConverter* realice la serialización / deserialización entre eventos específicos de middleware y tipos de objetos de dominio / POJO
- *Esquemas Message* : utilizados para la serialización y deserialización de mensajes, estos esquemas pueden leerse estáticamente desde una ubicación o cargarse dinámicamente, lo que respalda la evolución de los tipos de objetos de dominio

3.2. Patrones de comunicación

Los mensajes designados a los destinos se entregan mediante el patrón de mensajería **Publicar-Suscribir**. Los editores clasifican los mensajes en temas, cada uno identificado por un nombre. Los suscriptores expresan interés en uno o más temas. El middleware filtra los mensajes, entregando los temas interesantes a los suscriptores.

Ahora, los suscriptores podrían agruparse. Un *grupo de consumidores* es un conjunto de suscriptores o consumidores, identificados por una identificación de *grupo*, dentro de los cuales los mensajes de un tema o partición de tema se entregan de manera equilibrada.

4. Modelo de programación

Esta sección describe los conceptos básicos de la construcción de aplicaciones Spring Cloud Stream.

4.1. Pruebas Funcionales

El soporte de prueba es una implementación de carpeta que permite interactuar con los canales e inspeccionar mensajes.

Envíemos un mensaje al servicio *enrichLogMessage* anterior y verifiquemos si la respuesta contiene el texto "[1]:" al comienzo del mensaje:



```
1  @RunWith(SpringJUnit4ClassRunner.class)
2  @ContextConfiguration(classes = MyLoggerServiceApplication.class)
3  @DirtiesContext
4  public class MyLoggerApplicationTests {
5
6      @Autowired
7      private Processor pipe;
8
9      @Autowired
10     private MessageCollector messageCollector;
11
12     @Test
13     public void whenSendMessage_thenResponseShouldUpdateText() {
14         pipe.input()
15             .send(MessageBuilder.withPayload(new LogMessage("This is my message"))
16                 .build());
17
18         Object payload = messageCollector.forChannel(pipe.output())
19             .poll()
20             .getPayload();
21
22         assertEquals("[1]: This is my message", payload.toString());
23     }
24 }
```

4.2. Canales personalizados

En el ejemplo anterior, utilizamos la interfaz del *procesador* proporcionada por Spring Cloud, que tiene solo un canal de entrada y un canal de salida.

Si necesitamos algo diferente, como una entrada y dos canales de salida, podemos crear un procesador personalizado:

```

1 public interface MyProcessor {
2     String INPUT = "myInput";
3
4     @Input
5     SubscribableChannel myInput();
6
7     @Output("myOutput")
8     MessageChannel anOutput();
9
10    @Output
11    MessageChannel anotherOutput();
12 }

```

Spring nos proporcionará la implementación adecuada de esta interfaz. Los nombres de los canales se pueden configurar mediante anotaciones como en `@Output("myOutput")`.

De lo contrario, Spring usará los nombres de los métodos como los nombres de los canales. Por lo tanto, tenemos tres canales llamados *myInput*, *myOutput* y *otroOutput*.

Ahora, imaginemos que queremos enrutar los mensajes a una salida si el valor es menor que 10 y a otra salida si el valor es mayor o igual a 10:

```

1 @Autowired
2 private MyProcessor processor;
3
4 @StreamListener(MyProcessor.INPUT)
5 public void routeValues(Integer val) {
6     if (val < 10) {
7         processor.anOutput().send(message(val));
8     } else {
9         processor.anotherOutput().send(message(val));
10    }
11 }
12
13 private static final <T> Message<T> message(T val) {
14     return MessageBuilder.withPayload(val).build();
15 }

```

4.3. Despacho Condicional

Usando la anotación `@StreamListener`, también podemos **filtrar los mensajes que esperamos en el consumidor** usando cualquier condición que definamos con expresiones SpEL (`/spring-expression-language`).

Como ejemplo, podríamos usar el despacho condicional como otro enfoque para enrutar mensajes a diferentes salidas:

```

1 @Autowired
2 private MyProcessor processor;
3
4 @StreamListener(
5     target = MyProcessor.INPUT,
6     condition = "payload < 10")
7 public void routeValuesToAnOutput(Integer val) {
8     processor.anOutput().send(message(val));
9 }
10
11 @StreamListener(
12     target = MyProcessor.INPUT,
13     condition = "payload >= 10")
14 public void routeValuesToAnotherOutput(Integer val) {
15     processor.anotherOutput().send(message(val));
16 }

```

La única **limitación de este enfoque es que estos métodos no deben devolver un valor.**

5. Configuración

Configuremos la aplicación que procesará el mensaje del corredor RabbitMQ.

5.1. Configuración de carpeta

Podemos configurar nuestra aplicación para usar la implementación predeterminada de la carpeta a través de *META-INF / spring.binders*:

```
1 rabbit:\n2 org.springframework.cloud.stream.binder.rabbit.config.RabbitMessageChannelBinderConfiguration
```

O podemos agregar la biblioteca de carpetas para RabbitMQ al classpath incluyendo *esta dependencia* (<https://search.maven.org/classic/#search%7Cgav%7C1%7Cg%3A%22org.springframework.cloud%22%20AND%20a%3A%22spring-cloud-stream-binder-rabbit%22>):

```
1 <dependency>\n2   <groupId>org.springframework.cloud</groupId>\n3   <artifactId>spring-cloud-stream-binder-rabbit</artifactId>\n4   <version>1.3.0.RELEASE</version>\n5 </dependency>
```

Si no se proporciona la implementación de la carpeta, Spring utilizará la comunicación directa de mensajes entre los canales.

5.2. Configuración RabbitMQ

Para configurar el ejemplo en la sección 3.1 para usar el cuaderno RabbitMQ, necesitamos actualizar el archivo *application.yml* ubicado en *src / main / resources*:

```
|
```

```

1  spring:
2    cloud:
3      stream:
4        bindings:
5          input:
6            destination: queue.log.messages
7            binder: local_rabbit
8          output:
9            destination: queue.pretty.log.messages
10           binder: local_rabbit
11        binders:
12          local_rabbit:
13            type: rabbit
14            environment:
15              spring:
16                rabbitmq:
17                  host: <host>
18                  port: 5672
19                  username: <username>
20                  password: <password>
21                  virtual-host: /

```

El enlace de *entrada* usará el intercambio llamado *queue.log.messages* , y el enlace de *salida* usará el intercambio *queue.pretty.log.messages* . Ambas vinculaciones utilizarán la carpeta llamada *local_rabbit* .

Tenga en cuenta que no necesitamos crear los intercambios o colas RabbitMQ por adelantado. Al ejecutar la aplicación, **ambos intercambios se crean automáticamente** .

Para probar la aplicación, podemos usar el sitio de administración RabbitMQ para publicar un mensaje. En el panel *Publicar mensaje* del intercambio *queue.log.messages* , necesitamos ingresar la solicitud en formato JSON.

5.3. Personalizar la conversión de mensajes

Spring Cloud Stream nos permite aplicar la conversión de mensajes para tipos de contenido específicos. En el ejemplo anterior, en lugar de usar el formato JSON, queremos proporcionar texto sin formato.

Para hacer esto, **aplicaremos una transformación personalizada a *LogMessage* usando un *MessageConverter*** :

```

1  @SpringBootApplication
2  @EnableBinding(Processor.class)
3  public class MyLoggerServiceApplication {
4      //...
5
6      @Bean
7      public MessageConverter providesTextPlainMessageConverter() {
8          return new TextPlainMessageConverter();
9      }
10
11     //...
12 }

```

```

1 public class TextPlainMessageConverter extends AbstractMessageConverter {
2
3     public TextPlainMessageConverter() {
4         super(new MimeTypes("text", "plain"));
5     }
6
7     @Override
8     protected boolean supports(Class<?> clazz) {
9         return (LogMessage.class == clazz);
10    }
11
12    @Override
13    protected Object convertFromInternal(Message<?> message,
14        Class<?> targetClass, Object conversionHint) {
15        Object payload = message.getPayload();
16        String text = payload instanceof String
17            ? (String) payload
18            : new String((byte[]) payload);
19        return new LogMessage(text);
20    }
21 }

```

Después de aplicar estos cambios, volviendo al panel *Publicar mensaje*, si configuramos el encabezado `contentType` en `text/plain` y la carga útil en `Hello World`, debería funcionar como antes.

5.4. Grupos de consumidores

Cuando se ejecutan varias instancias de nuestra aplicación, **cada vez que hay un nuevo mensaje en un canal de entrada, se notificará a todos los suscriptores**.

La mayoría de las veces, necesitamos que el mensaje se procese solo una vez. Spring Cloud Stream implementa este comportamiento a través de grupos de consumidores.

Para habilitar este comportamiento, cada enlace de consumidor puede usar la *propiedad* `spring.cloud.stream.bindings.<CHANNEL>.group` para especificar un nombre de grupo:

```

1 spring:
2   cloud:
3     stream:
4       bindings:
5         input:
6           destination: queue.log.messages
7           binder: local_rabbit
8           group: logMessageConsumers
9           ...

```

6. Microservicios basados en mensajes

En esta sección, presentamos todas las características requeridas para ejecutar nuestras aplicaciones Spring Cloud Stream en un contexto de microservicios.

6.1. Ampliar

Cuando se ejecutan varias aplicaciones, es importante asegurarse de que los datos se dividan correctamente entre los consumidores. Para hacerlo, Spring Cloud Stream ofrece dos propiedades:

- **`spring.cloud.stream.instanceCount`**: número de aplicaciones en ejecución
- **`spring.cloud.stream.instanceIndex`** - índice de la aplicación actual

Por ejemplo, si hemos implementado dos instancias de la aplicación *MyLoggerServiceApplication* anterior, la propiedad `spring.cloud.stream.instanceCount` debería ser 2 para ambas aplicaciones, y la propiedad `spring.cloud.stream.instanceIndex` debería ser 0 y 1 respectivamente.

Estas propiedades se establecen automáticamente si implementamos las aplicaciones Spring Cloud Stream utilizando Spring Data Flow como se describe en este artículo ([/spring-cloud-data-flow-stream-processing](#)) .

6.2. Fraccionamiento

Los eventos de dominio pueden ser mensajes *particionados* . Esto ayuda cuando estamos **ampliando el almacenamiento y mejorando el rendimiento de la aplicación** .

El evento de dominio generalmente tiene una clave de partición para que termine en la misma partición con mensajes relacionados.

Digamos que queremos que los mensajes de registro se particionen por la primera letra del mensaje, que sería la clave de partición, y se agrupen en dos particiones.

Habría una partición para los mensajes de registro que comienzan con *AM* y otra partición para *NZ*. Esto se puede configurar usando dos propiedades:

- `spring.cloud.stream.bindings.output.producer.partitionKeyExpression` - la expresión para particionar las cargas útiles
- `spring.cloud.stream.bindings.output.producer.partitionCount` - el número de grupos

A veces, la expresión para particionar es demasiado compleja para escribirla en una sola línea. Para estos casos, podemos escribir nuestra estrategia de partición personalizada utilizando la propiedad `spring.cloud.stream.bindings.output.producer.partitionKeyExtractorClass` .

6.3. Indicador de salud

En un contexto de microservicios, también necesitamos **detectar cuándo un servicio está inactivo o comienza a fallar** . Spring Cloud Stream proporciona la propiedad `management.health.binders.enabled` para habilitar los indicadores de salud para las carpetas.

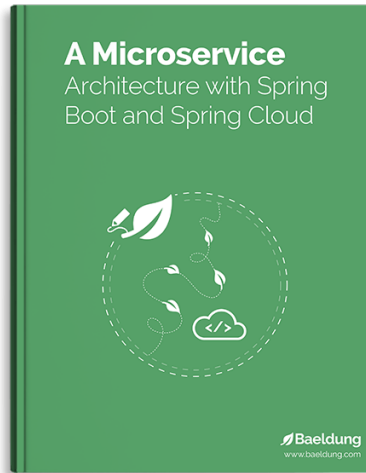
Al ejecutar la aplicación, podemos consultar el estado de salud en `http://<host>:<port>/health` .

7. Conclusión

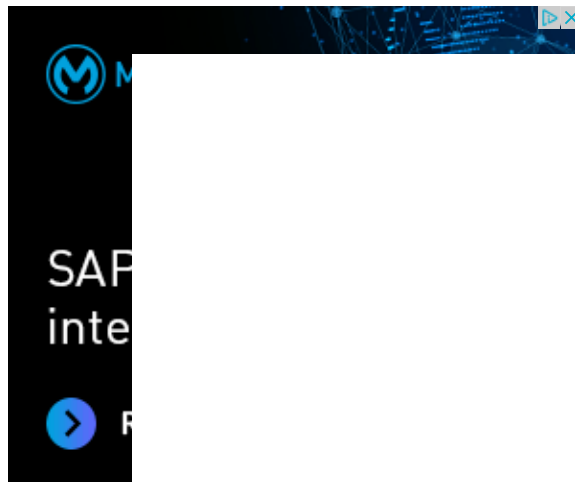
En este tutorial, presentamos los conceptos principales de Spring Cloud Stream y mostramos cómo usarlo a través de algunos ejemplos simples sobre RabbitMQ. Puede encontrar más información sobre Spring Cloud Stream aquí (<https://docs.spring.io/spring-cloud-stream/docs/current/reference/htmlsingle/>) .


El código fuente de este artículo se puede encontrar en GitHub (<https://github.com/eugenp/tutorials/tree/master/spring-cloud/spring-cloud-stream/spring-cloud-stream-rabbit>) .

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

[>> VER EL CURSO \(/ls-course-end\)](#)

Cree su arquitectura de microservicios con
Spring Boot y Spring Cloud

[>> Descargar ahora](#)[▲ el más nuevo](#) [▲ más antiguo](#) [▲ más votado](#)




Invitado

ayuda (<http://help.com>)

El mensaje de registro está en desuso y no podemos ejecutar lo anterior localmente

+ 0 -

Hace 2 años



(<https://www.baeldung.com/author/grzegorz-authors/>)

Invitado

Grzegorz Piwowarek (<http://4comprehension.com>)

¿Con qué versión estás trabajando?

+ 0 -

Hace 2 años

¡Los comentarios están cerrados en este artículo!

CATEGORÍAS

- 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)

[TRABAJO DE CONSULTORÍA \(/CONSULTING\)](#)

[META BAELDUNG \(HTTP://META.BAELDUNG.COM/\)](http://meta.baeldung.com/)

[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 \(/CONTACT\)](#)