UNIVERSIDAD DE LAS AMERICAS

Facultad de Ingenierías y Ciencias Aplicadas

Ingeniería de Software

Docente: Darío Villamarin

Alumno/s: Mateo Cartagena

David Terán

Martin Zumarraga

Tarea: Taller - Implementación del patrón Pub/Sub con Apache Camel y RabbitMQ

Objetivo:

Simular un sistema de notificaciones en el que:

- Un publicador emite alertas cada 5 segundos.
- Dos suscriptores (consumidores) reciben el mismo mensaje, de forma desacoplada.

1. Introducción 3
2. Marco Teórico 4
2.1. Sistemas de Mensajería (Message-Oriented Middleware - MOM) 4
2.2. El Patrón Productor-Consumidor 4
2.3. Desacoplamiento en Sistemas de Software y sus Ventajas 4
2.4. Patrón de Integración Publicación/Suscripción (Pub/Sub) 5
3. Tecnologías y Componentes Utilizados 6
3.1. RabbitMQ 6
3.2. Spring Boot 6
3.3. Apache Camel 7
4. Análisis de la Implementación 7
4.1. Estructura del Proyecto y Dependencias (pom.xml) 7
4.2. El Publicador (PublisherRoute.java) 7
4.3. Los Consumidores (Consumer1Route.java y Consumer2Route.java) 8
4.4. Control Dinámico de Rutas (ConsumerController.java) 10
5. Proceso y Relación entre Componentes 10
5.1. Flujo de Mensajes 10
5.2. Demostración del Desacoplamiento 11
6. Conclusiones 12
7. Referencias 12

1. Introducción

En el desarrollo de software moderno, la integración de sistemas distribuidos es un pilar fundamental. Los sistemas monolíticos están siendo reemplazados por arquitecturas de microservicios, donde componentes independientes necesitan comunicarse de manera eficiente y confiable. Una comunicación fuertemente acoplada, donde los servicios se llaman directamente entre sí, genera fragilidad, dificulta la escalabilidad y complica el mantenimiento.

Para resolver este desafío, los patrones de integración basados en mensajería asíncrona se han vuelto esenciales. Estos patrones utilizan un intermediario (message broker) para gestionar la comunicación, eliminando las dependencias directas entre los componentes del sistema.

El objetivo de este informe es documentar y analizar la implementación práctica del patrón de integración **Publicación/Suscripción (Pub/Sub)**. Para ello, se ha desarrollado una aplicación utilizando un stack tecnológico compuesto por **Spring Boot** como marco de aplicación, **Apache Camel** como framework de integración y **RabbitMQ** como message broker. El sistema simula un escenario donde un publicador emite alertas de forma periódica, y múltiples consumidores, cada uno con una responsabilidad distinta, reciben y procesan dichas alertas.

Adicionalmente, para evidenciar el principio de desacoplamiento, se ha implementado una funcionalidad clave: un endpoint de API que permite pausar y reanudar consumidores de forma individual en tiempo de ejecución. Esto demuestra de manera tangible cómo la falla o el mantenimiento de un componente no afecta al resto del sistema, una de las ventajas más significativas de esta arquitectura.

2. Marco Teórico

2.1. Sistemas de Mensajería (Message-Oriented Middleware - MOM)

Los Sistemas de Mensajería, o Middleware Orientado a Mensajes (MOM), son una categoría de software que facilita la comunicación entre aplicaciones distribuidas. En lugar de que las aplicaciones se comuniquen directamente, envían y reciben datos en forma de mensajes a través de este middleware. El MOM es responsable de garantizar que los mensajes se entreguen de manera confiable desde el emisor (productor) al receptor (consumidor), gestionando colas, protocolos de red y la entrega asíncrona.

2.2. El Patrón Productor-Consumidor

Este es un patrón de diseño fundamental en computación concurrente. Describe dos tipos de procesos, el **productor** y el **consumidor**, que comparten un búfer o cola común.

- **Productor:** Su única función es generar datos o "mensajes" y depositarlos en la cola.
- **Consumidor:** Su función es extraer mensajes de la cola y procesarlos.

La clave de este patrón es que el productor y el consumidor operan a ritmos diferentes e independientes, utilizando la cola como un mecanismo de amortiguación y sincronización.

2.3. Desacoplamiento en Sistemas de Software y sus Ventajas

El desacoplamiento se refiere al grado de interdependencia entre los módulos de un sistema. Un sistema desacoplado es aquel en el que los componentes tienen un conocimiento mínimo o nulo sobre los detalles internos de otros componentes. La comunicación se realiza a través de interfaces bien definidas, como las que proporciona un sistema de mensajería.

Las ventajas de un alto desacoplamiento son:

- **Resiliencia:** Si un consumidor falla o se desconecta, el productor y los demás consumidores pueden seguir funcionando sin interrupciones. El message broker puede retener los mensajes hasta que el consumidor esté disponible nuevamente.
- **Escalabilidad:** Se pueden añadir nuevos consumidores para procesar la misma información y así escalar horizontalmente la capacidad de procesamiento, sin necesidad de modificar el productor.
- **Flexibilidad y Mantenimiento:** Los componentes pueden ser desarrollados, desplegados y actualizados de forma independiente. Un consumidor puede ser reescrito en otro lenguaje de programación sin que el resto del sistema se vea afectado.

2.4. Patrón de Integración Publicación/Suscripción (Pub/Sub)

El patrón Publicación/Suscripción es una evolución del modelo productor-consumidor. En este modelo, el emisor del mensaje, llamado **publicador (publisher)**, no envía los mensajes directamente a los receptores. En su lugar, clasifica los mensajes en canales o "tópicos" (en RabbitMQ, esto se gestiona a través de un **exchange**), sin saber qué suscriptores (si los hay) recibirán el mensaje.

Por otro lado, los **suscriptores (subscribers)** expresan su interés en uno o más tópicos y solo reciben los mensajes que son de su interés, sin saber quiénes son los publicadores. El message broker se encarga de filtrar y dirigir los mensajes desde los publicadores a todos los suscriptores interesados. Esto permite que un único mensaje sea consumido por múltiples sistemas para diferentes propósitos.

3. Tecnologías y Componentes Utilizados

3.1. RabbitMQ

Es un message broker de código abierto, uno de los más populares del mercado. Implementa el protocolo AMQP (Advanced Message Queuing Protocol) y es ideal para arquitecturas complejas. En este proyecto, se utiliza para:

- **Exchange:** Recibe los mensajes del publicador. Se configuró un exchange de tipo `fanout`, que retransmite cada mensaje que recibe a todas las colas que están vinculadas a él.
- **Queue:** Almacena los mensajes para los consumidores. Cada consumidor en nuestro sistema tiene su propia cola exclusiva, lo que garantiza que cada uno reciba una copia del mensaje.

3.2. Spring Boot

Es un marco de trabajo que simplifica la creación de aplicaciones autocontenidas y listas para producción basadas en Spring. Su rol en el proyecto es:

- Configurar y ejecutar la aplicación (`App.java`).
- Gestionar las dependencias del proyecto a través de Maven (`pom.xml`).
- Proveer un servidor web embebido (Tomcat) para exponer la API REST de control de consumidores (`ConsumerController`).

3.3. Apache Camel

Es un potente framework de integración de código abierto basado en los conocidos *Enterprise Integration Patterns*. Permite definir la lógica de enrutamiento y mediación de mensajes de una manera declarativa y concisa. En este proyecto, se utiliza para:

- **Rutas (Routes):** Definen el flujo de los mensajes. Hemos definido tres rutas principales: una para el publicador y dos para los consumidores.
- **Componentes:** Camel utiliza componentes como conectores para interactuar con diferentes tecnologías. Se usó `camel-rabbitmq` para conectar con RabbitMQ, `timer` para generar mensajes periódicos y `log` para mostrar la salida en consola.
- **Contexto (`CamelContext`):** Es el motor de Camel que gestiona y ejecuta las rutas. Se inyecta en el `ConsumerController` para permitir la manipulación de las rutas en tiempo de ejecución.

4. Análisis de la Implementación

El código proporcionado estructura una aplicación completa que demuestra el patrón Pub/Sub.

4.1. Estructura del Proyecto y Dependencias (`pom.xml`)

El archivo `pom.xml` define las dependencias necesarias para el proyecto:

- `camel-spring-boot-starter`: Integra Apache Camel con el ecosistema de Spring Boot.
- `camel-rabbitmq-starter`: Proporciona el componente de Camel para la interacción con RabbitMQ.
- `spring-boot-starter-web`: Habilita las capacidades web para crear el `RestController` que permite controlar las rutas.

La configuración es estándar para una aplicación de Spring Boot con integración de Camel.

4.2. El Publicador (`PublisherRoute.java`)

Esta clase define la lógica del publicador de mensajes.

Java

```
// PublisherRoute.java
from("timer:alerta?period=15000")
    .setBody().simple("🚨 Alerta generada: ${date:now:yyyy-MM-dd
```

```
HH:mm:ss}")
    .log(" 📡 Publicando mensaje: ${body}")
    .to("rabbitmq:alert-exchange?exchangeType=fanout&...");
```

- `from("timer:alerta?period=15000")`: El punto de partida de la ruta es un temporizador que se activa cada 15,000 milisegundos (15 segundos).
- `.setBody().simple(...)`: Se construye el contenido del mensaje. En este caso, un texto simple con la fecha y hora actuales.
- `.to("rabbitmq:alert-exchange?exchangeType=fanout...")`: El destino del mensaje es un exchange de RabbitMQ llamado `alert-exchange`. El parámetro `exchangeType=fanout` es crucial, ya que instruye a RabbitMQ para que envíe el mensaje a todas las colas vinculadas a este exchange. El publicador no necesita saber cuáles o cuántas colas existen.

4.3. Los Consumidores (`Consumer1Route.java` y `Consumer2Route.java`)

Existen dos consumidores, cada uno definido en su propia clase. Su estructura es similar pero con diferencias clave.

Consumer1Route:

Java

```
// Consumer1Route.java
from("rabbitmq:alert-
exchange?queue=console.consumer.queue&exchangeType=fanout&...")
    .routeId("consumer1")
    .log(" 📥 [Consumer 1] Mensaje recibido: ${body}");
```

Consumer2Route:

Java

```
// Consumer2Route.java
from("rabbitmq:alert-
exchange?queue=email.consumer.queue&exchangeType=fanout&...")
    .routeId("consumer2")
    .log("📨 [Consumer 2] Simulando envío por email: ${body}");
```

Análisis de los puntos clave:

- `from("rabbitmq:alert-exchange?...")`: Ambos consumidores escuchan del mismo `alert-exchange`.
- `queue=console.consumer.queue` y `queue=email.consumer.queue`: **Este es el aspecto fundamental del patrón.** Cada consumidor define su propia cola privada. Cuando el exchange `fanout` recibe un mensaje, lo distribuye a *ambas* colas. De esta forma, cada consumidor recibe una copia del mensaje para procesarla de manera independiente.
- `.routeId(...)`: Se asigna un identificador único a cada ruta (`consumer1` y `consumer2`). Este ID es el que se utilizará en la API para pausar o reanudar un consumidor específico.
- `.log(...)`: Cada consumidor realiza una acción diferente (simulada). El primero simplemente lo muestra en la consola, y el segundo simula un envío por correo electrónico.

4.4. Control Dinámico de Rutas (`ConsumerController.java`)

Esta clase expone una API REST para gestionar el ciclo de vida de las rutas de Camel.

Java
```
// ConsumerController.java
@RestController
public class ConsumerController {

    @Autowired
    private CamelContext camelContext;

    @GetMapping("/pause/{routeId}")
    public String pauseConsumer(@PathVariable String routeId) throws
Exception {
        camelContext.getRouteController().stopRoute(routeId);
        // ...
    }

    @GetMapping("/resume/{routeId}")
    public String resumeConsumer(@PathVariable String routeId)
throws Exception {
        camelContext.getRouteController().startRoute(routeId);
        // ...
    }
}
```

- Inyecta el `CamelContext`, que es el corazón del motor de Camel.
- Define dos endpoints: `/pause/{routeId}` y `/resume/{routeId}`.
- Utiliza `camelContext.getRouteController()` para acceder a los controles de las rutas y llama a `stopRoute(routeId)` o `startRoute(routeId)` según corresponda. Esto permite una gestión administrativa en vivo del flujo de integración.

5. Proceso y Relación entre Componentes

5.1. Flujo de Mensajes

El flujo de datos en el sistema es el siguiente:

1. La ruta `PublisherRoute` (productor) se activa cada 15 segundos.

2. Crea un mensaje de alerta y lo envía al exchange `alert-exchange` en RabbitMQ.
3. El exchange `alert-exchange`, al ser de tipo `fanout`, duplica el mensaje y lo envía a todas las colas vinculadas a él: `console.consumer.queue` y `email.consumer.queue`.
4. La ruta `Consumer1Route` (consumidor 1) recoge el mensaje de su cola (`console.consumer.queue`) y lo procesa (imprime en el log).
5. La ruta `Consumer2Route` (consumidor 2) recoge el mensaje de su cola (`email.consumer.queue`) y lo procesa (simula un envío de email).

5.2. Demostración del Desacoplamiento

La arquitectura implementada demuestra el desacoplamiento de las siguientes maneras:

- **Independencia del Productor:** El `PublisherRoute` no tiene conocimiento alguno de la existencia, número o estado de los consumidores. Su única responsabilidad es entregar el mensaje al exchange.
- **Independencia entre Consumidores:** `Consumer1Route` y `Consumer2Route` operan en total aislamiento. El procesamiento de uno no interfiere con el otro.

- **Resiliencia a Fallos:** La funcionalidad del `ConsumerController` lo demuestra de forma práctica. Si se realiza una llamada `GET` a `http://localhost:8080/pause/consumer1`, el `Consumer 1` dejará de procesar mensajes. Sin embargo:
  - El `Publisher` continuará publicando mensajes cada 15 segundos.
  - El `Consumer 2` continuará recibiendo y procesando cada mensaje sin interrupción.
  - RabbitMQ seguirá entregando mensajes a la cola `console.consumer.queue`, donde se acumularán (dado que la cola se declaró como durable).
  - Cuando se llame a `http://localhost:8080/resume/consumer1`, el consumidor se reactivará y comenzará a procesar todos los mensajes que se acumularon en su cola mientras estaba detenido.

Este comportamiento confirma un sistema resiliente y desacoplado, donde la indisponibilidad temporal de un componente no provoca un fallo en cascada del sistema completo.

6. Conclusiones

La implementación realizada ha permitido validar con éxito la eficacia del patrón de integración **Publicación/Suscripción** para construir sistemas distribuidos desacoplados. La combinación de Spring Boot, Apache Camel y RabbitMQ demostró ser un stack tecnológico robusto y eficiente para este propósito.

Se ha logrado el objetivo de crear un sistema donde un publicador puede emitir información que es consumida por múltiples suscriptores de forma independiente y asíncrona. La capacidad de controlar dinámicamente el ciclo de vida de los consumidores a través de una API REST sirvió como una demostración práctica y contundente de las ventajas del desacoplamiento, como la resiliencia y la mantenibilidad.

En conclusión, la mensajería asíncrona no es solo un mecanismo de comunicación, sino un pilar arquitectónico que habilita la construcción de aplicaciones modernas, escalables y resistentes a fallos, elementos indispensables en el panorama actual de la ingeniería de software.

7. Referencias

- Hohpe, G., & Woolf, B. (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional.
- Documentación oficial de Apache Camel. (s.f.). Recuperado de https://camel.apache.org/
- Documentación oficial de RabbitMQ. (s.f.). Recuperado de https://www.rabbitmq.com/

- Documentación oficial de Spring Boot. (s.f.). Recuperado de https://spring.io/projects/spring-boot