

Universidad de Las Américas  
Facultad de Ingenierías y Ciencias Agropecuarias  
*Ingeniería De Software*  
Progreso 3

**Nombres: Enrique Merizalde, Jossue Játiva, Doménica Escobar y Juan Aristizabal**

**Fecha: 25/06/2025**

## **TALLER PUB – SUB RABBITMQ**

### **1. Objetivo General**

Simular un sistema de notificaciones en el que:

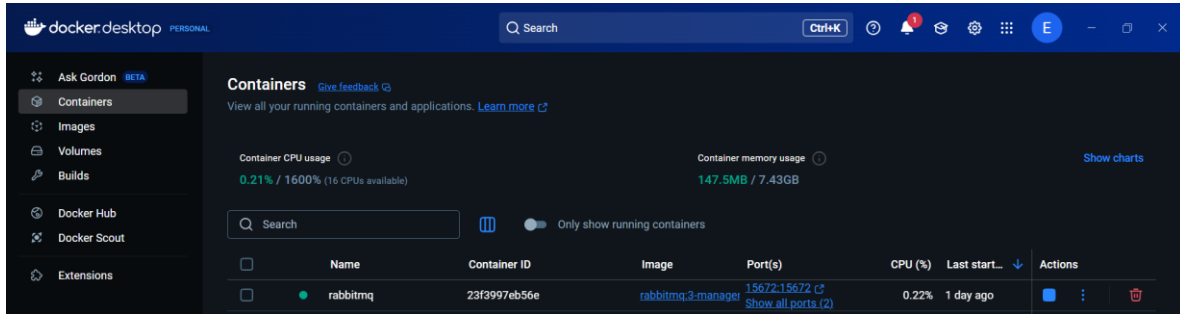
- Un publicador emite alertas cada 5 segundos.
- Dos suscriptores (consumidores) reciben el mismo mensaje, de forma desacoplada.

### **2. Requisitos previos**

Tener instalado:

- Java 11 o superior
- Apache Maven
- Visual Studio Code (u otro IDE)
- Docker Desktop

### **3. Tener Docker corriendo con Rabbit MQ**



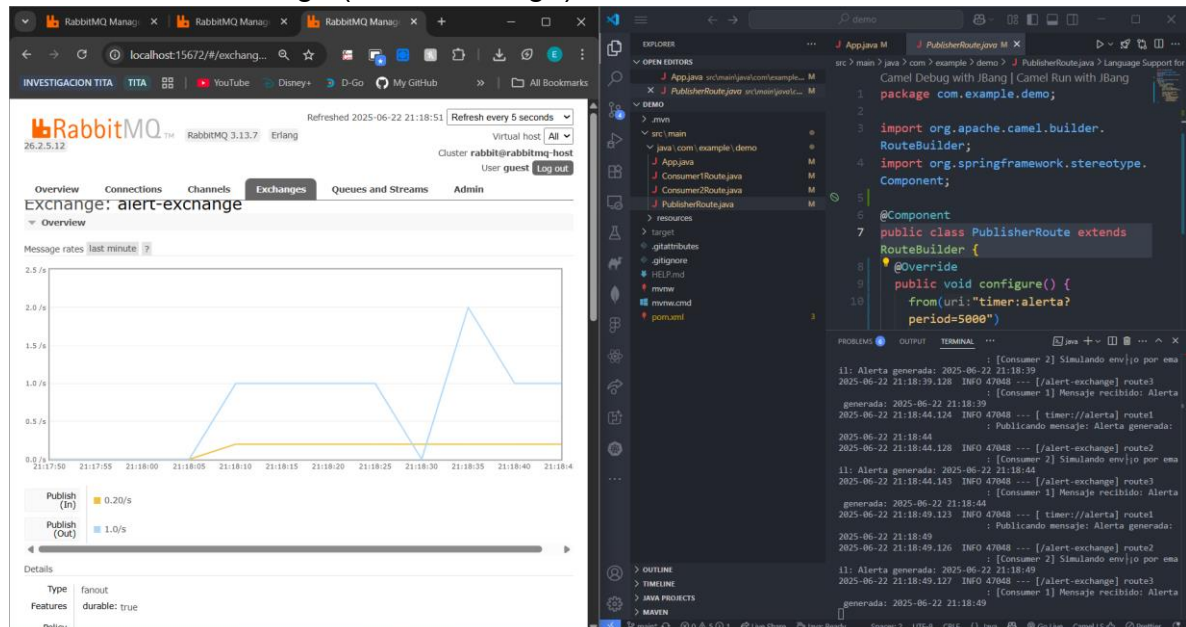
## Entregables del taller

### 1. Link a repositorio:

[https://github.com/CHACHO617/PubSub\\_Integracion](https://github.com/CHACHO617/PubSub_Integracion)

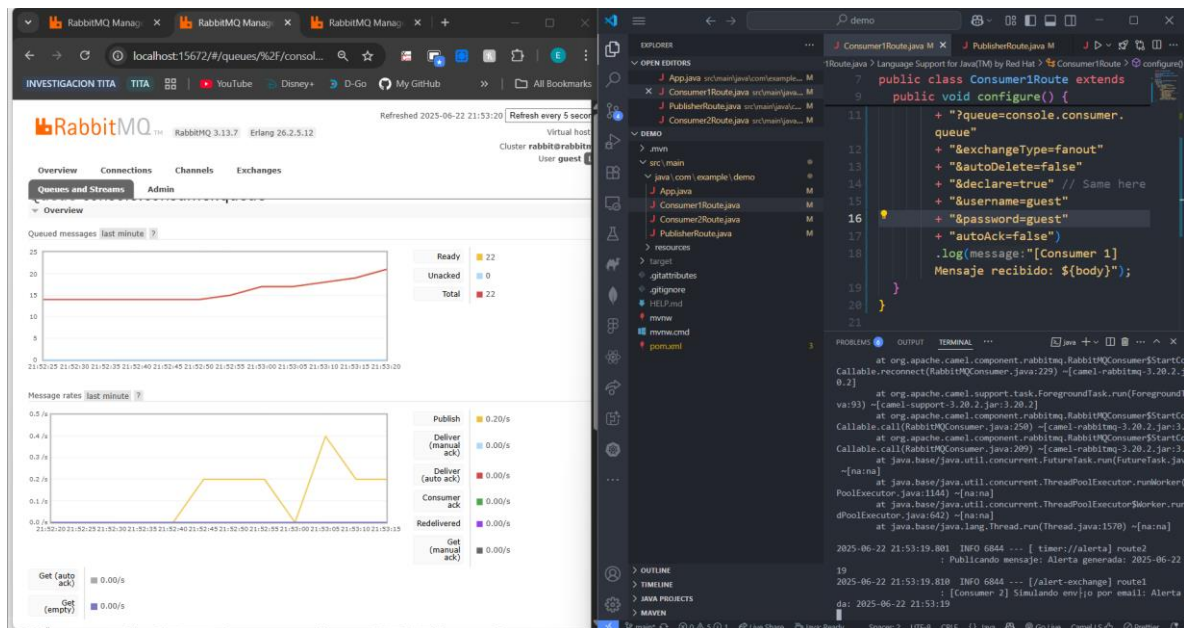
### 2. Capturas de pantalla RabbitMQ e implementación del Publicar y Suscriptores:

#### RabbitMQ – Exchange (alert-exchange)

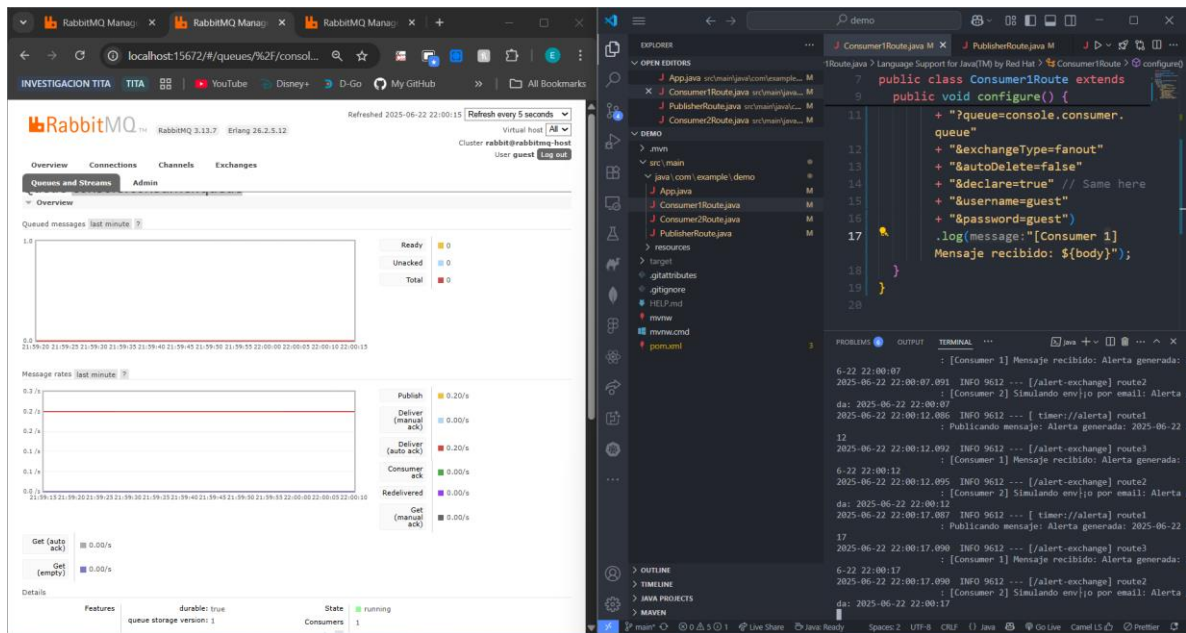


RabbitMQ – Queue (console.consumer.queue)

Se agregó “autoAck=false” para forzar que se acumulen mensajes.

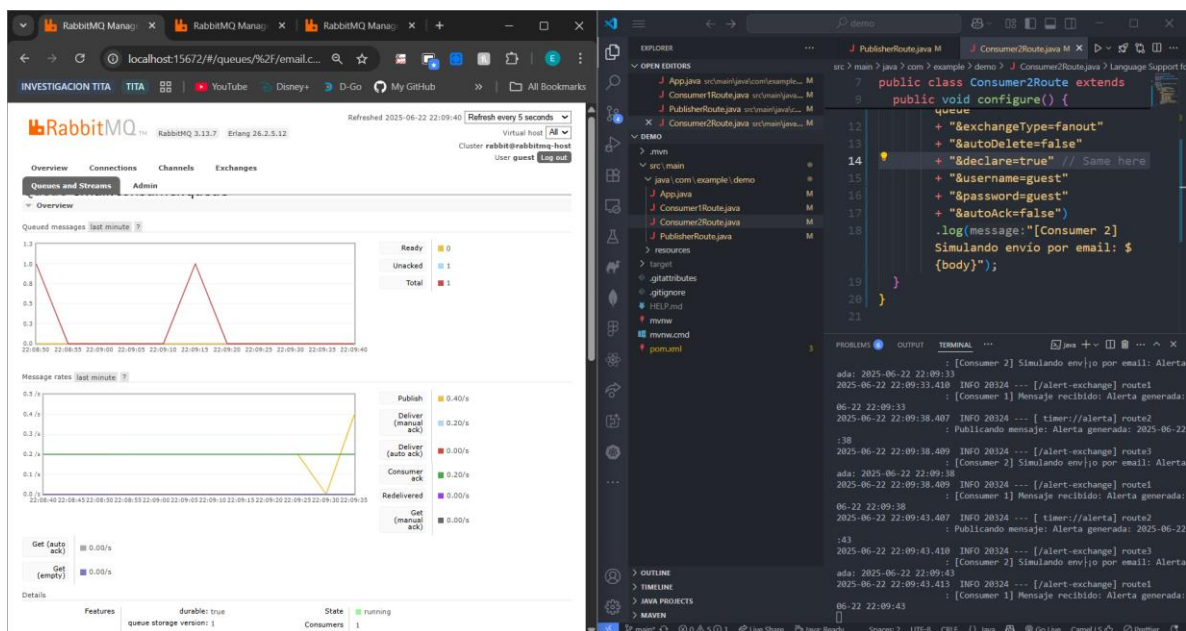


autoAck=true Evita que se acumulen los mensajes.

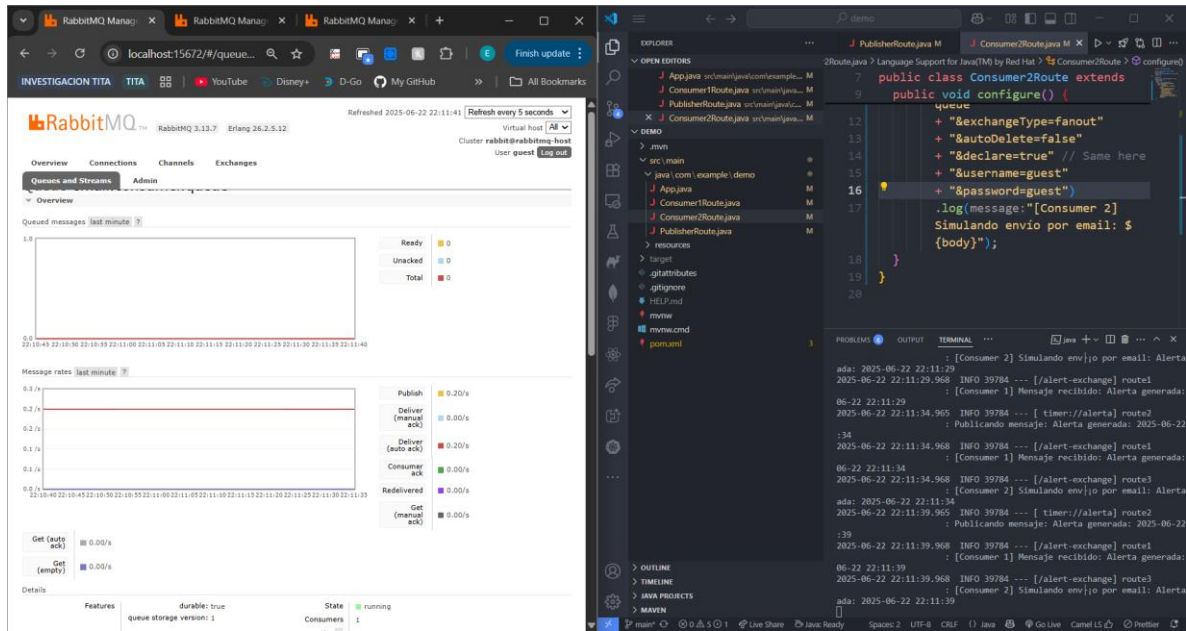


RabbitMQ – Queue (email.consumer.queue)

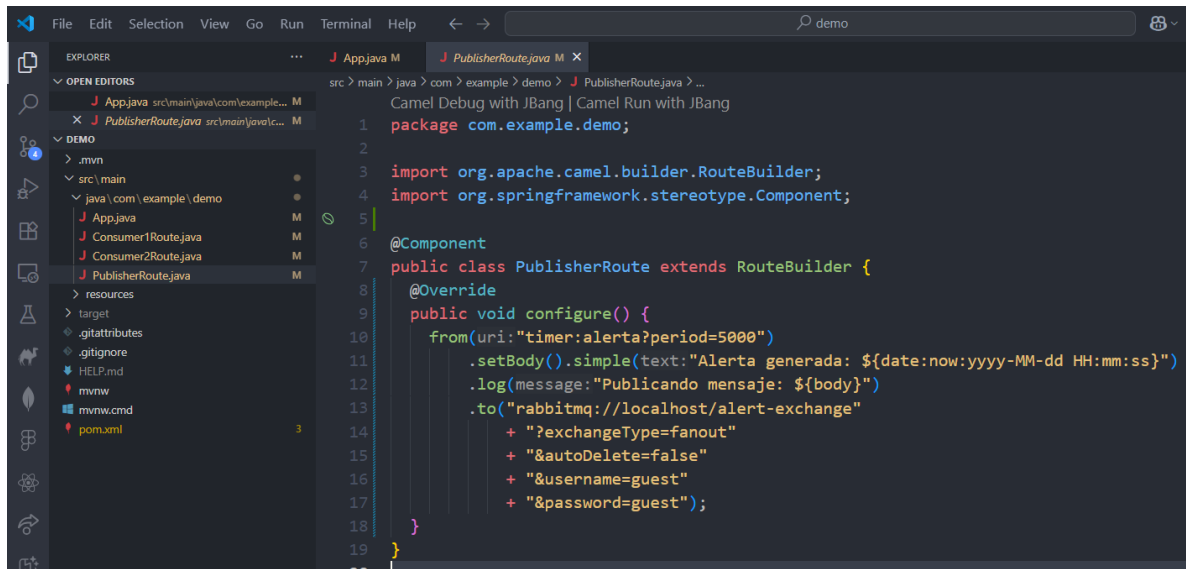
Se agregó "autoAck=false" para forzar que se acumulen mensajes.



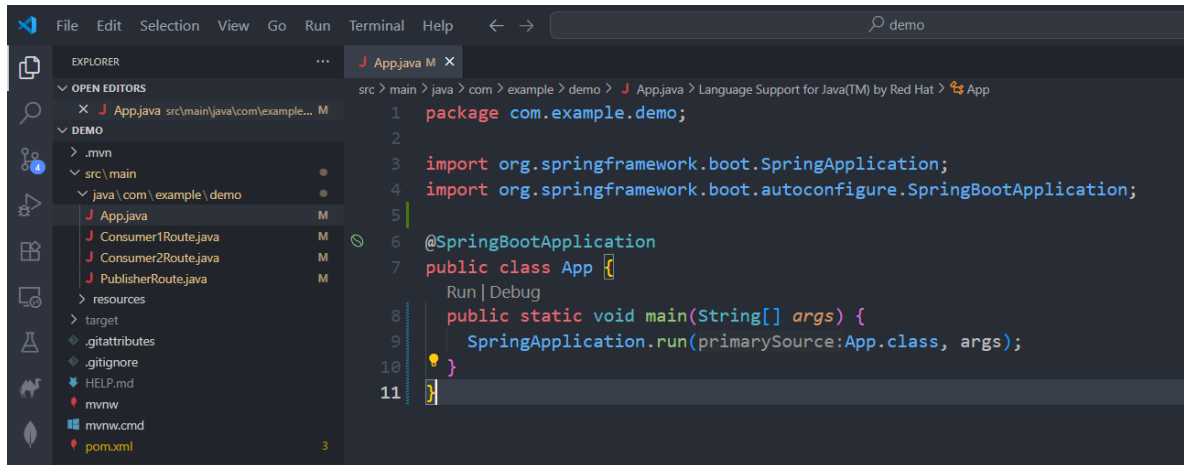
Sin autoAck=false Evita que se acumulen los mensajes.



## PublisherRoute.java



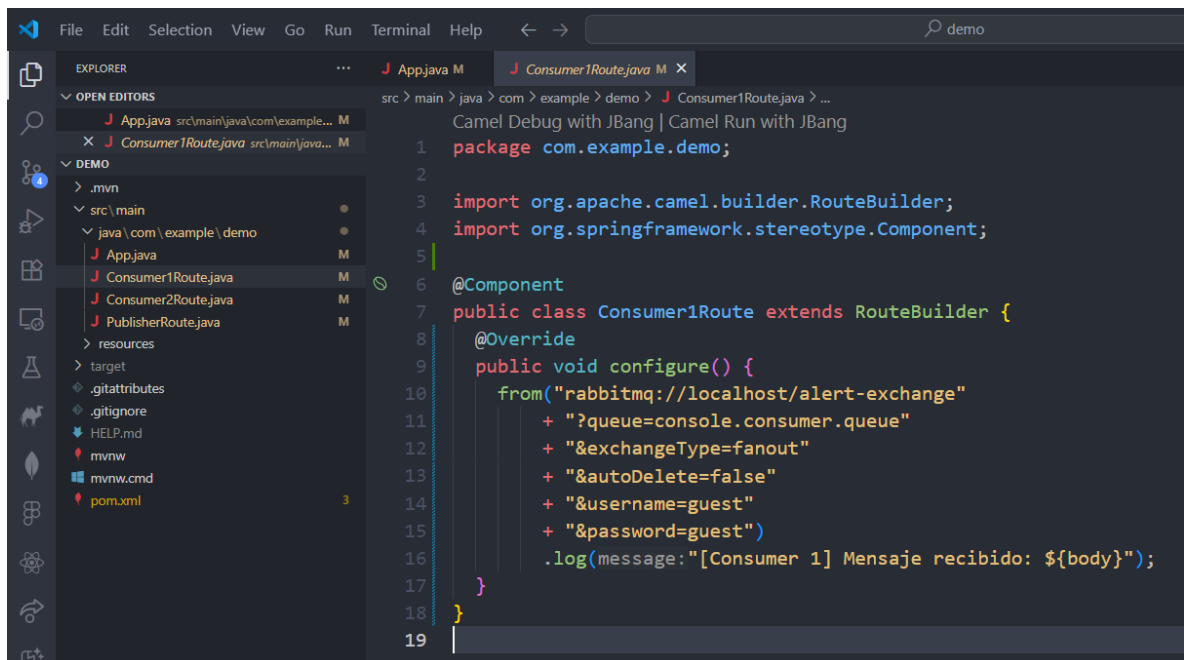
## App.java



```
File Edit Selection View Go Run Terminal Help demo
EXPLORER
  OPEN EDITORS
    App.java src/main/java/com/example... M
  DEMO
    .mvn
    src/main
      java/com/example/demo
        App.java M
        Consumer1Route.java M
        Consumer2Route.java M
        PublisherRoute.java M
    resources
    target
    .gitattributes
    .gitignore
    HELP.md
    mvnw
    mvnw.cmd
    pom.xml 3

src > main > java > com > example > demo > J App.java > Language Support for Java(TM) by Red Hat > App
1 package com.example.demo;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5
6 @SpringBootApplication
7 public class App {
8     Run | Debug
9     public static void main(String[] args) {
10         SpringApplication.run(primarySource:App.class, args);
11     }
12 }
```

## Consumer1Route.java

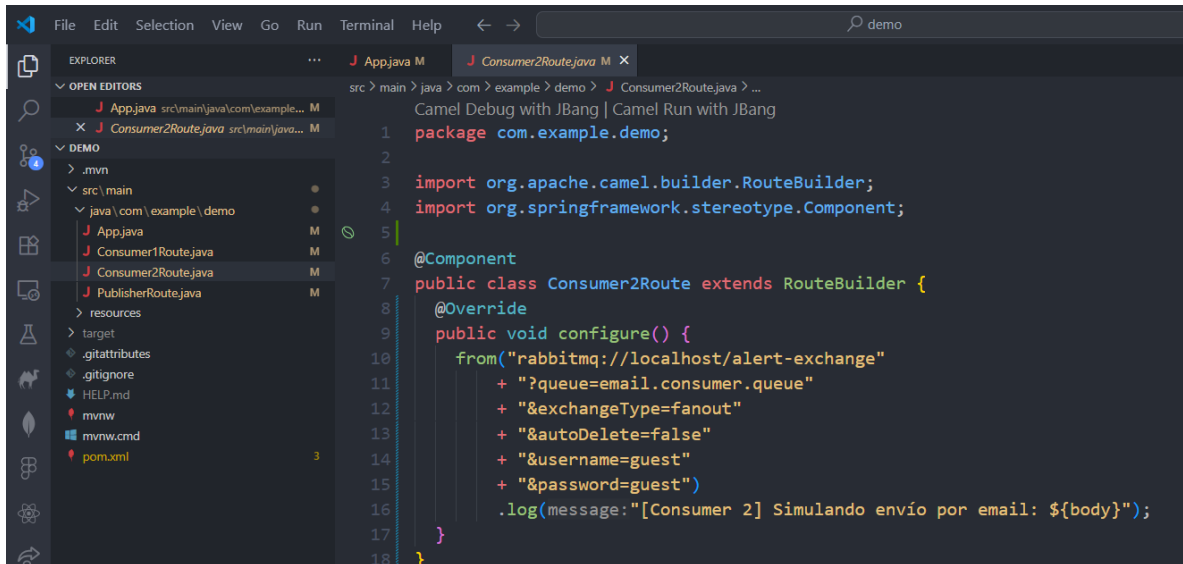


```
File Edit Selection View Go Run Terminal Help demo
EXPLORER
  OPEN EDITORS
    App.java src/main/java/com/example... M
    Consumer1Route.java src/main/java... M
  DEMO
    .mvn
    src/main
      java/com/example/demo
        App.java M
        Consumer1Route.java M
        Consumer2Route.java M
        PublisherRoute.java M
    resources
    target
    .gitattributes
    .gitignore
    HELP.md
    mvnw
    mvnw.cmd
    pom.xml 3

src > main > java > com > example > demo > J Consumer1Route.java > ...
Camel Debug with JBang | Camel Run with JBang
1 package com.example.demo;
2
3 import org.apache.camel.builder.RouteBuilder;
4 import org.springframework.stereotype.Component;
5
6 @Component
7 public class Consumer1Route extends RouteBuilder {
8     @Override
9     public void configure() {
10         from("rabbitmq://localhost/alert-exchange"
11             + "?queue=console.consumer.queue"
12             + "&exchangeType=fanout"
13             + "&autoDelete=false"
14             + "&username=guest"
15             + "&password=guest")
16             .log(message:"[Consumer 1] Mensaje recibido: ${body}");
17     }
18 }
19 }
```

## Consumer2Route.java





```
1 package com.example.demo;
2
3 import org.apache.camel.builder.RouteBuilder;
4 import org.springframework.stereotype.Component;
5
6 @Component
7 public class Consumer2Route extends RouteBuilder {
8     @Override
9     public void configure() {
10         from("rabbitmq://localhost/alert-exchange"
11             + "?queue=email.consumer.queue"
12             + "&exchangeType=fanout"
13             + "&autoDelete=false"
14             + "&username=guest"
15             + "&password=guest")
16             .log(message: "[Consumer 2] Simulando envío por email: ${body}");
17     }
18 }
```

### 3. ¿Qué patrón de integración se aplicó?

El patrón de integración aplicado fue el Patrón de Publicador/Suscriptor (Pub/Sub), mediante el uso de Apache Camel como framework de orquestación e integración, y RabbitMQ como broker de mensajería.

Este patrón permite que un productor (PublisherRoute) publique mensajes a un exchange fanout sin preocuparse por los destinatarios específicos. Luego, múltiples consumidores (Consumer1Route y Consumer2Route) reciben simultáneamente una copia de cada mensaje al estar suscritos al mismo exchange.

Este enfoque sigue el principio de desacoplamiento y escalabilidad, ideal para sistemas distribuidos que requieren procesamiento paralelo o notificación a múltiples servicios.

### 4. Cómo se logró el desacoplamiento productor-consumidor

El desacoplamiento se logró a través de RabbitMQ y su mecanismo de exchange tipo fanout, en conjunto con las rutas declarativas de Apache Camel.

- El productor (PublisherRoute) publica mensajes a alert-exchange sin conocer a quién van dirigidos.

- Los consumidores (Consumer1 y Consumer2) están vinculados a colas diferentes (console.consumer.queue, email.consumer.queue), que a su vez están unidas al exchange.
- Este diseño permite que los componentes se desarrollen, desplieguen y escalen de forma independiente, y que se agreguen o eliminen consumidores sin modificar la lógica del publicador.

Además, Apache Camel abstraer la lógica de conexión, permitiendo que cada parte solo conozca su fuente o destino, pero nunca el flujo completo del sistema.

## **5. Ventajas que se observaron durante la práctica.**

Durante la práctica se identificaron varias ventajas:

- Desacoplamiento total: productores y consumidores no dependen unos de otros ni están directamente conectados.
- Escalabilidad: es posible añadir tantos consumidores como se quiera, para procesar el mismo flujo sin afectar al publicador.
- Observabilidad: gracias a los logs automáticos de Camel y la consola web de RabbitMQ, es fácil ver el flujo de mensajes en tiempo real.
- Flexibilidad: los mensajes pueden procesarse de distintas formas en cada consumer (por consola, simulando un envío por email, etc.)
- Simplicidad declarativa: usar Camel con anotaciones como `@Component` permite definir rutas de integración de manera sencilla y muy legible.
- Pruebas y simulaciones claras: fue fácil evidenciar el funcionamiento al visualizar en consola la publicación y el consumo del mismo mensaje por múltiples rutas.