Kafka: del problema original a tu primer evento

5 de noviembre de 2026 kafka java spring-boot arquitectura event-driven

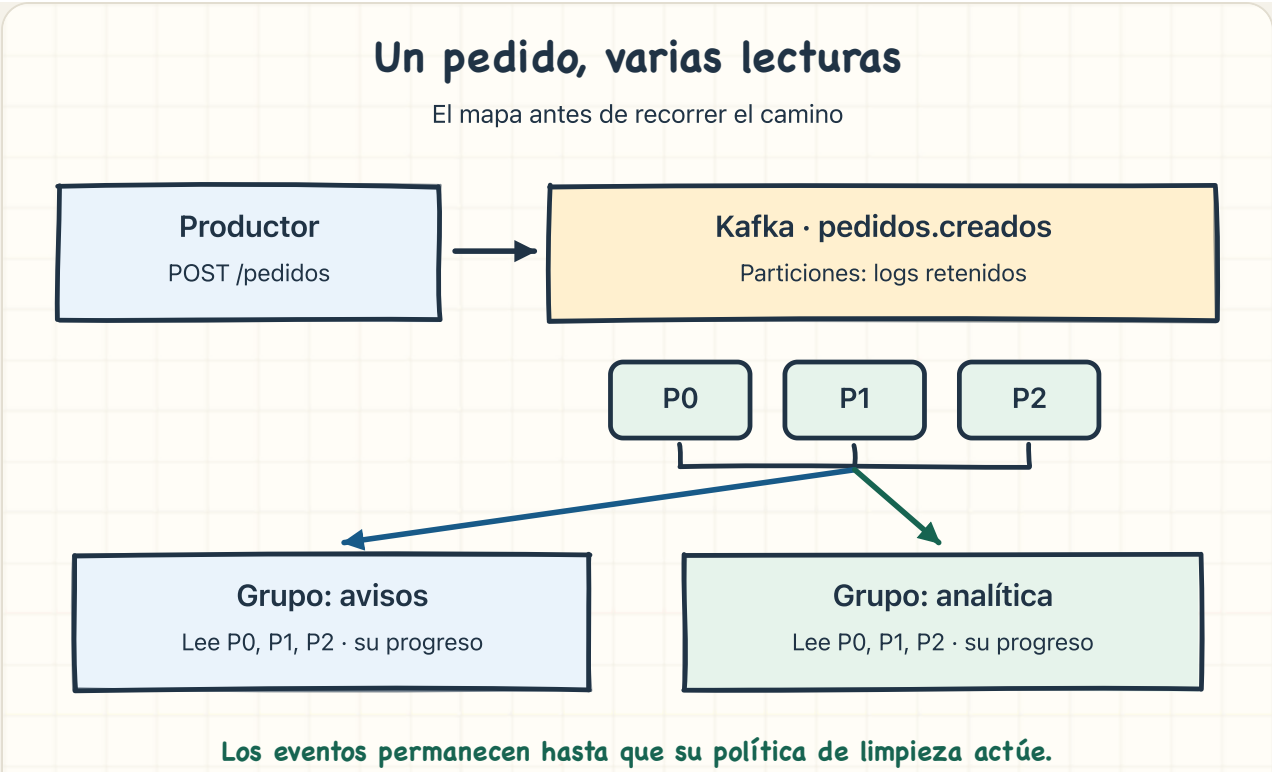
Acabas de terminar la compra de una tienda. El pedido entra, pero el correo tarda ocho segundos y el tablero de ventas está caído. Quieres responder al cliente sin depender de ambos, y mañana necesitarás reconstruir las ventas de hoy. Vamos a resolver ese problema juntos.

Apache Kafka es una plataforma distribuida para publicar, conservar y leer flujos de eventos. Un productor registra lo que ocurrió; Kafka guarda esos registros en secuencias llamadas particiones; los consumidores los leen y recuerdan su progreso. Cada grupo puede avanzar a su ritmo y volver sobre la historia que permanezca disponible. Tu aplicación decide qué hacer con cada evento: enviar un aviso, actualizar una vista o calcular una métrica.



Aprenderemos con un pedido: publicarlo, encontrarlo y volver a leerlo.

Quiero que al terminar puedas explicar por qué elegirías Kafka, seguir el viaje de un registro y operar un ejemplo pequeño. Primero dibujaremos la necesidad; enseguida publicarás tu primer pedido con Java y Spring Boot. Después abriremos las piezas que hicieron posible ese recorrido.



Kafka conserva el flujo; los grupos de consumidores interpretan los hechos para propósitos diferentes.

Tu punto de partida	Recorrido recomendado dentro de esta guía
Empiezas con Kafka	Lee el problema y el origen, ejecuta el primer evento y sigue con fundamentos, piezas y experimentos.
Ya integras aplicaciones	Prueba el primer evento, revisa los criterios y contrasta grupos, replay y fallos.
Diseñas u operas sistemas	Ve a internos, abre las notas avanzadas y evalúa ecosistema, contratos y operación.

La práctica usa **Java 21, Spring Boot 4.1.2 y Kafka 4.4.0 KRaft**. El proyecto completo contiene las fuentes y sus pruebas. Los fragmentos de ambos idiomas proceden del mismo código verificado. Conservaré los nombres **`PedidoCreado`** y **`pedidoId`** para que encuentres cada pieza en el repositorio.

▼ EN ESTE ARTÍCULO

01. 1. El problema antes de Kafka
02. 2. Cómo nació y por qué tenía sentido entonces
03. 3. Publica tu primer evento
04. 4. Cuándo merece la pena y cuándo complica de más

05. 5. Los fundamentos que necesitas, sin exigir una arquitectura
06. 6. Las piezas: abre la tarjeta del pedido
07. 7. El viaje por dentro: de enviar a confirmar
08. 8. El laboratorio explicado y sus experimentos
09. 9. Lo que pasa cuando algo falla
10. 10. Del laboratorio a una decisión real

1. El problema antes de Kafka

El pedido que conoce demasiadas cosas

Imagina un método **`crearPedido`** que guarda el pedido, llama al servicio de correo y actualiza el tablero de ventas. Las tres acciones parecen parte de la misma compra porque ocurren cerca en el tiempo. Sin embargo, tienen responsabilidades diferentes. Guardar el pedido pertenece al proceso de compra; enviar un correo comunica un resultado; calcular ventas construye una vista para otro lector.

Cuando las encadenas, el tiempo de respuesta incluye las esperas de todas. También unes sus fallos: si analítica responde con error, debes decidir qué le dices a un cliente cuyo pedido ya quedó guardado. Repetir el método completo podría registrar dos pedidos o enviar dos correos. No repetirlo podría dejar el tablero incompleto. La dificultad aparece incluso con poco tráfico.

Una cola durable puede ser una excelente respuesta: deja una tarea y permite que un trabajador la complete después. No necesitas Kafka para desacoplar el envío de un correo. Tampoco necesitas microservicios para usar mensajería: una aplicación puede publicar y consumir desde distintos módulos del mismo despliegue. Antes de elegir tecnología, conviene escribir qué necesitas proteger y durante cuánto tiempo.

Ahora aparece un segundo lector

Supón que notificaciones recibe el aviso del pedido y manda el correo. Más tarde, el equipo de analítica quiere procesar ese mismo hecho para calcular ventas por hora. No está compitiendo por enviar el correo: necesita una interpretación diferente de la misma compra. Si ambos se conectan a una cola de trabajadores y cada mensaje llega a uno de ellos, estarás repartiendo trabajos que los dos necesitaban observar.

Puedes publicar copias a dos colas. Es un patrón válido de pub/sub. La pregunta siguiente es qué ocurre cuando aparece un tercer lector dentro de un mes y necesita revisar las compras anteriores. ¿Hay una copia histórica? ¿Por cuánto tiempo? ¿Puede recorrerla sin afectar al lector que ya está trabajando? La arquitectura necesita responderlo expresamente; el nombre del broker no responde por ella.

Aquí empieza a resultar atractiva otra abstracción: un **registro retenido de hechos**, con una posición de lectura independiente para cada grupo. El productor escribe una vez en ese registro. Los lectores avanzan por él según sus necesidades. Leer no equivale a retirar la página del cuaderno. Esa separación entre conservar datos y recordar el progreso es la idea central que quiero que mantengas mientras aparecen nombres nuevos.

Tres necesidades que debes separar

La primera es **comunicar**: avisar a otro componente de que ocurrió algo. La segunda es **amortiguar**: permitir que el productor avance aunque el lector esté temporalmente ocupado. La tercera es **conservar una historia utilizable**: permitir que un lector nuevo o corregido vuelva sobre eventos todavía disponibles. Una solución puede cubrir una o varias; compara las necesidades reales, no una lista de características fuera de contexto.

> **Lo que debes llevarte:** empieza por identificar quién necesita el hecho, qué puede esperar y qué historia necesita recuperar. Kafka es una posible implementación de esa conversación.

---

2. Cómo nació y por qué tenía sentido entonces

Dos caminos que necesitaban encontrarse

Kafka se construyó en LinkedIn para mover actividad de usuarios y registros operativos hacia lectores con necesidades diferentes. Un sistema quería reaccionar continuamente; otro quería cargar datos para análisis por lotes. Mantener una integración distinta para cada destino multiplicaba conexiones, formatos y procedimientos de recuperación.

Jay Kreps, Neha Narkhede y Jun Rao describieron el diseño en *Kafka: a Distributed Messaging System for Log Processing*, presentado en NetDB 2011. El artículo original contrasta la mensajería empresarial y los agregadores de logs de aquella época. Quiero que ubiques el problema antes que la marca: había que mover un flujo grande y hacerlo utilizable tanto por aplicaciones continuas como por sistemas offline.

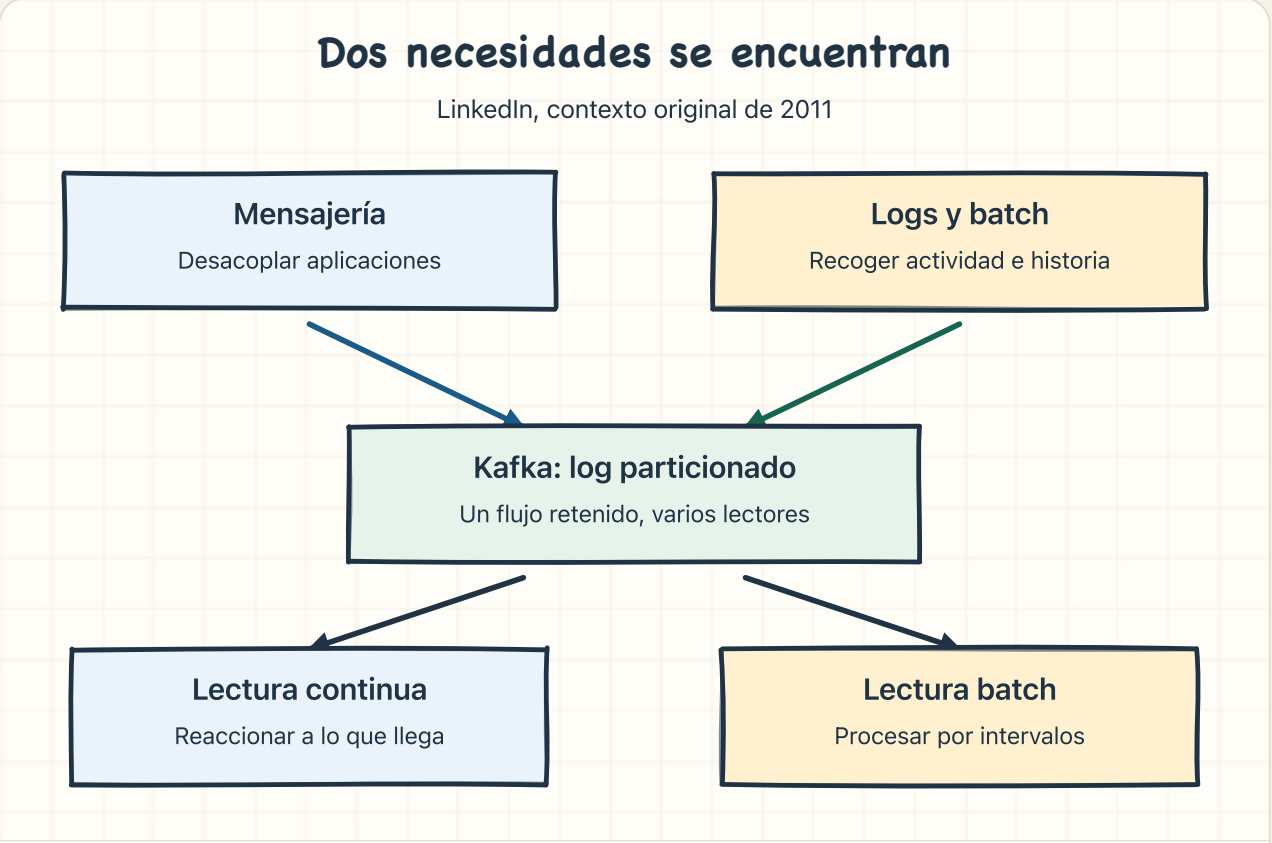
Antecedente o alternativa de la época	Qué aportaba	Qué necesidad empujaba otro diseño
Mensajería empresarial; APIs como JMS y brokers como ActiveMQ o RabbitMQ	Comunicación entre aplicaciones, entrega y enrutamiento	Buscar una combinación apropiada de capacidad, historial compartido y lectores independientes
Agregadores de logs como Scribe y Flume	Reunir actividad y llevarla a destinos de datos	Servir también lectores continuos con sus propias posiciones
Procesamiento batch con Hadoop	Analizar grandes conjuntos por lotes	Reaccionar a la misma actividad mientras sigue llegando

Antecedente o alternativa de la época	Qué aportaba	Qué necesidad empujaba otro diseño
Logs internos de bases de datos	Ordenar cambios para recuperación y replicación	Aplicar la secuencia de cambios a la integración entre sistemas

JMS es una API y Hadoop un entorno de procesamiento: no son productos equivalentes entre sí. La tabla compara trabajos que había que conectar. Tampoco describe todas las capacidades de sus versiones actuales. Por ejemplo, RabbitMQ Streams permite hoy lecturas no destructivas sobre un log retenido.

La combinación que eligió Kafka

La idea fue conservar registros en **logs particionados** y permitir que distintos lectores los recorrieran por posición. Particionar distribuye trabajo; agrupar registros en lotes amortiza costes de red y escritura. Separar almacenamiento y progreso hace posible incorporar un lector sin pedir al productor una llamada adicional por cada destino.



La necesidad original: aprovechar una misma actividad en sistemas con ritmos diferentes.

Referencia	Qué ayuda a entender
NetDB, junio de 2011: artículo de Kreps, Narkhede y Rao	El problema original y sus decisiones de diseño

Referencia	Qué ayuda a entender
LinkedIn, julio de 2011: presentación pública de Narkhede	Su uso para actividad, logs y consumo online/offline
Kreps, diciembre de 2013: *The Log*	El log como abstracción compartida para integración, replicación y procesamiento
Kafka 4: KRaft en lugar de ZooKeeper	La coordinación de metadatos que encontraremos en la práctica actual

La presentación de LinkedIn y The Log son referencias para seguir ese recorrido. KRaft se explica en la documentación actual.

Esa historia sigue siendo útil cuando una compra alimenta avisos, métricas y nuevas vistas del negocio. Tu escala puede ser pequeña; lo que debes evaluar es si necesitas sus propiedades y puedes sostener su operación. Vamos a observar primero el mecanismo y después a decidir cuánto sentido tiene para tu aplicación.

---

3. Publica tu primer evento

Para empezar necesitas Docker con Compose, Java 21 y tres terminales. Maven viene incluido mediante su Wrapper. En Windows utiliza **`mvnw.cmd`** en lugar de **`./mvnw`**; los comandos de shell que ves aquí corresponden a macOS/Linux o una shell compatible.

En la primera terminal, descarga el proyecto, entra al laboratorio y prepara el ejecutable:

```
git clone https://github.com/workmanuelherreramesen-lab/manuelherreramesen-profile-site.git
cd manuelherreramesen-profile-site/examples/spring/kafka
java -version
docker compose version
docker compose up -d --wait
./mvnw -DskipTests package
java -jar target/kafka-pedidos-1.0.0.jar --spring.profiles.active=productor
```

Espera al mensaje **`Started PedidosApplication`**. El productor escucha en **`localhost:8091`** y Kafka en **`localhost:9092`**; ambos puertos deben estar libres. Mantén la misma carpeta de trabajo en las otras terminales. El primer arranque puede tardar mientras se descargan imágenes y dependencias. Si Docker falla, revisa **`docker compose logs kafka`** antes de continuar.

En la segunda terminal arranca el lector. Su perfil desactiva los componentes del productor y el argumento final evita abrir un segundo servidor HTTP:

```
java -jar target/kafka-pedidos-1.0.0.jar \
  --spring.profiles.active=consumidor \
  --spring.main.web-application-type=none
```

Cuando aparezca la asignación de particiones, publica desde la tercera terminal:

```
curl -i -X POST http://localhost:8091/pedidos \
  -H 'Content-Type: application/json' \
  -d '{"pedidoId":"pedido-4821"}'
```

La respuesta HTTP 202 incluye la identidad del evento, su partición y su offset. El lector imprime esas mismas coordenadas. En el laboratorio explicado compararemos la salida completa de una ejecución y estudiaremos cada clase. Por ahora identifica tres pasos: la petición crea el evento, el broker reconoce su publicación y el consumidor lo interpreta.

Prueba algo pequeño: detén el consumidor con **`Ctrl+C`**, publica **`pedido-4822`** y vuelve a iniciar el mismo lector. ¿Qué esperas que ocurra con ese segundo pedido? Observa su terminal antes de abrir la explicación.

> **– Solución**
>
> el lector encuentra el pedido pendiente porque Kafka conserva el registro y el grupo puede retomar su progreso. La recuperación depende de que esos datos sigan retenidos; más adelante pondremos una ventana concreta y veremos cómo inspeccionarla.

Ya tienes un flujo funcionando. Puedes dejar ambos procesos abiertos para los experimentos o volver aquí cuando llegues a ellos. Si repites el arranque del productor mientras sigue vivo, encontrarás el puerto ocupado; conserva una sola instancia del rol HTTP.

---

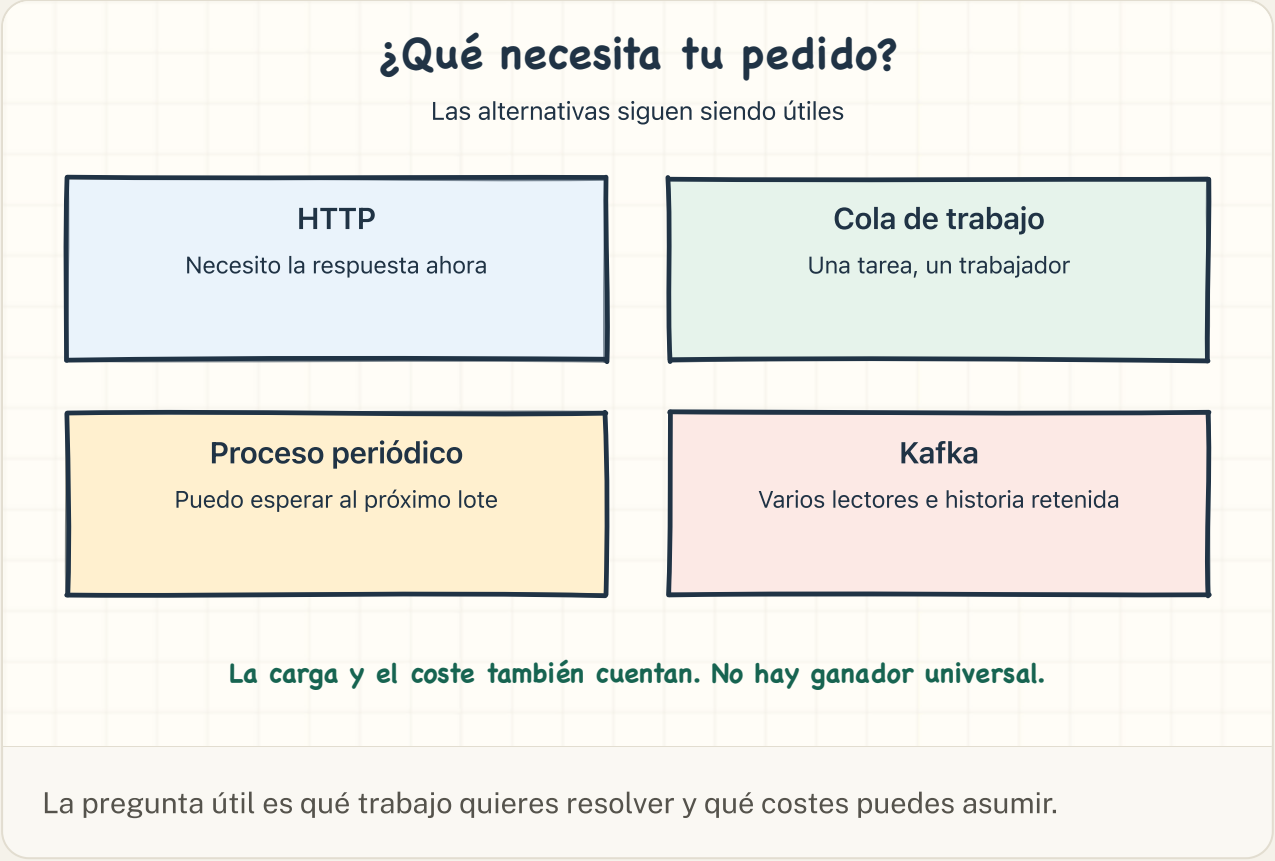
4. Cuándo merece la pena y cuándo complica de más

Empieza por una frase de negocio

Antes de dibujar topics, completa esta frase: “Necesitamos que estos lectores procesen estos hechos, con esta demora aceptable, durante esta ventana de recuperación”. Si no puedes nombrar los lectores ni la ventana, probablemente estás decidiendo demasiado pronto. Tampoco necesitas un documento enorme: una compra, dos lectores y una exigencia de reconstrucción ya permiten comparar alternativas con bastante claridad.

Para nuestra tienda, supongamos que notificaciones debe reaccionar pronto y analítica puede ir unos minutos por detrás. Ambos necesitan observar todas las compras. Además, analítica debe poder rehacer sus cálculos de las últimas veinticuatro horas después de corregir un error. Es un caso razonable para estudiar Kafka. No demuestra que sea la opción más barata, pero sí nombra propiedades que conviene evaluar.

Si el único requisito es enviar un correo por compra, una cola de trabajo puede ofrecer un modelo más directo. Si necesitas confirmar el saldo antes de aceptar una compra, una llamada síncrona y una transacción local quizá sigan siendo esenciales. Publicar eventos después no elimina esa decisión. La comunicación asíncrona tiene sentido cuando puedes definir qué queda pendiente y cómo lo verá el usuario.



La pregunta útil es qué trabajo quieres resolver y qué costes puedes asumir.

Necesidad dominante	Primera alternativa que evaluaría	Qué verificaría
Obtener una respuesta inmediata de otro sistema	HTTP/RPC	Latencia, timeout, disponibilidad e idempotencia de reintentos
Entregar tareas a trabajadores	Cola de trabajo	Confirmaciones, reintentos, enrutamiento y tratamiento de fallos
Consolidar información cada cierto intervalo	Proceso periódico	Ventana, carga sobre el origen y recuperación de lotes
Mantener un flujo para varios lectores y replay	Kafka u otro log distribuido	Retención, particiones, contratos y capacidad de operación

Qué ganas cuando encaja

Ganas independencia de ritmos: un lector puede avanzar detrás de otro mientras los datos sigan retenidos. Ganas una forma de incorporar nuevos consumidores sin pedir al productor que agregue una llamada por cada integración. Ganas una base para reprocesar hechos después de corregir lógica. Y ganas una unidad de paralelismo, la partición, que puedes distribuir entre miembros de un grupo para repartir trabajo.

Lo que tienes que pagar

El primer coste es conceptual. Un mensaje enviado no significa que el negocio terminó. Un consumidor activo no significa que esté al día. Un evento leído no significa que su efecto quedó confirmado. Vas a trabajar con progreso parcial, recuperación y contratos que deben convivir en el tiempo. Si tu equipo todavía no puede observar esos estados, Kafka puede esconder problemas detrás de un “lo publicamos” que suena demasiado definitivo.

El segundo coste es operativo: almacenamiento, transferencia, capacidad de consumidores, actualizaciones, permisos y atención a incidentes. Un proveedor gestionado reduce parte del trabajo de infraestructura, pero no decide tu clave de partición ni vuelve idempotente el correo. El tercero es organizativo: quién es dueño del evento, quién autoriza un cambio y quién atiende un consumidor que lleva horas detenido.

Cuándo elegiría algo más sencillo

En un CRUD pequeño con una única integración y sin necesidad de reconstrucción, empezaría por mantener la simplicidad. Para una tarea programada nocturna quizá basta una consulta incremental bien diseñada. Para solicitudes donde el cliente necesita una decisión inmediata, diseñaría primero ese contrato de respuesta. Y si nadie puede responsabilizarse de operar el flujo, no confundiría instalar un contenedor con haber resuelto el sistema.

> **Lo que debes llevarte:** Kafka se justifica por las propiedades del flujo y de su recuperación. La arquitectura debe poder explicar esas propiedades sin empezar por el nombre del producto.

---

5. Los fundamentos que necesitas, sin exigir una arquitectura

Un evento cuenta; un comando pide

“Crea el pedido” es una petición de acción. Puede ser rechazada porque falta información o porque el negocio no permite completarla. “Pedido creado” expresa que algo ya ocurrió dentro del límite que definimos. Un consumidor puede reaccionar a ese hecho, pero no debería reinterpretarlo como una solicitud todavía pendiente de aprobación. El nombre en pasado ayuda, aunque lo importante es el contrato que el equipo acuerda.

La asincronía separa momentos. El productor puede terminar su publicación antes de que el consumidor termine su trabajo. Para el usuario final necesitas representar esa diferencia: “recibimos tu pedido” y “tu pedido está listo” son estados distintos. Una respuesta HTTP 202 puede comunicar aceptación de trabajo, pero debes explicar exactamente qué aceptó tu API. En nuestro ejemplo significa que el broker reconoció la publicación.

La red transmite bytes, no tus objetos

Tu clase Java no cruza la red como un objeto vivo. Un serializador la transforma en bytes y un lector interpreta esos bytes con un formato acordado. Aquí usaremos JSON dentro de cadenas: es fácil de inspeccionar y evita introducir un registro de esquemas en la primera práctica. Eso no convierte a JSON en un contrato automático; dos programas pueden interpretar de forma diferente un campo llamado **`fecha`** o **`importe`**.

Separaré tres identificadores que suelen mezclarse. **`pedidoId`** identifica la entidad de negocio. **`eventId`** identifica un hecho concreto, útil para seguirlo y eventualmente deduplicarlo. La clave Kafka determina cómo particionamos con el cliente usado. En este ejemplo elegiremos **`pedidoId`** como clave, pero seguirán siendo conceptos distintos: un pedido puede producir muchos eventos con identificadores diferentes y la misma clave.

Un log y sus posiciones

Un log es una secuencia a la que se agregan registros. Imagina páginas numeradas dentro de una libreta. La numeración permite decir desde dónde quieres seguir leyendo, sin que el escritor necesite saber qué entendió cada lector. En Kafka esa secuencia vive por partición. No hay una sola numeración que ordene todos los topics del clúster. Tendremos que conservar esa precisión cuando hablemos de orden.

Tampoco confundas una secuencia de hechos con una tabla de estado actual. Para responder “¿cuál es el estado del pedido ahora?” quizá necesites procesar varios eventos y construir una vista. Esa vista puede vivir en una base de datos. La elección del mecanismo de comunicación y la elección del mecanismo de consulta están relacionadas, pero no son la misma decisión.

Concurrencia, fallos parciales e idempotencia

Dos consumidores pueden avanzar a ritmos distintos y dos productores pueden publicar al mismo tiempo. En ese mundo necesitas decidir qué orden importa. A menudo el negocio exige respetar la secuencia de un pedido, pero no exige ordenar todas las compras de todos los clientes entre sí. Identificar ese límite permite paralelizar sin prometer un orden global costoso e innecesario.

Un fallo parcial significa que una parte puede completar su acción mientras otra no logra saberlo. El broker puede guardar un evento y perderse la respuesta hacia el productor. Desde el cliente hay incertidumbre: un timeout dice que no obtuvo confirmación a tiempo, no demuestra que el evento no existe. Es una de las razones por las que repetir una operación requiere diseño.

Una operación idempotente tolera repeticiones sin multiplicar su efecto lógico. Registrar dos veces el mismo identificador de evento y aplicar una sola vez su consecuencia es un ejemplo de estrategia posible. En cambio, enviar dos correos con dos llamadas exitosas no se vuelve idempotente porque ambos textos sean iguales. Importan la identidad, el estado y el límite donde se protege el efecto.

La **consistencia eventual** aparece cuando distintas vistas tardan en reflejar el mismo hecho. La compra puede estar publicada mientras el tablero todavía muestra la cifra anterior. No es una excusa para aceptar cualquier retraso: define una expectativa y observa si se cumple. Necesitas saber cuánto atraso tolera el producto y qué comunicarás cuando se supere.

Qué es requisito y qué es una elección

Para comprender la guía	Para ejecutar el laboratorio	Arquitecturas que puedes combinar
Procesos, red y formatos de datos	Docker con Compose	Monolito modular o microservicios
Asincronía y estados parciales	Java 21 y terminales	DDD y arquitectura hexagonal
Secuencias, claves y progreso	Maven Wrapper incluido	CQRS, Event Sourcing y plataformas de datos

Kafka necesita procesos que ejecuten el servidor y clientes compatibles que puedan comunicarse con él. No necesita Spring, Java en todos los consumidores, Kubernetes ni un proveedor de nube. El broker del laboratorio trae su entorno Java en la imagen; tu aplicación Java sí requiere el JDK indicado. Elegimos Spring Boot por continuidad con las otras guías y porque permite centrar la práctica en el flujo.

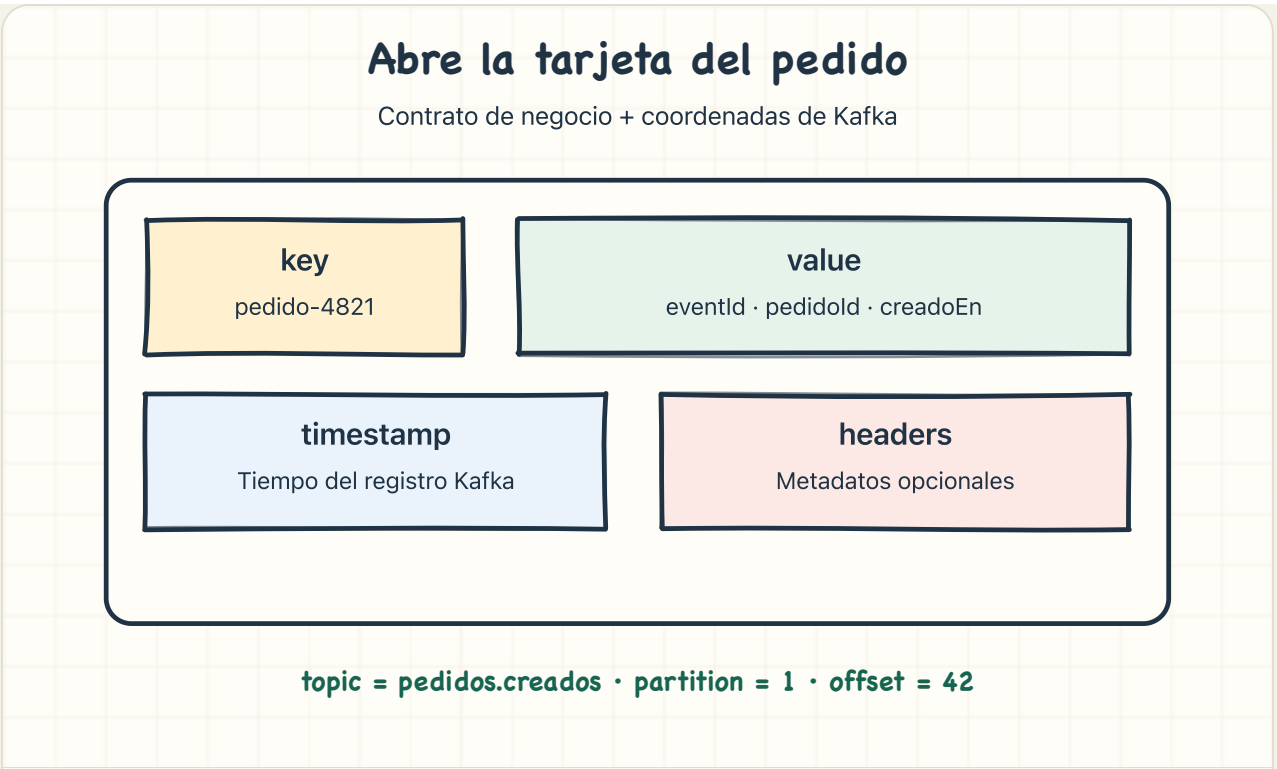
---

6. Las piezas: abre la tarjeta del pedido

Registro, clave y valor

Un **registro** es la unidad que publicamos y leemos. Conceptualmente incluye clave, valor, timestamp y headers opcionales. El valor de nuestro registro será el JSON de **`PedidoCreado`**. La clave será el identificador del pedido. Los headers podrían transportar metadatos como un identificador de correlación; no son un lugar seguro para esconder datos que el consumidor no debería conocer. La introducción de Apache presenta esta terminología.

El timestamp del registro Kafka y **`creadoEn`** dentro de nuestro JSON merecen nombres distintos. El segundo pertenece al contrato del evento; el primero forma parte de la infraestructura de Kafka y su tratamiento depende de configuración. Si después calculas ventanas temporales, necesitarás elegir cuál representa el tiempo de negocio. Leer un evento hoy no significa que el hecho haya ocurrido hoy.



El contrato del evento y sus coordenadas de almacenamiento responden preguntas diferentes.

Topic y partición

Un **topic** es un flujo nombrado, por ejemplo **`pedidos.creados`**. No es una cola individual para cada lector. Dentro del topic existen **particiones**, secuencias ordenadas que permiten repartir almacenamiento y trabajo. Nuestro topic tendrá tres. Cada registro queda en una de ellas, no en las tres por el hecho de que sean tres particiones. Replicar un registro será otra decisión.

Una **clave** ayuda al productor a elegir la partición. Con el particionamiento predeterminado de este cliente, una clave no nula estable llega a la misma partición mientras se mantengan el número de particiones y la estrategia. Así los eventos de **`pedido-4821`** pueden conservar un orden local. No estoy prometiendo que todos los productores del mundo coordinen por sí solos el orden de negocio: también importa en qué secuencia publican.

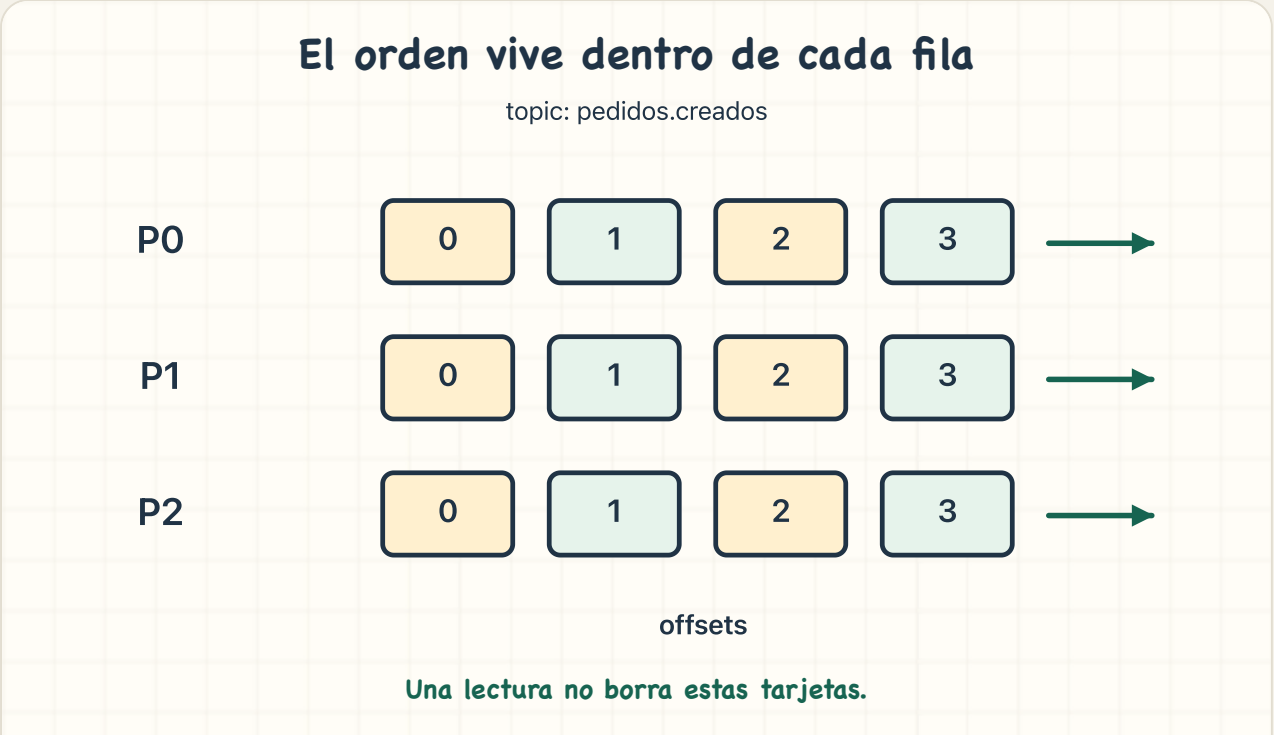
Aumentar particiones cambia las posibilidades de paralelismo, pero también puede cambiar el mapa de claves a particiones. Si dependes del orden histórico por entidad, no lo trates como una ampliación transparente. Define cómo migrar, drenar o versionar el flujo. Un diseño que funciona con tres particiones debe explicar qué ocurrirá cuando necesite seis.

Offset: una coordenada local

El **offset** identifica una posición dentro de una partición. Para ubicar un registro necesitas la combinación topic, partición y offset. El offset 42 de la partición cero y el 42 de la partición uno son coordenadas

diferentes. No puedes compararlos para concluir qué compra ocurrió primero. Tampoco son una identidad de negocio reutilizable en otros sistemas.

Al leer avanzarás una **posición actual**. Al confirmar guardarás un **offset comprometido**, la posición desde la que el grupo podrá continuar. Convencionalmente se confirma el siguiente registro por procesar: después de terminar el 42, el progreso confirmado puede quedar en 43. La configuración del consumidor es la referencia para reset, commits y comportamiento de lectura.



Tres particiones son tres secuencias. No hay un orden total entre las filas.

Productor, broker y controlador

El **productor** es el cliente que publica. En nuestra aplicación, `KafkaTemplate` facilita usar el cliente Java y recibir un resultado asíncrono. El **broker** es el proceso servidor que recibe peticiones y almacena particiones. Un **clúster** reúne los procesos coordinados para ofrecer el servicio. Un clúster de un broker sirve para aprender, pero no mantiene otra copia en otro broker.

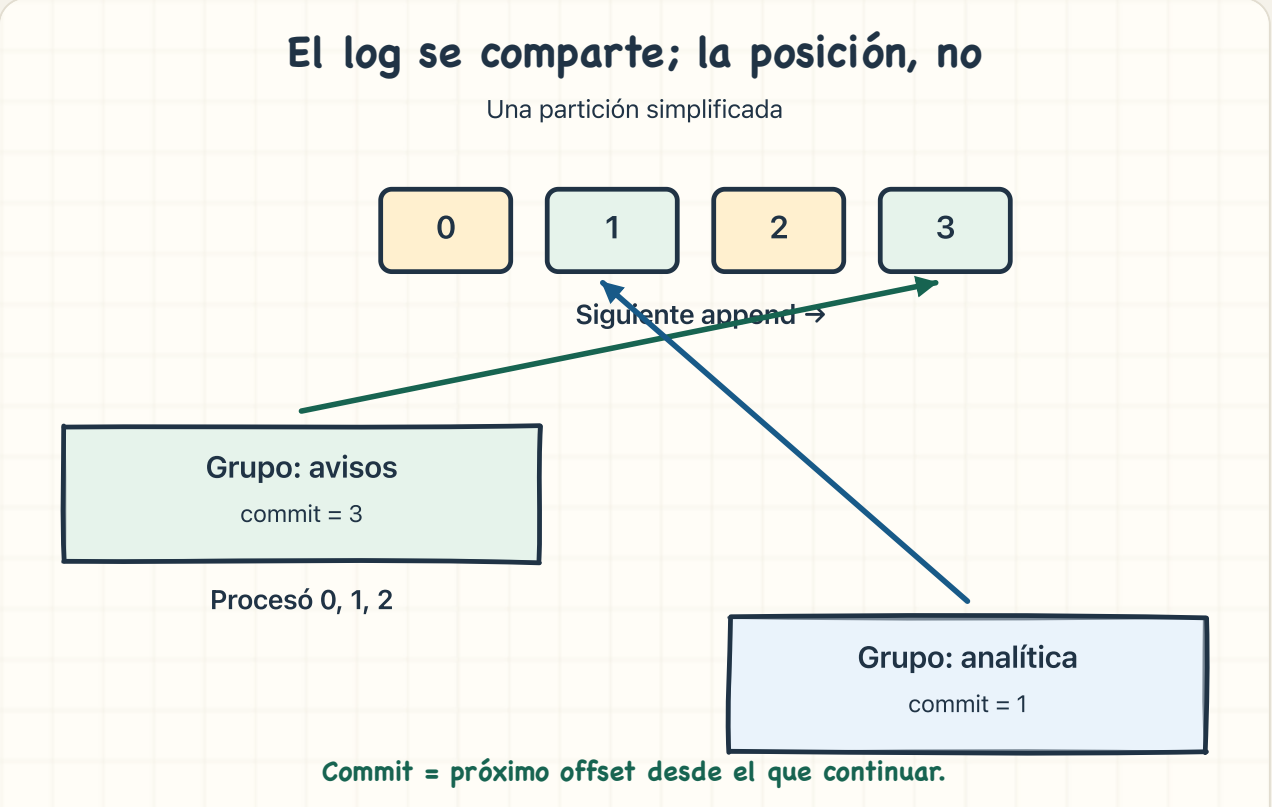
El **controlador** administra metadatos del clúster: brokers registrados, particiones, líderes y cambios relacionados. En KRaft, los controladores utilizan un log de metadatos y consenso Raft. Eso no significa que cada pedido se escriba mediante una transacción Raft en todos los brokers. La replicación del contenido de las particiones tiene su propio mecanismo. Mantener separados el plano de metadatos y los registros de negocio evita una confusión frecuente.

En el laboratorio un mismo proceso tendrá los roles de broker y controlador. Es una comodidad local. La guía de KRaft explica el quórum y los roles para escenarios operativos. No deduzcas de un contenedor combinado la topología que necesitas para una instalación crítica.

Consumidor y grupo

El **consumidor** solicita registros, los interpreta y ejecuta su trabajo. Un **grupo de consumidores** coordina miembros que se reparten particiones de una suscripción. En el modelo que usamos, cada partición queda asignada a un miembro a la vez dentro de ese grupo. Otro grupo puede leer las mismas particiones desde su propia posición sin competir por los registros con el primero.

Si hay tres particiones y dos consumidores en **`avisos-demo`**, uno puede recibir dos particiones y el otro una. El reparto concreto depende del asignador y del estado del grupo. Si agregas un cuarto miembro, no habrá cuatro particiones para repartir: alguno quedará sin trabajo para ese topic. Agregar procesos no elimina el límite del paralelismo que elegiste.



Avisos y analítica comparten registros, pero no su progreso. Confirmar no borra el log.

Un reparto posible de tres particiones

Sin asignar	Miembro A	Miembro B
	✓ P0 ✓ P1 ✓ P2	

Las fichas representan asignaciones de particiones, no registros que se borran al leer.

Entra A. Puede leer las tres particiones del topic en este grupo.

Entra B. Tras el rebalance, un reparto posible asigna dos particiones a A y una a B. No hay propiedad compartida dentro del grupo.

B sale. Tras la reasignación, A retoma todas desde los offsets confirmados del grupo. El log sigue conservando sus registros.

> – Para profundizar
>
> en los **Share Groups**, los miembros comparten trabajo mediante adquisición y confirmación por registro, con bloqueo temporal y posibilidad de redelivery. Así pueden colaborar más consumidores que particiones. En nuestros consumer groups convencionales, cada partición tiene un único miembro asignado y se confirma una posición del log; un registro ya leído puede reaparecer tras un fallo. Son modelos de progreso diferentes, con APIs y configuración propias. El laboratorio usa `KafkaConsumer`, no `KafkaShareConsumer`. Consulta Share Consumer en el diseño oficial de Kafka 4.4 antes de trasladar reglas de un modelo al otro.

---

7. El viaje por dentro: de enviar a confirmar

1. Descubrir adónde escribir

Tu aplicación conoce una dirección inicial mediante `bootstrap.servers`. Esa dirección permite obtener metadatos: qué brokers forman parte del clúster y dónde están los líderes de las particiones. El productor no manda necesariamente todos los registros al primer servidor de la lista para que este los reparta. Usa la información recibida para contactar al destino correspondiente.

Esta es la razón práctica de un error muy común en Docker: conectas a `localhost:9092`, pero el broker anuncia un nombre que tu aplicación no puede resolver. El primer contacto funciona y la siguiente conexión falla. `listeners` define dónde escucha el proceso; `advertised.listeners` define las direcciones que presenta a sus clientes. La configuración de listeners explica esa diferencia. Usaremos una dirección para el host y otra para la red interna.

Tres responsabilidades que no debes mezclar

Rol	Qué coordina	En el recorrido del pedido
Controlador KRaft	Metadatos del clúster y cambios de liderazgo	Registra qué broker lidera cada partición
Líder de una partición	Escrituras y replicación de ese log	Recibe nuestro `Produce`
Coordinador del grupo	Membresía, reparto y offsets del grupo	Ayuda a `avisos-demo` a continuar su lectura

Un broker puede desempeñar varios papeles. El coordinador del grupo es un broker responsable de ese grupo; no es necesariamente el líder de la partición del pedido. KRaft no vota por cada `PedidoCreado`: el consenso de metadatos y la replicación de eventos son recorridos distintos.

2. Serializar, particionar y agrupar

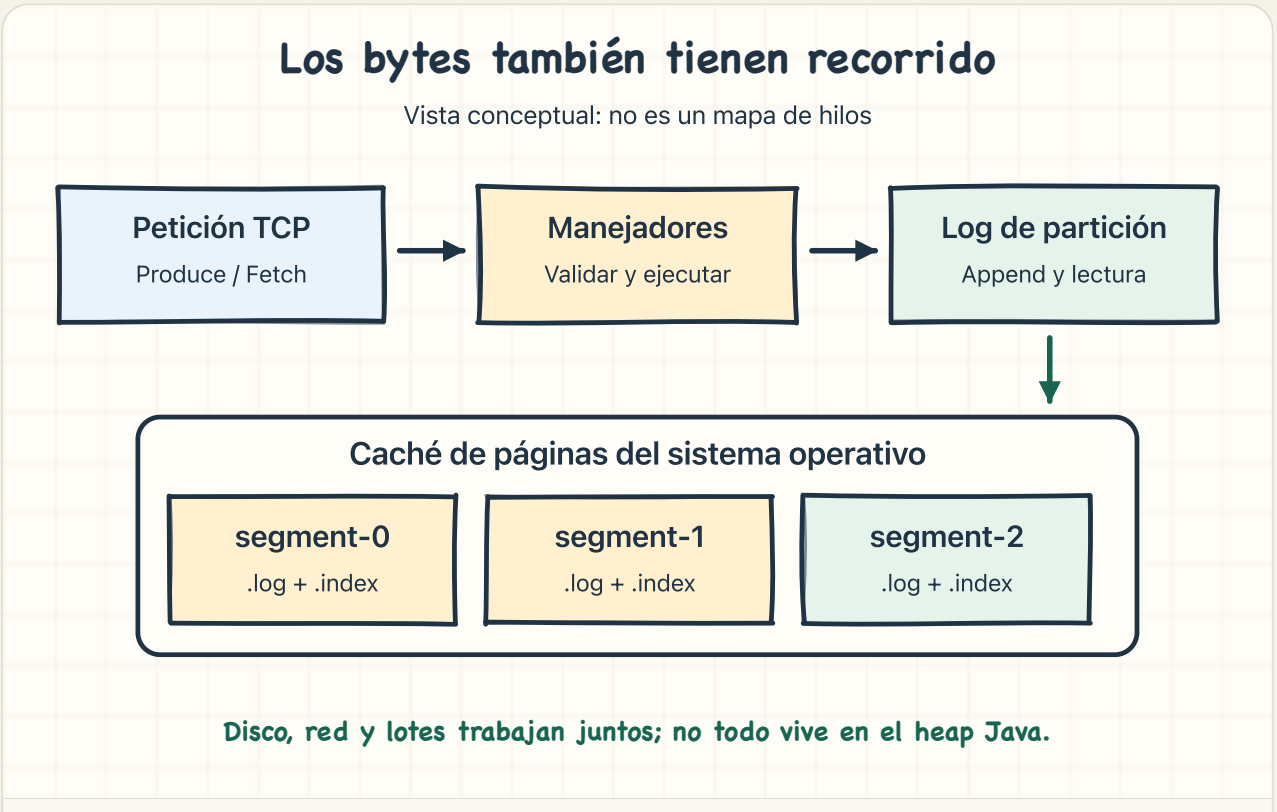
El productor transforma clave y valor en bytes y determina una partición. En lugar de pagar el coste de una petición por cada pequeño registro, puede agruparlos en **batches**. Esta es una parte importante del rendimiento: amortizar costes. El tamaño de los lotes y el tiempo que se permite esperar para formarlos influyen en capacidad, memoria y latencia.

Los detalles de **acks**, idempotencia, timeouts y batching pertenecen al cliente productor. En la práctica fijaremos tiempos acotados para no dejar una petición HTTP esperando indefinidamente. Si esos tiempos vencen, informaremos incertidumbre. Un timeout no nos autoriza a responder "no existe ningún evento" porque no sabemos si falló la escritura o su confirmación.

3. Recibir y agregar al log

El broker recibe peticiones por su capa de red y las procesa mediante manejadores. Una petición **`Produce`** termina agregando registros a los logs apropiados, sujeta a validaciones y condiciones del servidor. Una petición **`Fetch`** pide datos a partir de posiciones determinadas. La capa de red y el formato de mensajes documentan esas interfaces y la organización por lotes.

El log de una partición se divide en **segmentos**. El segmento activo recibe nuevas escrituras; los anteriores permiten mantener y retirar partes del historial sin tratar el topic entero como un solo archivo enorme. Los índices ayudan a localizar posiciones. La implementación del log es una buena lectura para pasar de "Kafka guarda mensajes" a una imagen más concreta de lo que hay en disco.



El log tiene archivos, índices y una ruta de acceso. No es una lista infinita de objetos

dentro del heap.

Kafka aprovecha el sistema operativo y su caché de páginas. La escritura y lectura secuenciales, el trabajo por lotes y determinadas rutas de transferencia eficientes ayudan a mover datos sin crear un objeto Java permanente por cada evento retenido. No confundas esa explicación con “todo está en memoria” ni con “cada confirmación obliga a hacer fsync en todas las réplicas”. Durabilidad, replicación y sincronización física son mecanismos relacionados pero distintos.

> – **Para profundizar**
>
> cuando la ruta lo permite, zero-copy evita copiar los bytes del archivo a un buffer de la aplicación y de vuelta al kernel: el sistema operativo puede transferirlos desde su caché de páginas hacia el socket. Eso reduce copias y trabajo de CPU en la lectura. TLS puede requerir otra ruta para cifrar; no presupongas esa optimización en cualquier conexión. El productor también puede comprimir lotes: ahorra red y disco a cambio de CPU y decisiones de latencia. Tamaño de lote, compresión y caché explican parte del rendimiento; un benchmark con otra distribución de claves no predice tu capacidad.

4. Replicar y reconocer la publicación

Para una partición replicada hay un líder y réplicas seguidoras. El líder atiende las escrituras y los seguidores solicitan registros para mantener sus copias. Las réplicas consideradas al día forman el **ISR**, el conjunto de réplicas sincronizadas. El líder también cuenta dentro de ese conjunto. No confundas tres particiones con tres réplicas: puedes tener tres particiones que existan únicamente en un broker.

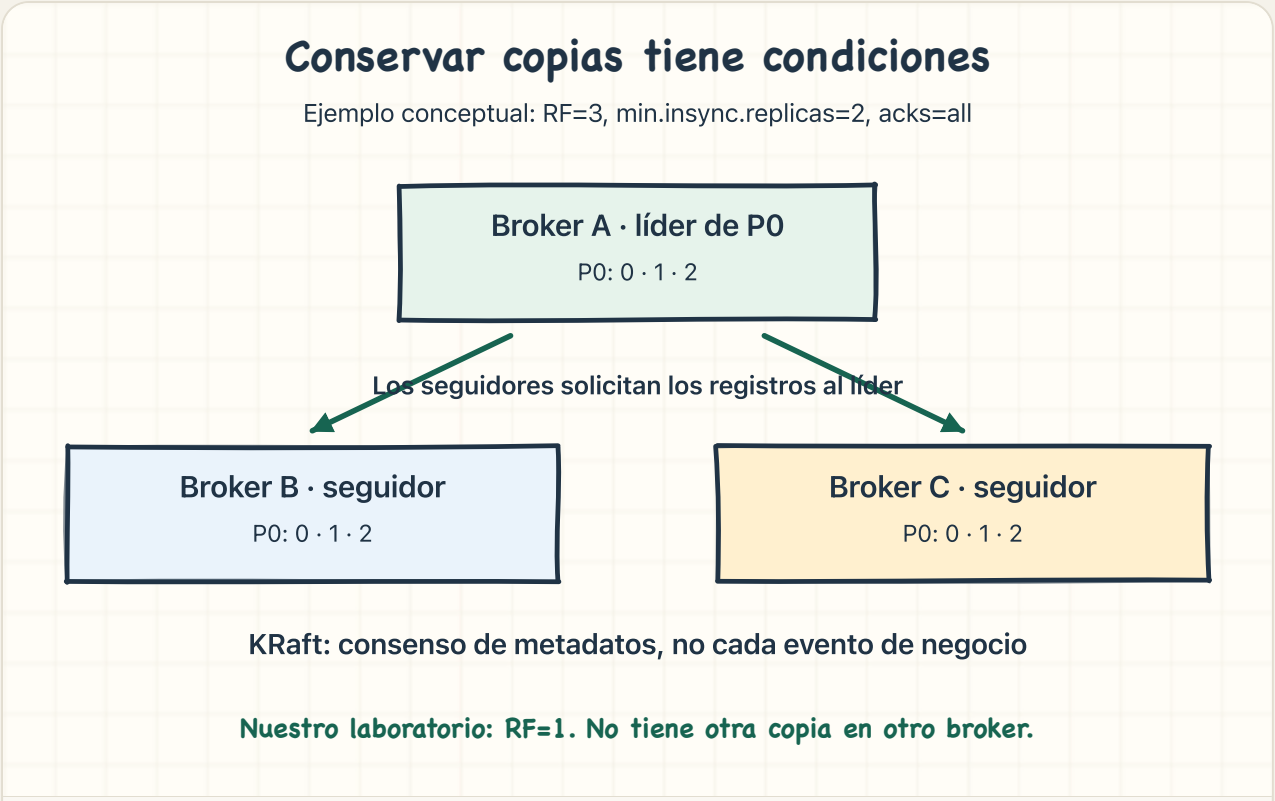
Con **`acks=all`**, el líder espera el reconocimiento de las réplicas del ISR. El mínimo **`min.insync.replicas`** es una condición para aceptar la escritura, no el número fijo de confirmaciones que se esperan. Para un topic con factor de replicación tres y mínimo dos:

ISR en ese momento	¿Qué ocurre con `acks=all`?
Líder + dos seguidores: 3	Se espera a las tres réplicas del ISR
Líder + un seguidor: 2	Se espera a ambas; la réplica fuera del ISR no bloquea por sí sola
Solo el líder: 1	No se cumple el mínimo; la escritura falla

El ISR puede cambiar durante una petición; un error tampoco demuestra por sí solo que ningún byte se haya escrito. Es una política de reconocimiento bajo el estado de replicación, no una promesa contra toda pérdida posible. La configuración del topic precisa estas condiciones.

Nuestro factor es uno. **`acks=all`** reconoce la única réplica disponible; no inventa una segunda. El volumen de Docker ayuda a conservar datos al recrear el contenedor, pero sigue estando en la misma máquina. Si

pierdes ese almacenamiento, no hay otro broker del laboratorio con una copia. La demostración sirve para estudiar comunicación y progreso, no para certificar tolerancia a fallos de infraestructura.



Topología conceptual de producción. El broker único del laboratorio no ofrece esta redundancia.

5. Leer no es retirar

El consumidor consulta registros con **`poll`**. El cliente gestiona conexiones, obtención de datos y participación en el grupo. Spring agrega un contenedor que ejecuta ese ciclo y llama a nuestro método listener. La anotación **`@KafkaListener`** no convierte al broker en quien llama a tu método Java: el programa consumidor mantiene una actividad de lectura mediante el cliente.

Cuando llegan registros, el lector avanza su posición local. Tu código todavía puede fallar al interpretarlos o al aplicar una consecuencia. Por eso debemos decidir cuándo se guarda el progreso. En nuestra configuración, desactivamos el commit automático del cliente y usamos el modo **`record`** del contenedor Spring: después de que el listener termina correctamente, el contenedor confirma el progreso correspondiente. La referencia del listener explica esa política.

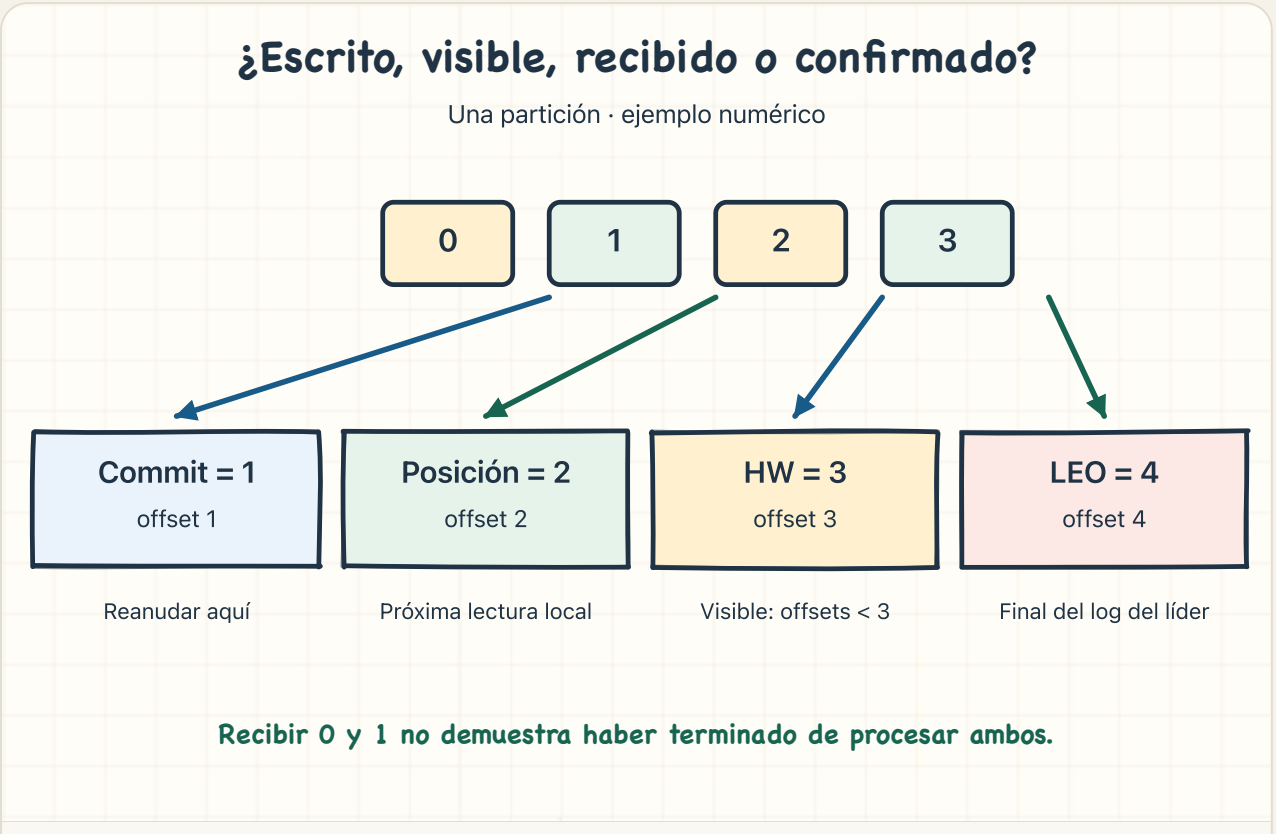
La confirmación se conserva para el grupo mediante mecanismos internos de Kafka, incluido el topic de offsets. No modifica ni borra el evento original. Si otro grupo está atrás, puede seguir leyendo lo que aún existe. Si se retira un segmento por retención, en cambio, el progreso del consumidor no impide que los datos de ese segmento desaparezcan. Retención y consumo tienen responsabilidades separadas.

Cuatro marcas sobre una misma partición

Mira una partición que contiene offsets `0, 1, 2, 3`. Es importante distinguir sus marcas:

Marca	Valor del ejemplo	Qué significa
Log end offset (LEO) del líder	4	Próximo offset al final de su log; incluye el registro 3
High watermark (HW)	3	Límite exclusivo de datos replicados y visibles para lectura ordinaria: 0, 1 y 2
Posición del cliente	2	La siguiente lectura empezará en 2; `poll` ya entregó registros anteriores
Offset confirmado del grupo	1	Un miembro que retome esa partición continuará desde 1

El consumidor pudo recibir 0 y 1, procesar solo 0 y confirmar 1. Si se detiene ahora, el 1 volverá a entregarse aunque su posición local ya fuera 2. Por otro lado, tener el 3 escrito en el líder no basta para volverlo legible si está más allá del HW. La posición del cliente no certifica que tu código haya terminado el trabajo.

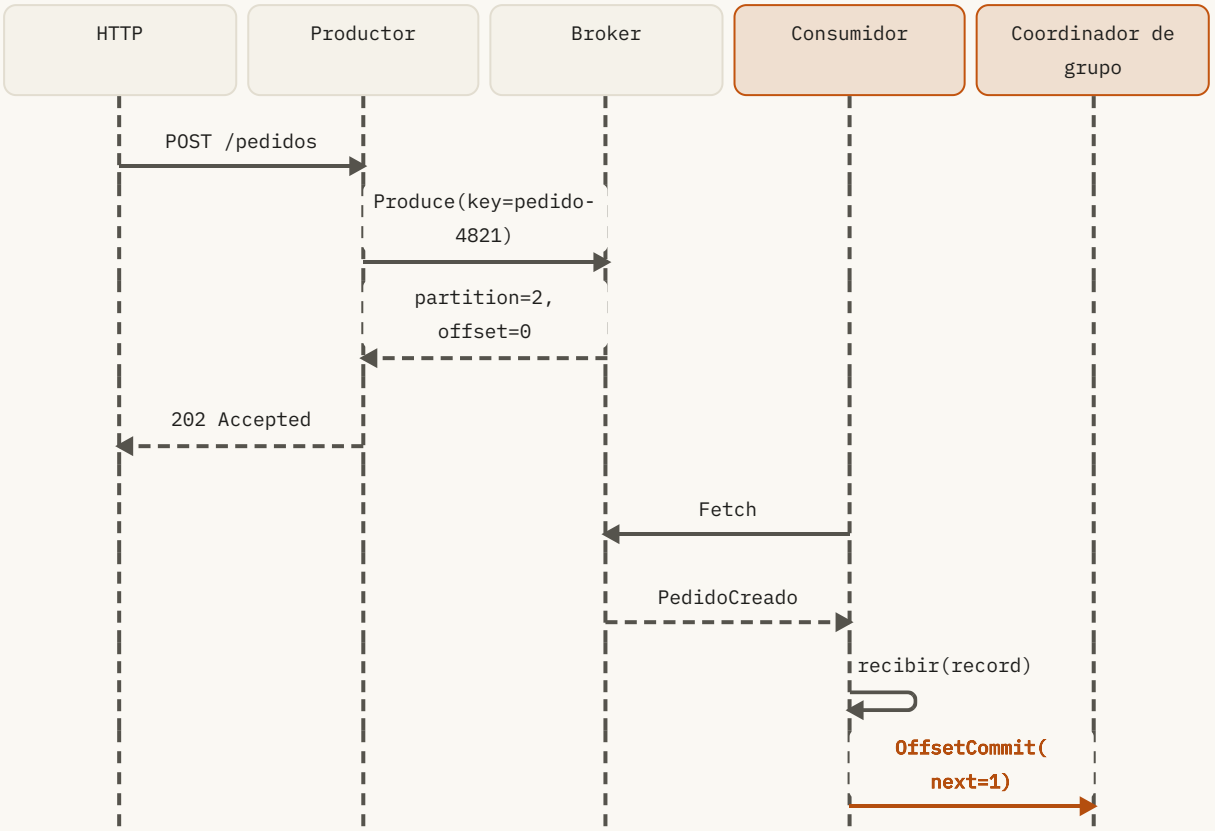


Cuatro marcas diferentes. Todas usan la convención del siguiente offset o límite exclusivo.

– Para profundizar

con **`isolation.level=read_committed`**, las transacciones introducen otra frontera: el **last stable offset (LSO)**. Se toma el menor entre el HW y el primer offset de una transacción todavía abierta; el consumidor no pasa esa frontera y descarta registros de transacciones abortadas. Si una transacción abierta empieza en 1 y el HW vale 3, el LSO vale 1: puede haber datos posteriores ya replicados que ese lector todavía no reciba. No confundas el LSO con tu commit de grupo. Las definiciones están en la API del consumidor.

Publicar, leer y confirmar son pasos distintos



La petición crea un evento en nuestra aplicación. Todavía no está confirmado en Kafka.

El cliente envía los bytes a la partición elegida. En este laboratorio hay una sola réplica.

El broker reconoce la publicación. Las coordenadas son ilustrativas, no valores fijos para toda ejecución.

La respuesta confirma publicación reconocida. El trabajo del consumidor aún puede estar pendiente.

El consumidor solicita datos. El broker no invoca directamente nuestro método Java.

El registro se lee y permanece en el log. Leer no elimina la tarjeta.

El listener interpreta el JSON y escribe el log. Un fallo aquí no es una publicación fallida.

Con ack-mode=record, Spring confirma después del retorno exitoso. El grupo podrá continuar desde el siguiente offset.

6. Retener, compactar y recuperar

En nuestro topic elegimos **`cleanup.policy=delete`** y una retención de veinticuatro horas. La eliminación ocurre por segmentos y tareas de mantenimiento; no interpretes el valor como una alarma individual que borra cada registro exactamente al cumplir un día. La configuración es una política de conservación, no un compromiso de precisión temporal por evento.

La **compactación** responde a otra necesidad: conservar, con sus matices de limpieza, el valor más reciente por clave. Puede servir para reconstruir un estado, pero no equivale a guardar cada transición de negocio. Si compactas todas las versiones antiguas del pedido, quizá pierdas la secuencia que necesitabas para auditoría. Las tombstones, los lectores atrasados y los plazos de limpieza requieren atención adicional antes de usarlo como mecanismo de estado.

> – **Para profundizar**
>
> imagina un topic compactado con **`0: A=pendiente`**, **`1: B=pendiente`**, **`2: A=pagado`** y **`3: B=null`**. Al limpiar, Kafka puede retirar el 0 porque el 2 conserva el último valor de A. El valor nulo del 3 es una **tombstone**, una marca de eliminación de B; no es el texto JSON **`"null"`**. El limpiador puede retirar el valor anterior de B y, pasado el plazo correspondiente, la propia marca. Los offsets conservados no se renumeran. La limpieza es asíncrona, así que una lectura puede encontrar varias versiones. Un reconstructor debe completar su lectura antes de perder las marcas que necesita; revisa **`delete.retention.ms`** en la configuración del topic.

Un replay significa recorrer de nuevo registros disponibles. No recupera lo que ya eliminó la retención ni reconstruye por arte de magia efectos que dependían de datos externos cambiantes. Si antes consultabas una tarifa y esa tarifa cambió, ejecutar otra vez la misma lógica puede producir un resultado diferente. Diseñar un replay incluye decidir qué información estaba en el evento y qué dependencias necesitas congelar o versionar.

> **Lo que debes llevarte:** publicación, escritura replicada, lectura, procesamiento y commit son pasos distintos. Cuando algo falle, necesitas saber cuál de ellos ocurrió y cuál sigue siendo incierto.

---

8. El laboratorio explicado y sus experimentos

Si seguiste el primer evento, ya tienes Kafka, el productor y un consumidor funcionando. Ahora leeremos sus decisiones y cambiaremos una variable a la vez. Si cerraste la sesión, utiliza los comandos de ese recorrido para volver a arrancarla.

El entorno que acabas de ejecutar

Este es el Compose del proyecto, fijado por versión y digest:

compose.yml

```yaml
services:
  kafka:
    image:
apache/kafka:4.4.0@sha256:833b867a9585a141da455dc2b57431c65cbccb27f11c91f694444cca9f777e
0f
    ports:
      - "127.0.0.1:9092:9092"
    environment:
      CLUSTER_ID: "MkU3OEVBNTcwNTJENDM2Qk"
      KAFKA_NODE_ID: 1
      KAFKA_PROCESS_ROLES: broker,controller
      KAFKA_LISTENERS: EXTERNAL://:9092,INTERNAL://:19092,CONTROLLER://:9093
      KAFKA_ADVERTISED_LISTENERS: EXTERNAL://localhost:9092,INTERNAL://kafka:19092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
EXTERNAL:PLAINTEXT,INTERNAL:PLAINTEXT,CONTROLLER:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: INTERNAL
      KAFKA_CONTROLLER_LISTENER_NAMES: CONTROLLER
      KAFKA_CONTROLLER_QUORUM_VOTERS: 1@kafka:9093
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
      KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
      KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
      KAFKA_AUTO_CREATE_TOPICS_ENABLE: "false"
      KAFKA_LOG_DIRS: /var/lib/kafka/data
      KAFKA_LOG_RETENTION_HOURS: 24
      KAFKA_HEAP_OPTS: -Xms128m -Xmx256m
    volumes:
      - kafka-datos:/var/lib/kafka/data
    healthcheck:
      test: ["CMD", "/opt/kafka/bin/kafka-topics.sh", "--bootstrap-server",
"kafka:19092", "--list"]
      interval: 5s
      timeout: 10s
      retries: 20
      start_period: 15s
volumes:
  kafka-datos:
```

`EXTERNAL` es la entrada desde tu máquina; **`INTERNAL`** permite que las herramientas dentro de la red de Compose hablen con **`kafka:19092`**. El listener del controlador sirve al rol KRaft. Son nombres de nuestra configuración, no tres productos distintos. La exposición externa se limita a la interfaz local. El laboratorio usa texto plano porque está diseñado para tu máquina.

El volumen conserva el log aunque detengas el contenedor. Las opciones de replicación uno permiten ejecutar topics internos con un solo broker. No copies esos valores como receta de producción. El

identificador de clúster es fijo para este laboratorio y debe corresponder al almacenamiento; cambiarlo sobre un volumen existente no equivale a crear un clúster nuevo limpio.

Puedes comprobar el entorno en cualquier momento con **`docker compose ps`** y sus logs. El healthcheck consulta la API de Kafka; el volumen conserva el historial entre sesiones.

Las dependencias y la aplicación

El proyecto hereda Spring Boot 4.1.2 y compila para Java 21. Estas dependencias traen la integración Kafka, HTTP y validación. Dejamos que Boot administre las versiones compatibles:

pom.xml: dependencias

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-kafka</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webmvc</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

La clase de entrada es la habitual. Si todavía no conoces qué configura Boot, la guía de Spring Boot del sitio explica ese recorrido con más espacio:

PedidosApplication.java

```
@SpringBootApplication
public class PedidosApplication {
    public static void main(String[] args) {
        SpringApplication.run(PedidosApplication.class, args);
    }
}
```

A continuación declaramos el topic como parte de la aplicación. **`NewTopic`** permite a la integración administrativa de Spring solicitar su creación al arrancar. Los imports y el proyecto completo están en el repositorio; los fragmentos muestran las piezas que estamos estudiando.

ConfiguracionKafka.java: topic

```
public static final String TOPIC = "pedidos.creados";

@Bean
```

```
NewTopic pedidosCreados(@Value("${demo.topic:pedidos.creados}") String topic) {
    return TopicBuilder.name(topic)
            .partitions(3)
            .replicas(1)
            .config("cleanup.policy", "delete")
            .config("retention.ms", "86400000")
            .build();
}
```

La expresión **`${demo.topic:pedidos.creados}`** pide a Spring un nombre configurable, con **`pedidos.creados`** como valor predeterminado. Las pruebas usan otro nombre para separar sus registros de tu sesión.

Si el topic ya existe con una configuración distinta, no asumas que reiniciar la aplicación lo recrea exactamente como este fragmento. Inspecciona sus propiedades con las herramientas administrativas. El laboratorio nuevo parte de un volumen nuevo; una práctica repetida puede conservar eventos y offsets anteriores, que es precisamente lo que queremos aprender a observar.

El evento y las decisiones de configuración

Nuestro contrato es deliberadamente pequeño: identidad del evento, identidad del pedido e instante de creación. Usamos un **`record`** de Java porque expresa datos inmutables sin convertir el evento en un objeto que el consumidor deba modificar.

PedidoCreado.java

```
public record PedidoCreado(UUID eventId, String pedidoId, Instant creadoEn) {}
```

Esta configuración es tan importante como las clases. Léela antes de ejecutar:

application.yml

```
server:
  port: ${PORT:8091}
spring:
  application:
    name: kafka-pedidos
  kafka:
    bootstrap-servers: ${KAFKA_BOOTSTRAP_SERVERS:localhost:9092}
    admin:
      fail-fast: true
    producer:
      key-serializer: org.apache.kafka.common.serialization.StringSerializer
      value-serializer: org.apache.kafka.common.serialization.StringSerializer
      acks: all
      properties:
        enable.idempotence: true
```

```
      max.block.ms: 5000
      request.timeout.ms: 5000
      delivery.timeout.ms: 10000
  consumer:
    group-id: ${KAFKA_GROUP:avisos-demo}
    auto-offset-reset: earliest
    enable-auto-commit: false
    key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
    value-deserializer: org.apache.kafka.common.serialization.StringDeserializer
  listener:
    ack-mode: record
    concurrency: 1
```

El serializador de cadenas transforma nuestro JSON en bytes; el deserializador devuelve una cadena que el listener interpreta. **`acks=all`** y la idempotencia del productor hacen explícita la intención de publicación. Los tiempos limitan la espera. **`enable-auto-commit=false`** deja la confirmación bajo el contenedor Spring, y **`ack-mode=record`** la relaciona con el retorno exitoso del método que procesa cada registro.

`earliest` se consulta cuando el grupo no tiene una posición válida desde la que continuar. No es una orden de volver al principio en cada arranque. Si **`avisos-demo`** ya confirmó avances, reiniciar con el mismo identificador los respeta. Dentro de unos minutos lo comprobaremos cambiando únicamente el grupo del consumidor.

El productor HTTP

El controlador recibe un **`pedidoId`** corto, crea el evento, lo serializa y publica con esa clave. Espera el resultado para devolver partición y offset. El 202 confirma publicación reconocida; no certifica que se haya enviado un correo ni calculado una venta. No hay esos efectos en este proyecto.

PedidoController.java

```
@RestController
@Profile("productor")
@RequestMapping("/pedidos")
public class PedidoController {
    private final KafkaTemplate<String, String> kafka;
    private final JsonMapper json;
    private final String topic;

    public PedidoController(KafkaTemplate<String, String> kafka, JsonMapper json,
            @Value("${demo.topic:pedidos.creados}") String topic) {
        this.kafka = kafka;
        this.json = json;
        this.topic = topic;
    }

    public record CrearPedido(
```

```
            @Pattern(regexp = "[A-Za-z0-9-]{1,40}") String pedidoId) {}
    public record Publicado(UUID eventId, String pedidoId, int particion, long offset)
{}

    @PostMapping
    @ResponseStatus(HttpStatus.ACCEPTED)
    public Publicado crear(@Valid @RequestBody CrearPedido entrada) {
        if (entrada.pedidoId() == null) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "Falta pedidoId");
        }
        var evento = new PedidoCreado(UUID.randomUUID(), entrada.pedidoId(),
Instant.now());
        try {
            var resultado = kafka.send(topic, evento.pedidoId(),
                    json.writeValueAsString(evento)).get(12, TimeUnit.SECONDS);
            var metadata = resultado.getRecordMetadata();
            return new Publicado(evento.eventId(), evento.pedidoId(),
                    metadata.partition(), metadata.offset());
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            throw noConfirmado(e);
        } catch (ExecutionException | TimeoutException |
org.apache.kafka.common.KafkaException
                | org.springframework.kafka.KafkaException e) {
            throw noConfirmado(e);
        }
    }

    private ResponseStatusException noConfirmado(Exception causa) {
        return new ResponseStatusException(HttpStatus.SERVICE_UNAVAILABLE,
                "Publicación no confirmada; un reintento puede duplicar el evento",
causa);
    }
}
```

Cuando no puede confirmar, devuelve 503 y conserva la advertencia de posible duplicado. La espera bloqueante es intencional en esta práctica pequeña: hace visible el límite de confirmación sin introducir otro flujo HTTP asíncrono. Para una API de mucha carga evaluarías otras formas de manejar concurrencia, pero seguirías necesitando observar el resultado de la publicación.

Cada petición crea un **`eventId`** nuevo. Repetir el mismo **`POST`** no es idempotente a nivel de negocio. La idempotencia del cliente Kafka protege determinados reintentos del protocolo, no deduplica dos solicitudes HTTP distintas. Quiero que esta limitación quede cerca del código y no escondida después de una afirmación de entrega garantizada.

Un error HTTP que sí explica la incertidumbre

El controlador lanza una excepción con contexto, pero no quiero que la advertencia se pierda al formar la respuesta HTTP. Este advice, limitado al controlador de pedidos, devuelve **`ProblemDetail`**: el formato de error de Spring MVC, con estado, detalle y ruta. No publica trazas ni la excepción interna.

```
@RestControllerAdvice(assignableTypes = PedidoController.class)
public class ErroresHttp {
    @ExceptionHandler(ResponseStatusException.class)
    public ProblemDetail responder(ResponseStatusException error) {
        var problema = ProblemDetail.forStatusAndDetail(error.getStatusCode(),
                error.getReason() == null ? "Solicitud no completada" :
error.getReason());
        problema.setTitle(error.getStatusCode().value() == 503
                ? "Publicación no confirmada" : "Solicitud rechazada");
        return problema;
    }
}
```

La prueba HTTP comprueba que un envío fallido produce **`application/problem+json`**, estado 503 y el texto “un reintento puede duplicar” en **`detail`**. También puedes observarlo al detener el broker, solicitar un pedido y volver a levantar Kafka. El productor conserva esperas acotadas.

Al detener el broker, esta fue la respuesta HTTP observada. El texto vive en el cuerpo, no únicamente en el log del servidor:

```
{
  "detail": "Publicación no confirmada; un reintento puede duplicar el evento",
  "instance": "/pedidos",
  "status": 503,
  "title": "Publicación no confirmada"
}
```

El consumidor

El listener recibe el registro completo para que podamos observar también sus coordenadas. Interpreta el JSON y escribe una línea de log. El grupo procede de configuración: ejecutar el mismo código con otro grupo cambia qué progreso comparte.

ConsumidorPedidos.java

```
@Component
@Profile("consumidor")
public class ConsumidorPedidos {
    private static final Logger log = LoggerFactory.getLogger(ConsumidorPedidos.class);
    private final JsonMapper json;

    public ConsumidorPedidos(JsonMapper json) {
```

```
        this.json = json;
    }

    @KafkaListener(id = "lector-pedidos", topics = "${demo.topic:pedidos.creados}",
            groupId = "${spring.kafka.consumer.group-id}")
    public void recibir(ConsumerRecord<String, String> record) {
        PedidoCreado evento;
        try {
            evento = json.readValue(record.value(), PedidoCreado.class);
        } catch (JacksonException error) {
            log.error("PEDIDO_INVALIDO topic={} key={} partition={} offset={}",
                    record.topic(), record.key(), record.partition(), record.offset());
            throw error;
        }
        log.info("PEDIDO eventId={} pedidoId={} key={} partition={} offset={}",
                evento.eventId(), evento.pedidoId(), record.key(),
                record.partition(), record.offset());
    }
}
```

En esta práctica un error de procesamiento detiene el contenedor de escucha. Así un JSON que no puede interpretar no se convierte silenciosamente en un supuesto éxito tras agotar reintentos. Es una política didáctica visible, no una política universal para producción:

ConfiguracionKafka.java: errores

```
@Bean
CommonContainerStoppingErrorHandler detenerAnteError() {
    return new CommonContainerStoppingErrorHandler();
}
```

Si se detiene por error, el proceso puede seguir vivo aunque el listener ya no esté trabajando. Debes corregir la causa y reiniciarlo; volverá a encontrar el registro pendiente. Más adelante veremos opciones como reintentos acotados y un topic de errores. Ninguna debe aplicarse sin pensar qué significa omitir o demorar ese evento.

Cómo leer los seis experimentos

Antes de cambiar algo, escribe tu predicción en una frase. Ejecuta la instrucción, compara el resultado y después abre la explicación. Los IDs y offsets de tu sesión pueden variar; las relaciones entre ellos son las que estamos estudiando.

Experimento 1: publicar y encontrar las coordenadas

Predice: ¿puede el productor publicar si todavía no hay un consumidor listo? ¿Qué confirma exactamente un HTTP 202?

Prueba: publica otro pedido y compara sus coordenadas con la terminal del lector.

```
curl -i -X POST http://localhost:8091/pedidos \
  -H 'Content-Type: application/json' \
  -d '{"pedidoId":"pedido-4821"}'
```

Observa: recibes HTTP 202 con **`eventId`**, **`pedidoId`**, **`particion`** y **`offset`**. En la terminal del consumidor aparece un mensaje **`PEDIDO`** e incluye la misma identidad y sus coordenadas. No copies una partición u offset de un ejemplo para decidir si funcionó: compara la respuesta de tu publicación con la lectura correspondiente.

Esta es una captura de la ejecución verificada el 6 de noviembre de 2026, con el log recortado para mostrar su mensaje. Tus identificadores y coordenadas pueden ser otros:

```
{
  "eventId": "b1f5c34a-e4a1-407a-b0ba-49c02e40cd94",
  "pedidoId": "pedido-4821",
  "particion": 2,
  "offset": 0
}
```

```
PEDIDO eventId=b1f5c34a-e4a1-407a-b0ba-49c02e40cd94 pedidoId=pedido-4821 key=pedido-4821
partition=2 offset=0
```

Después, la consulta del grupo mostró **`CURRENT-OFFSET=1`**, **`LOG-END-OFFSET=1`** y **`LAG=0`** para la partición 2. El evento estaba en 0; el commit guarda 1, la siguiente posición.

> **– Solución**
>
> sí: basta con que Kafka esté disponible y acepte la escritura. Un 202 confirma la publicación, no el procesamiento. Productor y consumidor se relacionan mediante el log. La prueba automatizada también comprueba que el offset confirmado del grupo avanza más allá del registro procesado. El log por sí solo muestra que se ejecutó el método; consultar el grupo muestra otro punto del flujo.

Experimento 2: detener al lector

Predice: ¿qué cambia al detener únicamente al lector? ¿Quién recuperará el pedido publicado durante su ausencia?

Prueba: pulsa **`Ctrl+C`** en la terminal del consumidor. Deja vivos el productor y Kafka. Publica **`pedido-4822`** cambiando el identificador en el comando anterior y reinicia el consumidor.

Observa: el pedido pendiente aparece cuando el grupo retoma su lectura. Quizá veas otros registros pendientes si habías hecho más publicaciones. No necesitas que el productor recuerde qué estaba haciendo el lector ni que le vuelva a mandar ese evento.

> **– Solución**
>
> el broker conserva el registro durante su política de retención. El progreso del grupo permite continuar. La demostración no cubre un consumidor detenido más allá de esa ventana; si los datos desaparecen, necesitarás otra fuente de recuperación o aceptar esa pérdida de historia según el contrato del sistema.

Experimento 3: repartir entre dos consumidores

Predice: con dos miembros y tres particiones, ¿recibirá cada miembro una copia de todos los pedidos o se repartirán el trabajo?

Prueba: deja el primero arriba y abre otra terminal con el mismo comando de consumidor. Espera al rebalance y consulta el grupo:

```
docker compose exec kafka /opt/kafka/bin/kafka-consumer-groups.sh \
  --bootstrap-server kafka:19092 --describe --group avisos-demo --members --verbose
```

Observa: ambos miembros tienen asignaciones y estas no se superponen dentro del grupo. Publica varios pedidos con claves diferentes. No esperes que una sola clave utilice las tres particiones: se mantiene en una. Tampoco asumas que tres claves arbitrarias necesariamente caerán en tres particiones distintas.

> **– Solución**
>
> el reparto ocurre por partición. Un evento será atendido por el miembro responsable de esa partición, mientras otro miembro puede trabajar en otra. La prueba automatizada envía registros a particiones conocidas para comprobar el reparto sin depender de casualidades del hash de nuestras claves.

Experimento 4: otro lector independiente

Predice: ¿un consumidor de **`analitica-demo`** compite con los de **`avisos-demo`**? ¿Puede leer un pedido que avisos ya confirmó?

Prueba: en otra terminal ejecuta:

```
java -jar target/kafka-pedidos-1.0.0.jar \
  --spring.profiles.active=consumidor \
  --spring.main.web-application-type=none \
  --spring.kafka.consumer.group-id=analitica-demo
```

Observa: **`analitica-demo`** puede leer la historia retenida desde el inicio porque es un grupo nuevo y configuramos **`earliest`**. Publica otro pedido. Tanto el grupo de avisos como el de analítica podrán procesarlo. Dentro del primero, sus miembros seguirán repartiéndose trabajo.

> – **Solución**
>
> no duplicamos el topic para cada aplicación ni borramos el registro al confirmar. Duplicamos la interpretación y el progreso. En una solución real cada grupo puede ejecutar otro código; usamos el mismo listener para aislar una sola variable del experimento.

Experimento 5: volver a leer

Predice: ¿reiniciar **`analitica-demo`** y arrancar un grupo nuevo producen el mismo replay? ¿Qué posición tiene prioridad cuando ya existe un commit?

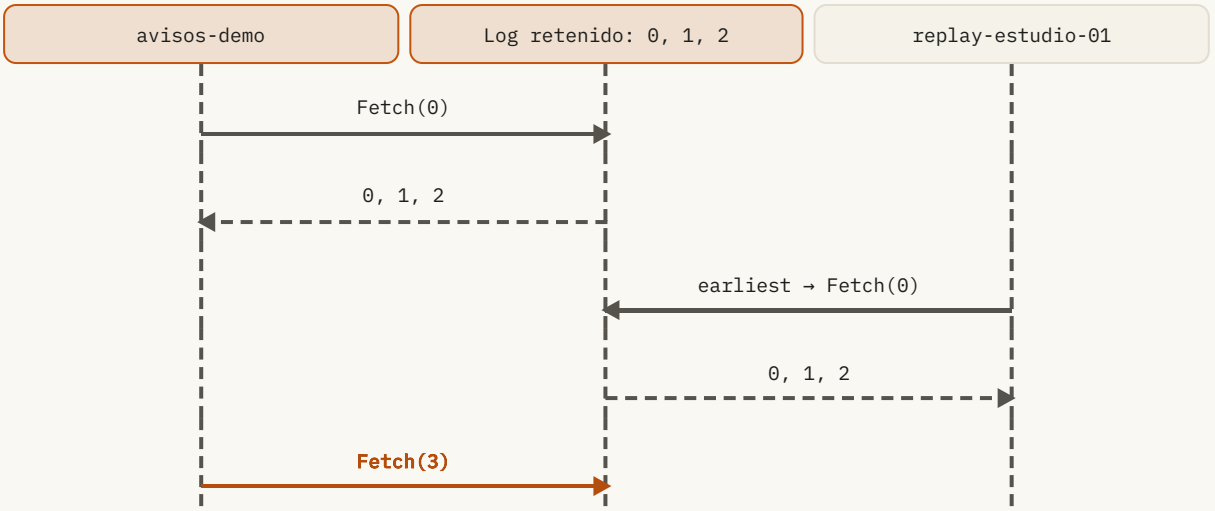
Prueba: reinicia primero **`analitica-demo`**. Después usa un grupo nuevo y reconocible:

```
java -jar target/kafka-pedidos-1.0.0.jar \
  --spring.profiles.active=consumidor \
  --spring.main.web-application-type=none \
  --spring.kafka.consumer.group-id=replay-estudio-01
```

Observa: el grupo nuevo recorre registros todavía retenidos. Si ya utilizaste ese nombre, elige otro como **`replay-estudio-02`**. La prueba verifica precisamente la diferencia entre retomar un grupo con commits y comenzar otro con **`earliest`**.

> – **Solución**
>
> replay describe una nueva lectura. No significa repetir sin control correos, cobros o llamadas externas. Para análisis puedes escribir una vista nueva y reemplazarla cuando esté lista; para efectos irreversibles necesitas un diseño de idempotencia y un modo de reconstrucción que sepa qué consecuencias debe omitir.

Replay: otro grupo, la misma historia retenida



Avisos comienza en 0 y procesa los tres registros disponibles.

Las tarjetas permanecen. El grupo confirma su siguiente posición: 3.

El grupo de replay es nuevo y no tiene commits; earliest elige el comienzo disponible.

El segundo lector ve la misma historia. No mueve los commits de avisos.

Al volver, avisos continúa en 3. earliest no reinicia un grupo con progreso válido. Ningún lector recupera datos ya eliminados por retención.

Experimento 6: misma clave y orden local

Predice: al repetir una clave, ¿qué esperas de la partición, los offsets y el **`eventId`**?

Prueba: repite el **`POST`** con **`pedido-4821`** tres veces, esperando cada respuesta antes del siguiente. Compara las tres respuestas y sus líneas en el consumidor.

Observa: la partición permanece y los offsets crecen. Pueden existir registros intercalados de otras claves. Eso no rompe el orden de este pedido. Si disparas peticiones simultáneas, en cambio, no conviertas el orden de tus clics o de sus respuestas HTTP en una secuencia de negocio que nunca definiste en el productor.

– Solución

conservas partición y offsets crecientes, pero cada petición crea otro **`eventId`**. Nuestra garantía se estudia en un productor, con clave estable y número de particiones fijo. La lógica del consumidor también importa: si recibe en orden pero lanza cada efecto a tareas concurrentes sin coordinación, puede terminar esos efectos fuera de orden. Orden de almacenamiento y orden de finalización son preguntas distintas.

Inspeccionar, probar y cerrar

Puedes consultar la estructura del topic y el atraso de los grupos:

```
docker compose exec kafka /opt/kafka/bin/kafka-topics.sh \
  --bootstrap-server kafka:19092 --describe --topic pedidos.creados

docker compose exec kafka /opt/kafka/bin/kafka-consumer-groups.sh \
  --bootstrap-server kafka:19092 --describe --group avisos-demo
```

`CURRENT-OFFSET` representa progreso confirmado y **`LOG-END-OFFSET`** el extremo del log que muestra la herramienta. Su diferencia ayuda a observar atraso, con los matices de transacciones y registros internos del modelo general. No es por sí sola una medida de segundos: dos mil registros pueden representar un segundo de una carga o una hora de otra.

Desde el laboratorio puedes ejecutar **`./mvnw test`** con el broker local activo. Para repetir el ensayo aislado utilizado en autoría, vuelve a la raíz del repositorio:

```
pnpm spring:ejemplos run --proyecto kafka
pnpm spring:ejemplos check --proyecto kafka
```

El runner prepara dependencias y luego prueba sin salida a Internet, con un broker real en una red interna. Conserva evidencia de la fuente y del resultado. Para cerrar tu sesión local, detén las aplicaciones con **`Ctrl+C`** y ejecuta **`docker compose down`**. El volumen permanece. Si quieres eliminar deliberadamente **todos los datos y offsets del laboratorio**, usa **`docker compose down -v`**; la siguiente sesión empezará con otro historial vacío.

---

9. Lo que pasa cuando algo falla

La confirmación que se perdió

La aplicación crea un **`eventId`** nuevo en cada petición HTTP. Si Kafka acepta el registro pero la respuesta no llega a tiempo, un reintento de **`POST /pedidos`** puede publicar otro evento con el mismo **`pedidoId`** y otro **`eventId`**. La idempotencia del productor evita duplicados de ciertos reintentos del protocolo; no reconoce dos peticiones de negocio como una sola compra.

Por eso devolvemos 503 con un cuerpo que dice **publicación no confirmada** y advierte que un reintento puede duplicar. La respuesta no dice que el pedido se haya procesado ni que sea seguro repetirlo sin comprobar nada. Para una API real, diseñaría una clave de idempotencia persistente y una política para reconciliar resultados inciertos; ese estado queda fuera del laboratorio.

Procesar y confirmar no son una sola operación

Imagina que el consumidor envía un correo, pero termina antes de confirmar el offset. Al volver, puede recibir el registro otra vez. El correo ya salió. Si confirmas antes de enviarlo, el fallo inverso puede perder el trabajo. El orden de las dos operaciones cambia qué riesgo aceptas; no puede convertir dos sistemas independientes en una transacción atómica.

Política	Punto vulnerable	Consecuencia posible
Confirmar antes de aplicar el efecto	El proceso falla después del commit	Omitir el efecto
Aplicar el efecto y después confirmar	El proceso falla antes del commit	Repetir el efecto
Coordinar progreso y efecto en el mismo límite transaccional	Requiere soporte y diseño adecuados	Garantías más fuertes dentro de ese límite

“Al menos una vez” exige que el procesamiento tolere repeticiones. No significa que debas aceptar un número ilimitado de correos ni que Kafka repita siempre todo. Describe una posibilidad que aparece en determinadas recuperaciones. En nuestro laboratorio imprimir dos veces sería inofensivo; sustituir ese log por un cobro cambiaría radicalmente la responsabilidad del código.

> – **Para profundizar**
>
> imagina un procesador que lee **`pedidos.creados`**, calcula una salida a **`pedidos.enriquecidos`** y avanza su offset. Una transacción Kafka puede confirmar **la salida y el progreso de entrada juntos**. Si aborta, un lector **`read_committed`** no verá esa salida y el procesador podrá volver a leer la entrada. Kafka Streams puede coordinar ese modelo con su configuración de procesamiento exactly-once. El correo enviado por HTTP no participa en esa transacción: podría haberse enviado antes del aborto. Siempre nombra las entradas, salidas y lectores incluidos en la garantía. Consulta el protocolo de transacciones antes de extenderla a otros sistemas.

Un registro que siempre falla

Un JSON inválido puede fallar sin importar cuántas veces lo reintentes. Una caída temporal de otro servicio puede recuperarse después. Tratarlos igual produce incidentes distintos: el primero bloquea trabajo indefinidamente; el segundo puede descartarse demasiado pronto. Clasifica errores antes de elegir cuántos intentos harás y cuánto tiempo puedes demorar la partición.

La práctica detiene el listener ante error mediante **`CommonContainerStoppingErrorHandler`**. Es una forma deliberada de mantener visible el problema. La aplicación requiere intervención para continuar. En producción podrías decidir reintentos acotados y publicación a un topic de errores, con contexto suficiente para investigar. Un topic de errores también necesita dueño, retención, permisos y una herramienta o procedimiento de recuperación.

Enviar un registro a otro topic y continuar puede cambiar el orden de negocio. Si **PedidoPagado** falla y dejas avanzar **PedidoEnviado**, quizá proceses consecuencias fuera de secuencia. No resuelvas eso con la frase “tenemos una DLQ”. Debes decidir si puedes continuar, qué estado representará el pendiente y cómo introducirás la corrección después. La referencia de errores de Spring Kafka explica mecanismos; la política pertenece a tu dominio.

Recuperar el laboratorio sin borrar su historia

Este procedimiento es opcional y se ejecuta al terminar los seis experimentos. Necesitas Kafka, el productor HTTP y al menos un consumidor de **avisos-demo** encendidos, con las terminales en **examples/spring/kafka**. Si ya cerraste el laboratorio, repite el arranque del primer evento antes de continuar.

Inyecta un registro inválido con clave conocida para ver el bloqueo; después publica un pedido válido con esa misma clave. Ambos irán a la misma partición mientras no cambie su número.

```
docker compose exec -T kafka /opt/kafka/bin/kafka-console-producer.sh \
  --bootstrap-server kafka:19092 --topic pedidos.creados \
  --producer-property acks=all --property parse.key=true --property key.separator=:
<<'EOF'
pedido-error:no-es-json
EOF

curl -sS -X POST http://localhost:8091/pedidos \
  -H 'Content-Type: application/json' -d '{"pedidoId":"pedido-error"}'
```

El listener registra **PEDIDO_INVALIDO** con topic, clave, partición y offset antes de detenerse. Anota esas coordenadas. Detén **todos los consumidores de avisos-demo**, incluidos procesos que sigan vivos aunque su listener haya parado. Las otras particiones también dejan de atenderse en ese contenedor: el error handler detiene el listener, no solo el registro fallido.

Primero conserva el registro y el contexto del incidente. En la ejecución documentada, el inválido estaba en partición 0, offset 0. **Sustituye ambos números por los de tu log** antes de copiar el comando. Este formato de terminal es suficiente para nuestras claves y valores de una línea; un recuperador general debe preservar también headers y metadatos que necesite tu contrato, sin depender de separadores de texto.

```
PARTICION=0
OFFSET=0
SIGUIENTE=$((OFFSET + 1))

docker compose exec -T kafka /opt/kafka/bin/kafka-console-consumer.sh \
  --bootstrap-server kafka:19092 --topic pedidos.creados \
  --partition "$PARTICION" --offset "$OFFSET" --max-messages 1 --timeout-ms 5000 \
```

```
  --formatter-property print.key=true --formatter-property key.separator=: > registro-
problematico.txt

printf 'topic=pedidos.creados partition=%s offset=%s\n' \
  "$PARTICION" "$OFFSET" > incidente.txt

docker compose exec -T kafka /opt/kafka/bin/kafka-topics.sh \
  --bootstrap-server kafka:19092 --create --if-not-exists \
  --topic pedidos.errores --partitions 1 --replication-factor 1

docker compose exec -T kafka /opt/kafka/bin/kafka-console-producer.sh \
  --bootstrap-server kafka:19092 --topic pedidos.errores \
  --producer-property acks=all --property parse.key=true --property key.separator=: \
  < registro-problematico.txt
```

Verifica que el productor de consola termina sin error y que **`pedidos.errores`** contiene esa clave y valor. Puedes inspeccionarlo con **`kafka-console-consumer.sh`**, cambiando el topic y usando **`--from-beginning --max-messages 1 --timeout-ms 5000`**; si ya había incidentes, localiza el registro nuevo en lugar de asumir que será el primero. Guarda **`incidente.txt`** junto al resultado de la revisión. No avances el grupo si la copia no quedó confirmada.

Después consulta el grupo: el estado debe ser **`Empty`** y tener cero miembros. El dry run debe proponer **solo esa partición**, con el siguiente offset del registro problemático. Revisa la salida antes del comando **`--execute`**; saltar un registro es una decisión explícita de recuperación, no una corrección del evento original.

```
docker compose exec kafka /opt/kafka/bin/kafka-consumer-groups.sh \
  --bootstrap-server kafka:19092 --describe --group avisos-demo --state

docker compose exec kafka /opt/kafka/bin/kafka-consumer-groups.sh \
  --bootstrap-server kafka:19092 --group avisos-demo \
  --topic "pedidos.creados:$PARTICION" --reset-offsets \
  --to-offset "$SIGUIENTE" --dry-run

docker compose exec kafka /opt/kafka/bin/kafka-consumer-groups.sh \
  --bootstrap-server kafka:19092 --group avisos-demo \
  --topic "pedidos.creados:$PARTICION" --reset-offsets \
  --to-offset "$SIGUIENTE" --execute
```

Reinicia un consumidor de **`avisos-demo`**. El pedido válido que estaba detrás debe aparecer y su progreso avanzar. Otro grupo desde el inicio todavía encontrará el JSON inválido: no borramos ni arreglamos la historia. Los commits de los demás grupos permanecen independientes. En un sistema real necesitas además investigar, corregir y volver a introducir el trabajo con una política acordada; la cuarentena no termina el incidente.

Rebalances y consumidores lentos

Cuando cambian miembros o suscripciones, el grupo puede reasignar particiones. Ese proceso es un **rebalance**. El consumidor que antes trabajaba una partición puede dejar de tenerla y otro retomarla desde su progreso confirmado. No debes guardar el estado de propiedad como una suposición permanente fuera del mecanismo del cliente.

La duración del procesamiento influye en la salud del grupo. Si haces trabajo muy lento entre consultas, puedes superar límites y provocar reasignaciones. No empieces aumentando todos los timeouts: mide qué tarda, reduce lotes si corresponde y diseña concurrencia conservando la propiedad de las particiones. El protocolo de rebalance ha evolucionado; revisa cliente y configuración antes de copiar recomendaciones de otra versión.

Un consumidor puede estar conectado y aun así atrasarse cada minuto. Si recibe cien eventos por segundo y solo procesa ochenta, la deuda crece. Un reinicio no crea capacidad adicional. Necesitas reducir trabajo, ampliar paralelismo útil o redistribuir la carga. Si casi todo el tráfico usa una única clave, agregar particiones no repartirá automáticamente esa entidad entre muchos consumidores conservando su orden local.

El problema de escribir en dos lugares

Si mañana agregas una base de datos, aparecerá otra frontera. Guardas el pedido y luego publicas; el proceso falla entre ambos. Tienes un pedido sin evento. Publicas primero y falla el guardado; tienes un evento sobre un pedido que no quedó registrado. Cambiar el orden no elimina la ventana de fallo.

Un **outbox transaccional** guarda en una misma transacción local el cambio de negocio y la intención de publicar. Después un relay o mecanismo CDC transmite los pendientes. Ese envío puede repetirse, así que los lectores todavía necesitan una estrategia de idempotencia. Debezium documenta un outbox event router como una opción de implementación. Nuestro ejemplo no lo necesita porque no escribe en una base de datos, pero reconocer el problema evita convertir el laboratorio en un diseño productivo por simple copia.

---

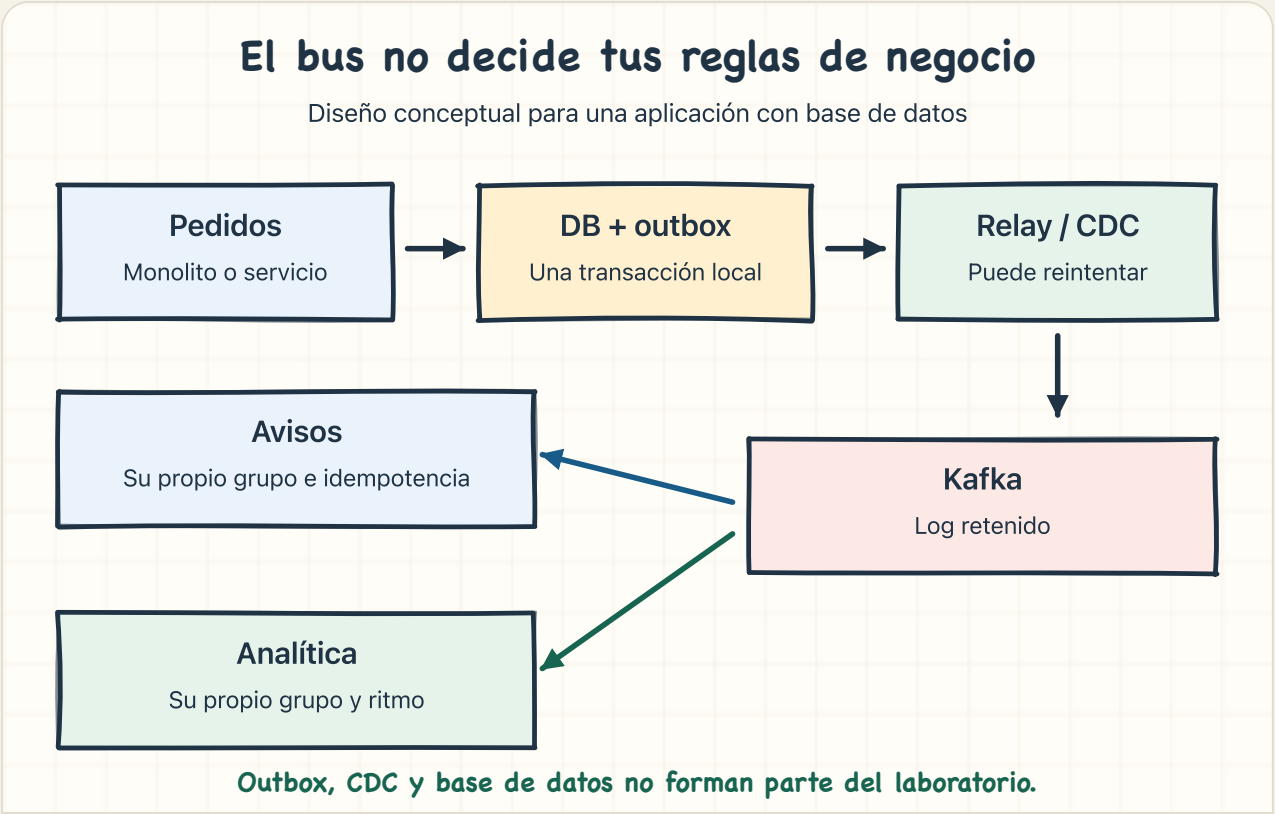
10. Del laboratorio a una decisión real

Dónde encaja en tu arquitectura

Un monolito puede ser dueño del proceso de pedidos y publicar hechos hacia aplicaciones externas. Una arquitectura de microservicios puede usar Kafka para cruzar fronteras entre contextos. Una plataforma de datos puede integrar fuentes y construir resultados continuos. Son usos diferentes que comparten el flujo retenido, no una obligación de dividir todo el sistema en servicios pequeños.

DDD puede ayudarte a nombrar hechos y definir qué contexto tiene autoridad sobre ellos. La arquitectura hexagonal permite dejar el cliente Kafka en un adaptador para que el dominio no dependa de sus clases.

CQRS separa modelos de lectura y escritura. Event Sourcing conserva cambios como fuente del estado. Ninguno aparece automáticamente al publicar un JSON y ninguno es requisito para ejecutar el broker.



Una posible evolución con base de datos. Outbox y CDC son decisiones adicionales; no forman parte del laboratorio.

El ecosistema: qué agregar según el trabajo

Nuestro laboratorio ya funciona con el broker y los clientes. Los complementos responden a necesidades concretas; instalarlos todos no convierte una aplicación en una mejor arquitectura.



Los clientes bastan para el ejemplo. Connect, Streams, CDC y un registro de esquemas cumplen trabajos diferentes.

Pieza	Trabajo concreto	Ejemplo de la tienda
Clientes y Spring Kafka	Publicar y consumir desde tu código	Nuestro endpoint y listener
Kafka Connect	Ejecutar conectores source —hacia Kafka— y sink —desde Kafka—	Llevar registros a un buscador sin escribir otro listener propio
Kafka Streams	Biblioteca Java de transformación y procesamiento con estado	Agrupar pedidos y contar ventas por tienda en ventanas de cinco minutos
CDC, por ejemplo Debezium	Capturar cambios desde el log de una base de datos	Publicar cambios de una tabla o transmitir un outbox
Registro de esquemas	Versionar contratos y verificar compatibilidad según sus reglas	Evitar retirar un campo que todavía necesitan lectores antiguos
Herramientas de operación	Explorar topics, medir salud y gestionar el clúster	Investigar lag, permisos o una partición sin líder

Connect tiene su propio runtime y tareas. Un conector source obtiene datos de un origen; un sink los entrega a otro sistema. **Streams** se incorpora a tu aplicación: no es otro broker ni un motor SQL que debas instalar para consumir. Sus ventanas, joins y almacenes de estado resuelven cálculos que un simple

`println` no necesita. Ambos pertenecen al proyecto Apache Kafka; sus documentos de integración y de procesamiento explican sus límites.

CDC significa **Change Data Capture**. Capturar "cambió una fila" no define automáticamente un hecho de dominio como `PedidoCreado`: necesitas un contrato y decidir qué cambio lo representa. Un outbox expresa esa intención dentro de la transacción de negocio; CDC puede transportarla. Debezium documenta esa captura de cambios como proyecto separado. También lo son los productos de registro de esquemas y muchas interfaces de administración. Kafka no valida por sí solo el JSON de nuestro ejemplo contra un esquema de negocio: esa responsabilidad vive en clientes, herramientas y políticas acordadas.

Contratos que sobreviven al siguiente despliegue

Publicar un cambio obliga a pensar en consumidores que no se actualizan al mismo tiempo. Agregar un campo opcional y renombrar uno obligatorio no tienen el mismo impacto. Cambiar el significado de `creadoEn` puede romper un cálculo aunque el JSON siga siendo válido. La compatibilidad tiene una dimensión estructural y otra semántica.

Define quién es dueño del topic y cómo anuncia cambios. Documenta ejemplos válidos, valores permitidos, identidad, tiempo y datos personales. Un schema registry puede automatizar ciertas comprobaciones y apoyar formatos como Avro, Protobuf o JSON Schema; no es parte obligatoria del broker ni sustituye la conversación entre equipos. Las reglas de compatibilidad de Confluent Schema Registry son un ejemplo concreto de esas políticas externas. Para empezar, un contrato pequeño y versionado con pruebas de consumidores ya obliga a explicitar muchas decisiones útiles.

Revisa también la vida de los lectores antiguos. Un replay puede encontrar registros escritos por varias versiones del productor. Si tu deserializador solo entiende el último formato, quizá el flujo en vivo funcione y la reconstrucción falle al tercer día de historia. Es una buena razón para probar con eventos antiguos de verdad, no solo con el ejemplo feliz que sale del despliegue más reciente.

Seguridad, capacidad y señales operativas

El Compose usa PLAINTEXT en una interfaz local. Un entorno real necesita decidir cifrado, autenticación y autorización, además de la red que permite llegar a los listeners. La guía de seguridad y las ACLs son referencias distintas: conectarse no implica tener permiso para leer cualquier topic o usar cualquier grupo.

Para estimar almacenamiento, comienza con bytes por segundo multiplicados por la ventana de retención y por el número de réplicas. Después incorpora compresión, índices, metadatos, márgenes y comportamiento de limpieza. Es una estimación inicial, no una reserva exacta. Para estimar recuperación, calcula si el lector puede procesar más rápido que la entrada mientras recupera atraso; de lo contrario nunca alcanzará la punta.

Observa latencia y errores de publicación, atraso por grupo y partición, velocidad de procesamiento, tiempo del evento más antiguo pendiente, rebalances, disco y salud de réplicas. El catálogo de métricas ayuda a elegir señales concretas. Una alerta de lag sin contexto de carga puede resultar poco útil; una

alerta basada en la demora que incumple la promesa de tu producto conecta mejor con la decisión de actuar.

Cinco ejercicios para comprobar tu modelo

1. Dos lectores. Avisos y analítica deben recibir todas las compras. ¿Usarías el mismo `group.id`? Explica qué pasaría si lo hicieras.

> – **Solución**
>
> usaría grupos diferentes. Un mismo grupo reparte particiones y trabajo entre sus miembros; no ofrece una copia de todos los registros a cada aplicación. Con grupos separados, cada propósito conserva su progreso y puede reconstruirse sin mover el del otro.

2. El rebobinado que no ocurrió. Reinicias un grupo con `earliest` y no vuelve a leer lo anterior. ¿Qué revisarías antes de culpar a la retención?

> – **Solución**
>
> revisaría si el grupo ya tiene offsets confirmados válidos. `earliest` no los reinicia. Para estudiar replay puedo crear otro grupo; para modificar el progreso de uno real necesito un procedimiento administrativo deliberado y consumidores detenidos cuando corresponda.

3. El correo duplicado. El correo salió, el proceso terminó antes del commit y volvió a salir al reiniciar. ¿Qué promesa faltaba diseñar?

> – **Solución**
>
> faltaba proteger el efecto frente a reentregas en el límite del correo, mediante identidad estable y una estrategia que coordine deduplicación y efecto según las capacidades del destino. La idempotencia del productor y el commit posterior no hacen atómico ese efecto externo.

4. El cuarto consumidor. Tienes tres particiones, cuatro miembros del mismo grupo y un solo topic. ¿El cuarto proceso garantiza más capacidad?

> – **Solución**

no. En el modelo convencional del laboratorio solo hay tres particiones que repartir. Además, si casi toda la carga está en una sola clave, puede concentrarse en una sola partición. Mediría distribución y tiempo de procesamiento antes de agregar procesos.

5. El pedido sin evento. Una aplicación guarda el pedido y termina antes de publicar. ¿Lo arregla publicar primero?

> – **Solución**
>
> cambia la ventana de fallo, pero no la elimina. Consideraría un outbox junto al cambio de negocio en una transacción local y un relay recuperable. Mantendría idempotencia en los lectores porque la publicación del outbox puede repetirse.

Glosario para volver cuando lo necesites

Término	La pregunta que responde
Evento	¿Qué hecho se está comunicando?
Topic	¿A qué flujo pertenece?
Clave	¿Cómo relacionamos y particionamos sus registros?
Partición	¿En qué secuencia ordenada se guarda?
Offset	¿En qué posición de esa partición está?
Broker	¿Qué proceso sirve y almacena las particiones?
Controlador KRaft	¿Quién coordina los metadatos del clúster?
Grupo	¿Qué lectores comparten reparto y progreso?
Commit	¿Desde dónde debe continuar el grupo?
Retención	¿Qué historia permanece disponible según la política?
Replay	¿Qué registros disponibles queremos recorrer de nuevo?
Lag	¿Qué distancia hay entre progreso y extremo del log?
ISR	¿Qué réplicas se consideran sincronizadas?
Idempotencia	¿Qué impide multiplicar un efecto al repetirlo?

La conversación que llevaría al equipo

Pregunta	Si no tienes respuesta, qué falta diseñar
¿Necesitamos varios lectores y replay?	El motivo para elegir un log retenido
¿Qué orden exige el negocio?	La clave, el productor y la concurrencia del lector
¿Cuánto puede atrasarse un grupo?	Retención, capacidad y alertas
¿Qué sucede con duplicados?	Identidad y protección de efectos
¿Quién mantiene el contrato?	Responsabilidad y evolución
¿Quién recupera el flujo?	Operación y procedimiento de incidentes

Si puedes responderlas, la discusión sobre Kafka ya tiene una base mucho más útil que una lista de tecnologías populares. Puedes adoptar la herramienta, probar una alternativa o posponerla con una razón concreta. Ese criterio también es parte de aprenderla.

Ahora vuelve a tu aplicación: ¿qué hecho necesitas compartir, quién debería poder volver a leerlo y qué decisión tomarías hoy con esas dos respuestas?