Universidad de las Américas

Integración de Sistemas

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

Martín Vargas y Kevin Rosero

Objetivo

Simular un sistema de notificaciones en el que:

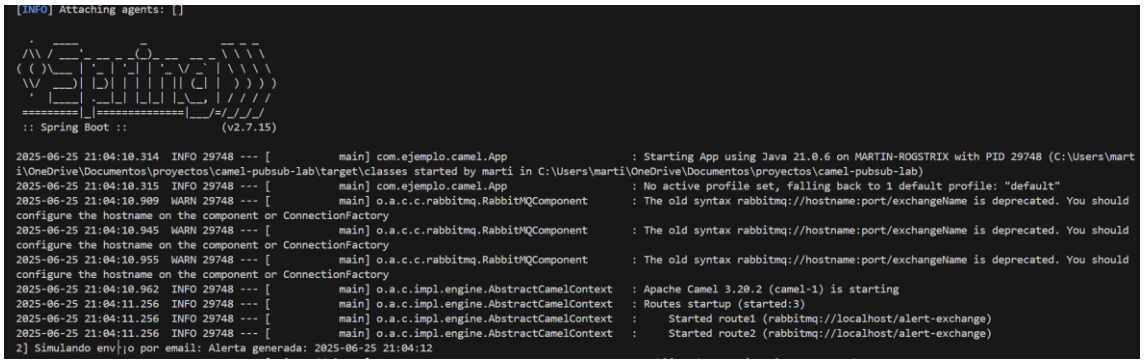
• Un publicador emite alertas cada 5 segundos.

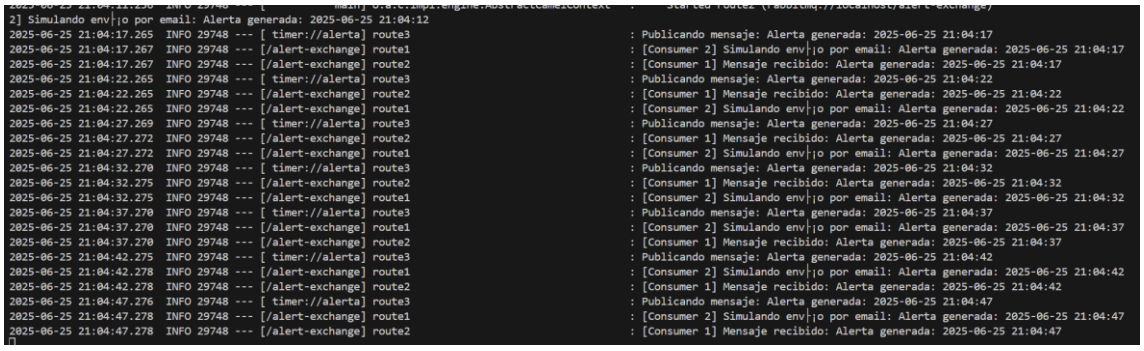
• Dos suscriptores (consumidores) reciben el mismo mensaje, de forma desacoplada.

Repositorio Github: https://github.com/MartinVargas07/Taller-Patron-Pub-Sub-Vargas-Rosero.git

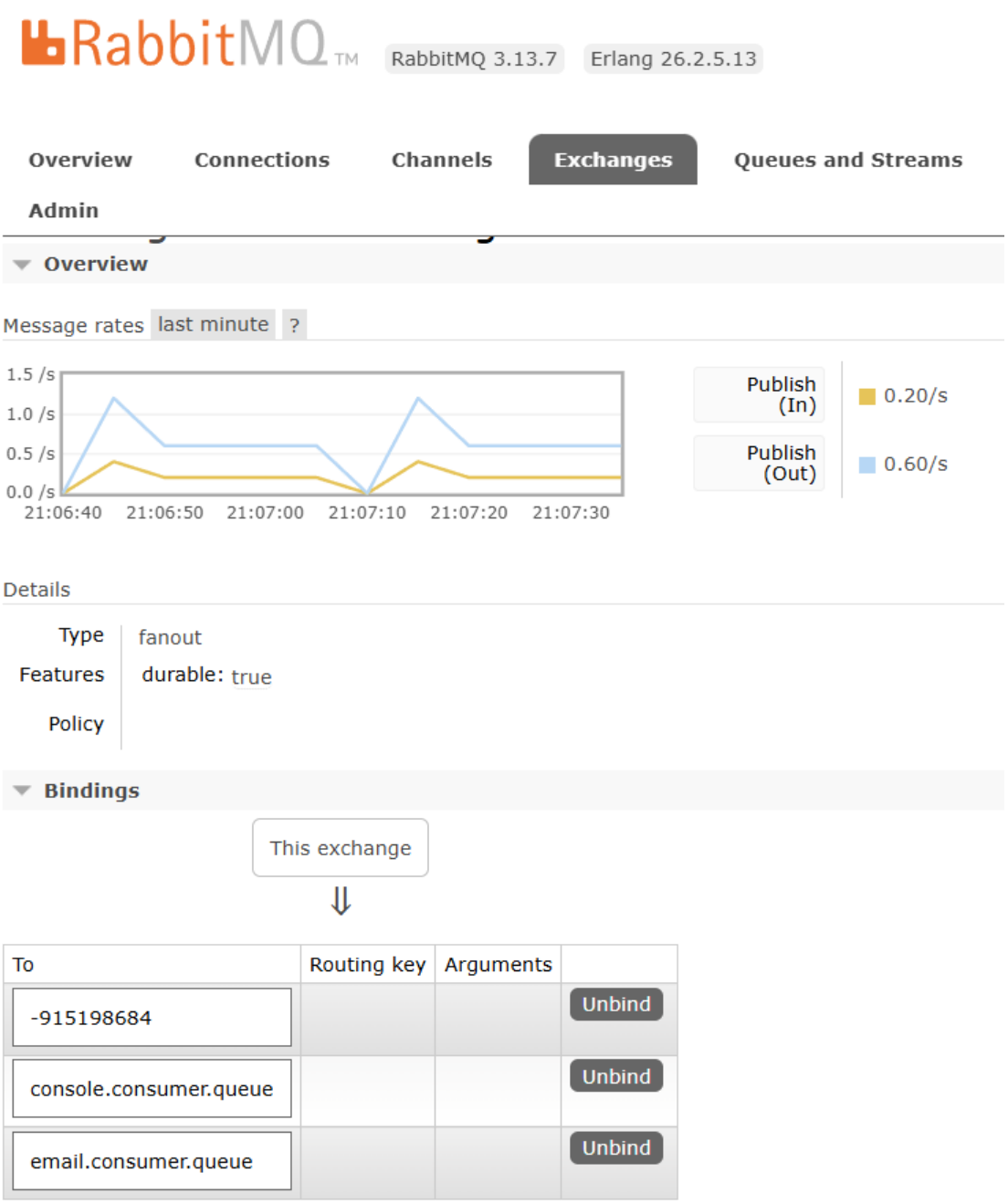
Capturas de Imágenes:

Mensajes recurrentes del Publicador y los Suscriptores 1 y 2

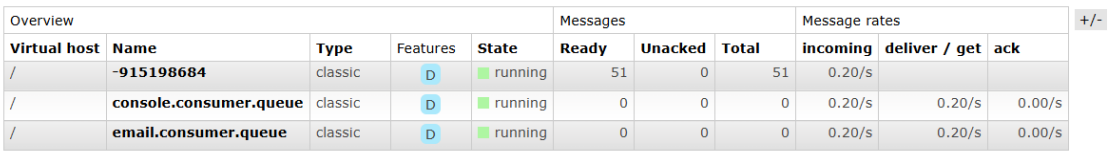




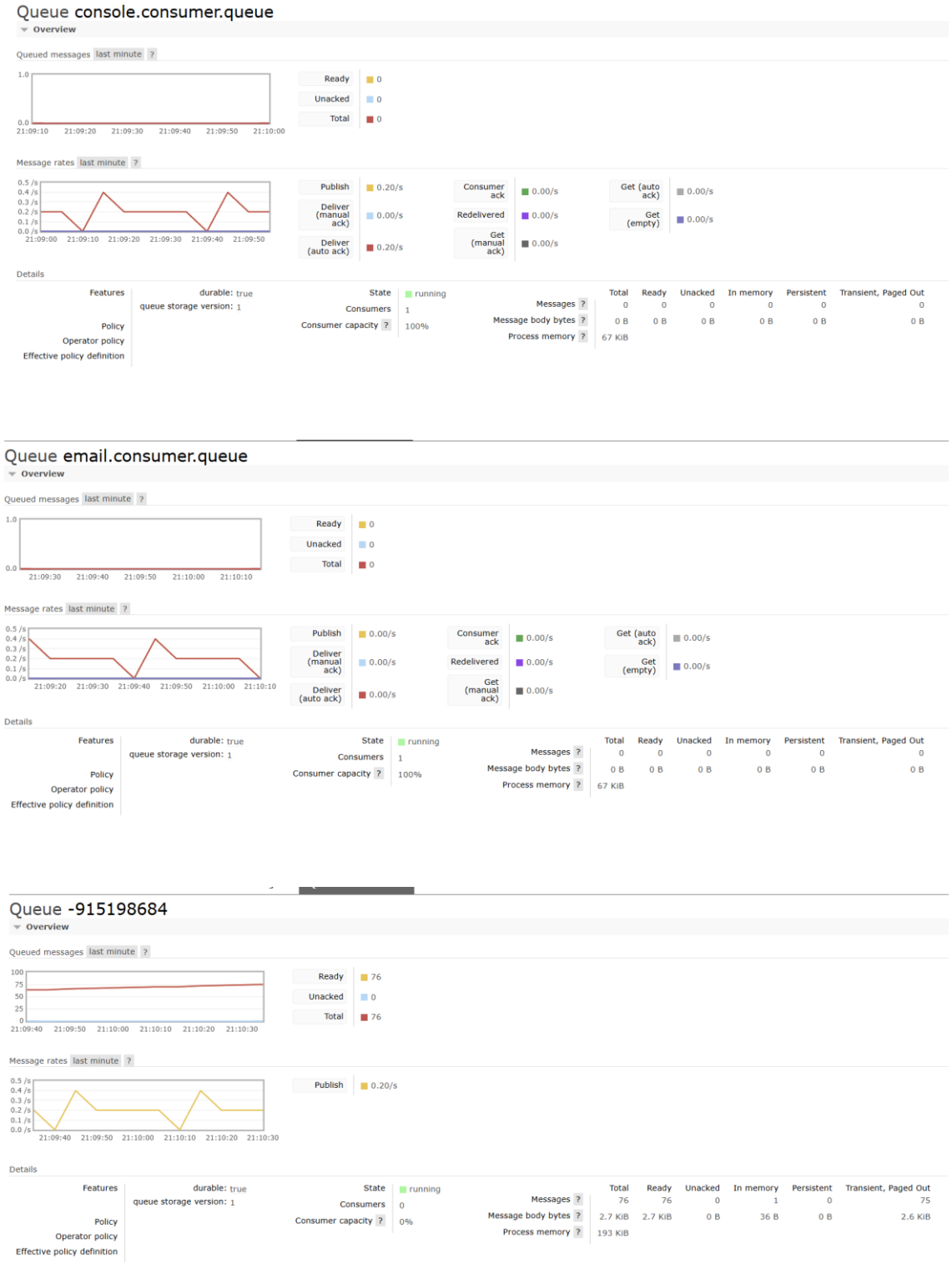
alert-exchange en RabbitMQ



Las dos colas: console.consumer.queue y email.consumer.queue.



Overview					Messages			Message rates			+/-
Virtual host	**Name**	**Type**	Features	**State**	**Ready**	**Unacked**	**Total**	**incoming**	**deliver / get**	**ack**	
/	**-915198684**	classic	D	running	51	0	51	0.20/s			
/	**console.consumer.queue**	classic	D	running	0	0	0	0.20/s	0.20/s	0.00/s	
/	**email.consumer.queue**	classic	D	running	0	0	0	0.20/s	0.20/s	0.00/s	



Esa cola con un nombre generado automáticamente apareció porque, durante una ejecución anterior con errores, uno de los consumidores se conectó al exchange sin especificar un nombre de cola (?queue=...). En esos casos, RabbitMQ crea una cola temporal con un nombre único para ese consumidor. Ahora que el código está corregido, esa cola quedó huérfana y sin un consumidor activo, por lo que simplemente acumula los mensajes que recibe del publicador y puede ser eliminada de forma segura.

Preguntas a las respuestas:

i. ¿Qué patrón de integración se aplicó?

En esta práctica, implementamos el patrón de integración Publicar-Suscribir (Publish-Subscribe o Pub/Sub). La idea de este patrón es bastante directa: teníamos un componente, al que llamamos "publicador", que se encargaba de generar y enviar mensajes de alerta. Sin embargo, este publicador no enviaba las alertas directamente a quienes debían recibirlas. En lugar de eso, las publicaba en un canal central, que en nuestro caso fue un "exchange" en RabbitMQ. Por otro lado, teníamos dos "suscriptores" que estaban "escuchando" ese canal. Cada vez que el publicador enviaba una alerta, el canal se encargaba de distribuir una copia de ese mismo mensaje a cada uno de nuestros suscriptores.

ii. ¿Cómo se logró el desacoplamiento productor-consumidor?

El desacoplamiento fue la clave de toda la práctica y lo logramos gracias a RabbitMQ, que actuó como un intermediario. El publicador (PublisherRoute) y los dos consumidores (Consumer1Route y Consumer2Route) nunca se comunicaron directamente entre sí.

El publicador solo conocía la dirección del "exchange" de RabbitMQ. Su única tarea era enviar los mensajes de alerta a ese punto central, sin preocuparse por quién o cuántos los recibirían.

Los consumidores, por su parte, solo conocían la dirección de ese mismo "exchange". Cada uno se conectaba de forma independiente para recibir los mensajes. No sabían nada sobre el publicador, ni siquiera sabían de la existencia del otro consumidor.

Esta separación, donde RabbitMQ actúa como un "cartero" que gestiona la distribución de los mensajes, es lo que garantiza que los componentes estén desacoplados. Pueden funcionar, modificarse o incluso fallar de manera independiente sin afectar a los demás.

iii. Ventajas que observamos durante la práctica

Al desarrollar el taller, notamos varias ventajas prácticas de usar este patrón:

Flexibilidad para crecer: Fue evidente lo fácil que sería escalar el sistema. Si mañana necesitamos un tercer consumidor que guarde las alertas en una base de datos o envíe una notificación a un celular, solo tendríamos que crearlo y suscribirlo al mismo canal. No necesitaríamos tocar ni una línea de código del publicador original.

Tareas diferentes con la misma información: Vimos que, aunque el mensaje era el mismo para ambos suscriptores, cada uno podía hacer algo completamente diferente con él. Uno simplemente lo mostraba en la consola, mientras que el otro simulaba enviarlo por correo. Esto es muy útil en sistemas reales, donde un mismo evento puede disparar diferentes procesos de negocio.

Agilidad en la comunicación: El publicador enviaba la alerta y se olvidaba de ella, quedando libre para hacer otras cosas (en nuestro caso, esperar 5 segundos para generar

la siguiente). No tenía que esperar a que los consumidores confirmaran la recepción, lo que hace que la comunicación sea mucho más rápida y eficiente.