El patrón **SAGA** es una técnica utilizada en sistemas distribuidos, particularmente en arquitecturas de microservicios, para gestionar transacciones distribuidas y asegurar consistencia sin necesidad de usar transacciones distribuidas ACID. En el patrón SAGA, una transacción larga se descompone en una serie de transacciones locales, cada una gestionada por un servicio independiente, con compensaciones en caso de fallos.

1. SAGA Coreografiada (Choreography-Based Saga)

- **Cómo funciona**: En esta implementación, los servicios se comunican de manera asíncrona mediante eventos. Cada servicio realiza su propia operación local y, si es exitosa, publica un evento que activa la siguiente operación en otro servicio. Si ocurre un error, los servicios compensan sus acciones publicando eventos de deshacer.
- **Ventajas**:
    - Es sencilla de implementar, especialmente en sistemas donde los servicios ya se comunican mediante eventos.
    - Elimina la necesidad de un coordinador central, lo que reduce el acoplamiento.
- **Desafíos**:
    - A medida que aumenta el número de servicios, la complejidad de coordinar los eventos puede volverse difícil de gestionar.
    - Dificulta la trazabilidad, ya que no hay un flujo centralizado que controle toda la transacción.
    - Las reglas de negocio distribuidas se implementan en cada servicio, lo que puede complicar los cambios futuros.

2. SAGA Orquestada (Orchestration-Based Saga)

- **Cómo funciona**: En este enfoque, hay un **orquestador** central (un servicio dedicado) que coordina la secuencia de las transacciones. El orquestador envía comandos a los servicios involucrados, y cada servicio realiza su operación local. Si algo falla, el orquestador inicia las transacciones compensatorias necesarias para deshacer los pasos anteriores.
- **Ventajas**:
    - Centraliza la lógica de coordinación y las reglas de negocio, lo que facilita el mantenimiento y la trazabilidad.
    - Permite un mayor control sobre el flujo y el orden de las transacciones.
- **Desafíos**:
    - Introduce un punto central de control que puede ser un cuello de botella o un punto único de fallo.
    - Mayor complejidad en la implementación del orquestador.

3. SAGA Basada en Compensaciones

- **Cómo funciona**: En este enfoque, cada paso de la transacción SAGA tiene una operación de compensación asociada que se ejecuta si falla alguna de las operaciones posteriores. Las compensaciones se encargan de deshacer cualquier

cambio realizado por los pasos anteriores. Esto es útil cuando no es posible o deseable garantizar consistencia inmediata entre servicios.

- **Ventajas**:
  - Asegura que cualquier fallo se pueda manejar mediante la reversión de los cambios.
  - Flexible para casos donde es aceptable tener inconsistencia temporal.
- **Desafíos**:
  - Las operaciones de compensación pueden ser complicadas de diseñar y deben ser idempotentes (capaces de ejecutarse varias veces sin efectos negativos).
  - Mayor complejidad debido a la necesidad de manejar tanto la lógica de las transacciones como la de las compensaciones.

4. SAGA Basada en Tiempo (Time-Based Saga)

- **Cómo funciona**: En este enfoque, cada paso de la SAGA tiene un límite de tiempo para completarse. Si el servicio no responde dentro de un tiempo determinado, se considera que ha fallado, y se inician automáticamente las compensaciones o se toma una acción predefinida. Es útil para sistemas en los que la disponibilidad es crítica y no se puede esperar indefinidamente por una respuesta.
- **Ventajas**:
  - Maneja bien los tiempos de espera en sistemas distribuidos donde los fallos de comunicación o respuestas lentas son frecuentes.
- **Desafíos**:
  - Debe definirse con precisión el tiempo límite, y puede ser difícil balancear entre ser demasiado conservador o permisivo.
  - Puede provocar compensaciones innecesarias si un servicio simplemente tarda más en responder, pero no ha fallado realmente.

5. SAGA Eventual (Eventual Saga)

- **Cómo funciona**: La SAGA eventual se basa en la consistencia eventual. Los servicios completan sus operaciones de forma independiente y, si ocurre un fallo, pueden desencadenar las operaciones compensatorias en segundo plano hasta que el sistema vuelva a un estado consistente. No garantiza la consistencia inmediata entre servicios, pero asegura que eventualmente los sistemas estarán sincronizados.
- **Ventajas**:
  - Excelente para aplicaciones que no requieren consistencia fuerte e inmediata, como sistemas de e-commerce o procesamiento de pedidos.
- **Desafíos**:
  - Implica que los sistemas deben aceptar estados temporales de inconsistencia.
  - Se deben diseñar cuidadosamente las reglas de compensación y las pruebas de resiliencia.

6. SAGA Multietapa (Multi-Step Saga)

- **Cómo funciona**: Una SAGA multietapa se implementa en situaciones donde las transacciones distribuidas implican una gran cantidad de pasos que deben ser gestionados en paralelo. Se divide la SAGA en múltiples sub-SAGAs que pueden ejecutarse de manera más eficiente, lo que permite controlar diferentes flujos en paralelo y mejorar el rendimiento.
- **Ventajas**:
    - Mejora el rendimiento cuando se necesitan múltiples pasos distribuidos.
    - Permite manejar transacciones más complejas de manera organizada.
- **Desafíos**:
    - Mayor complejidad en la coordinación y orquestación de sub-SAGAs.
    - La coordinación de sub-SAGAs puede introducir latencia adicional y ser difícil de depurar.

7. SAGA con Confirmación Explícita (Explicit Acknowledgment Saga)

- **Cómo funciona**: En este enfoque, los servicios deben confirmar explícitamente la finalización exitosa de cada paso antes de que la SAGA continúe con el siguiente. Si un servicio no confirma en el tiempo establecido o responde con un fallo, se inician las compensaciones.
- **Ventajas**:
    - Garantiza un nivel más alto de control y confiabilidad en cada paso de la SAGA.
- **Desafíos**:
    - Puede introducir una mayor latencia, ya que se necesita confirmación explícita de cada paso.
    - Aumenta la complejidad en la gestión de los tiempos de espera y las confirmaciones.

8. SAGA Reactiva (Reactive Saga)

- **Cómo funciona**: En una SAGA reactiva, los servicios utilizan programación reactiva para manejar los pasos de la SAGA de manera no bloqueante. Esto permite que los servicios reaccionen a eventos en lugar de estar esperando activamente la finalización de una operación. La programación reactiva puede mejorar la escalabilidad en sistemas distribuidos donde hay una alta concurrencia.
- **Ventajas**:
    - Mejor manejo de la concurrencia y optimización de recursos en sistemas distribuidos.
    - Mejor rendimiento en sistemas que necesitan manejar grandes volúmenes de operaciones en paralelo.
- **Desafíos**:
    - Mayor complejidad en el diseño e implementación de soluciones reactivas.
    - Requiere el uso de tecnologías y herramientas específicas para programación reactiva, lo que puede incrementar la curva de aprendizaje.

1. SAGA Coreografiada (Choreography-Based Saga)

Ejemplo del mundo real:
Imagina un sistema de reservas de viajes donde cada servicio (reservas de vuelos, hoteles y alquiler de coches) opera de forma independiente. Si se reserva un vuelo, se genera un evento. Este evento activa la reserva de hotel, que a su vez activa el alquiler de coche. Si hay un problema al reservar el coche, cada servicio ejecuta una compensación (cancelar vuelo y hotel).

Tecnologías:

- **Kafka** o **RabbitMQ** para la mensajería de eventos.
- **Spring Cloud Stream** para integración de microservicios con eventos.
- **Axon Framework** para manejar eventos y acciones compensatorias.

2. SAGA Orquestada (Orchestration-Based Saga)

Ejemplo del mundo real:
Un proceso de gestión de órdenes en un ecommerce. Un servicio orquestador gestiona la creación de una orden, la reserva de inventario y el procesamiento del pago. Si el pago falla, el orquestador envía comandos para deshacer la reserva de inventario y cancelar la orden.

Tecnologías:

- **Camunda** o **Netflix Conductor** para la orquestación.
- **Spring Boot** con un coordinador de transacciones.
- **Kafka** para mensajería en el caso de eventos.

3. SAGA Basada en Compensaciones

Ejemplo del mundo real:
Un sistema de facturación en el que una vez realizada la facturación, se puede iniciar el envío del producto. Si hay un error en el envío, se aplica una compensación que deshace la facturación o emite un reembolso.

Tecnologías:

- **Axon Framework** para la gestión de transacciones y compensaciones.
- **Eventuate** para manejar compensaciones en eventos.
- **Kafka** o **RabbitMQ** para la comunicación entre microservicios.

4. SAGA Basada en Tiempo (Time-Based Saga)

Ejemplo del mundo real:
En un sistema de subastas online, cada oferta tiene un límite de tiempo. Si un usuario no recibe confirmación de la puja dentro del tiempo determinado, se asume que la puja falló y se ejecutan las acciones compensatorias, como liberar el crédito retenido.

Tecnologías:

- **Akka** o **Vert.x** para manejo de tiempo y tareas programadas.
- **Quartz Scheduler** para gestión de tiempos de espera.
- **Kafka Streams** o **RabbitMQ** para el procesamiento de eventos.

5. SAGA Eventual (Eventual Saga)

Ejemplo del mundo real:
En un sistema de gestión de inventarios, si una orden es colocada, el stock puede no ser actualizado de inmediato. Las actualizaciones de inventario y órdenes se sincronizan eventualmente, y si una falla ocurre, las operaciones compensatorias se aplican en segundo plano.

Tecnologías:

- **Cassandra** o **MongoDB** para bases de datos eventuales.
- **Kafka Streams** para procesar flujos de datos y asegurar consistencia eventual.
- **Event Sourcing** con **Axon** para manejar inconsistencias temporales.

6. SAGA Multietapa (Multi-Step Saga)

Ejemplo del mundo real:
En una empresa de logística global, los pedidos de envío se gestionan en múltiples etapas: procesamiento de la orden, reserva de transporte y seguimiento. Estas etapas pueden ejecutarse en paralelo, mejorando la eficiencia y reduciendo tiempos de espera.

Tecnologías:

- **Netflix Conductor** o **Camunda** para orquestar las sub-SAGAs.
- **Apache Flink** para la coordinación y procesamiento de eventos en tiempo real.
- **Spring Boot** con microservicios desacoplados para ejecutar cada subproceso.

7. SAGA con Confirmación Explícita (Explicit Acknowledgment Saga)

Ejemplo del mundo real:
En una empresa de comercio electrónico, una orden de compra solo avanza si cada servicio (pago, inventario, y envío) confirma explícitamente el éxito de sus operaciones. Si alguno no confirma, el sistema cancela la transacción y deshace las operaciones realizadas hasta ese punto.

Tecnologías:

- **Kafka** o **RabbitMQ** con confirmaciones de mensaje explícitas.
- **Spring Cloud Stream** para gestión de eventos con confirmaciones.
- **Eventuate** para manejar eventos y confirmaciones de servicios.

8. SAGA Reactiva (Reactive Saga)

Ejemplo del mundo real:
Una aplicación bancaria en la que los servicios como creación de cuentas, aprobación de créditos y procesamiento de transacciones se gestionan de forma reactiva, permitiendo una alta concurrencia y respuestas rápidas a los clientes.

Tecnologías:

- **Project Reactor** o **RxJava** para programación reactiva.
- **WebFlux** en Spring Boot para manejar servicios reactivos.
- **Kafka Streams** para procesamiento reactivo de eventos.

Kafka

Una plataforma de mensajería distribuida que permite publicar, almacenar, y procesar flujos de datos en tiempo real.

RabbitMQ

Un broker de mensajes que facilita la comunicación asíncrona entre servicios mediante la cola y enrutamiento de mensajes.

Spring Cloud Stream

Un framework de Spring que simplifica la integración de microservicios con plataformas de mensajería como Kafka o RabbitMQ.

Axon Framework

Un framework especializado en la implementación de patrones de Event Sourcing y CQRS, ideal para manejar eventos y acciones compensatorias.

Eventuate

Una plataforma que soporta el desarrollo de microservicios transaccionales mediante Event Sourcing, manejo de eventos y patrones como SAGA.

Camunda

Una herramienta de orquestación de flujos de trabajo y gestión de procesos de negocio que permite coordinar tareas entre servicios.

Netflix Conductor

Un motor de orquestación de microservicios que facilita la gestión de flujos complejos distribuidos.

Akka

Un toolkit para construir sistemas distribuidos concurrentes y escalables basados en el modelo de actores.

Vert.x

Un framework asíncrono y basado en eventos que facilita la creación de aplicaciones distribuidas y de alta concurrencia.

Quartz Scheduler

Un framework de planificación de tareas que permite programar y gestionar tareas en sistemas distribuidos.

Kafka Streams

Una biblioteca para construir aplicaciones y microservicios que procesan flujos de datos de Kafka en tiempo real.

Cassandra

Una base de datos NoSQL distribuida que proporciona alta disponibilidad y consistencia eventual para grandes volúmenes de datos.

MongoDB

Una base de datos NoSQL orientada a documentos, diseñada para manejar grandes cantidades de datos y ofrecer escalabilidad.

Apache Flink

Un motor de procesamiento de flujos de datos en tiempo real que soporta el procesamiento distribuido y en paralelo.

Project Reactor

Una librería para programación reactiva que permite construir aplicaciones no bloqueantes y de alta concurrencia en Java.

RxJava

Un framework para programación reactiva que permite manejar flujos de datos asíncronos de manera eficiente.

WebFlux

Parte de Spring que soporta aplicaciones web no bloqueantes utilizando programación reactiva con Reactor.

Event Sourcing

Un patrón arquitectónico en el que el estado de un sistema se deriva de una secuencia de eventos en lugar de un estado persistente.