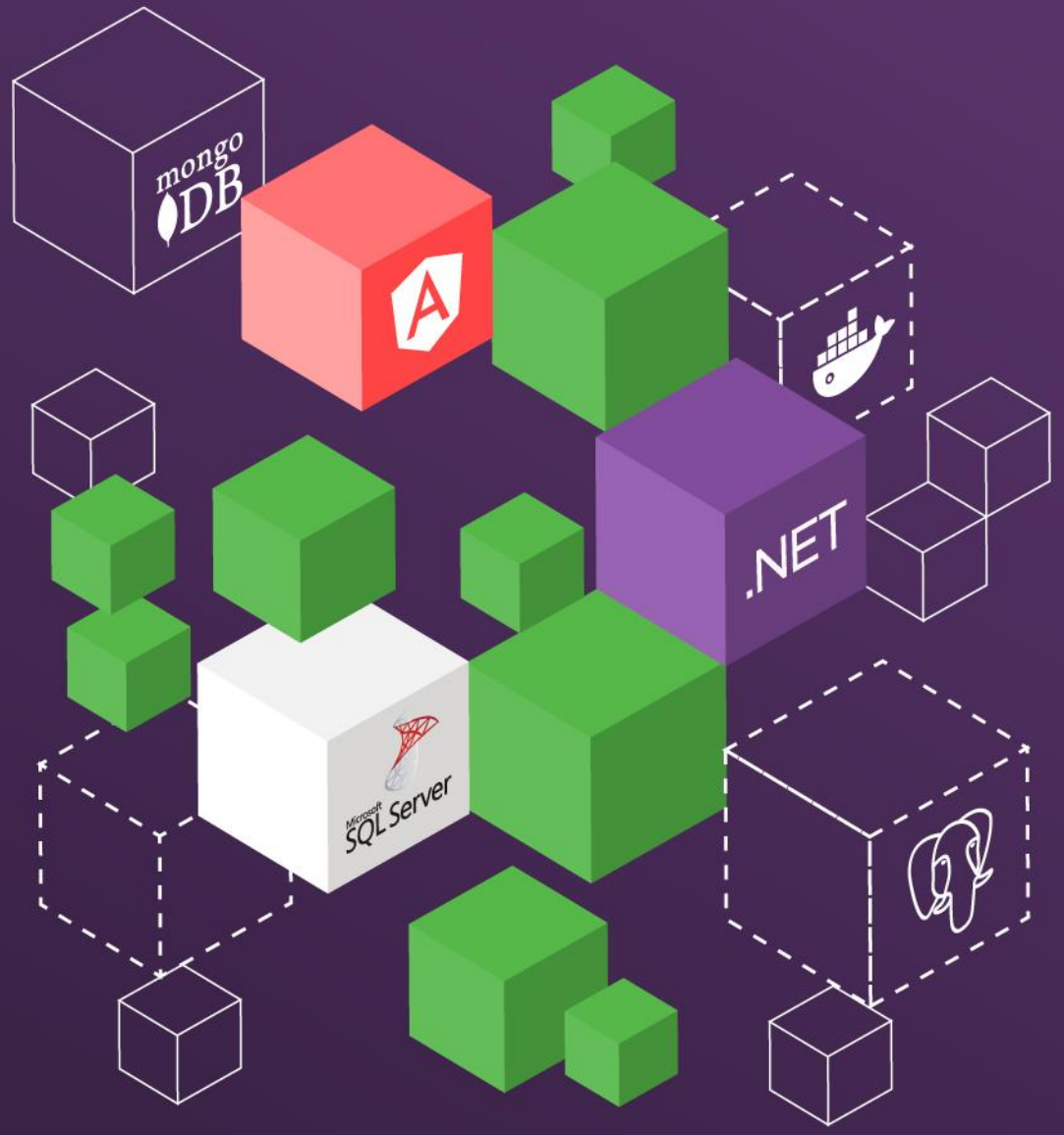


BIENVENIDOS
AL CURSO:

Arquitectura de Microservicios en Net

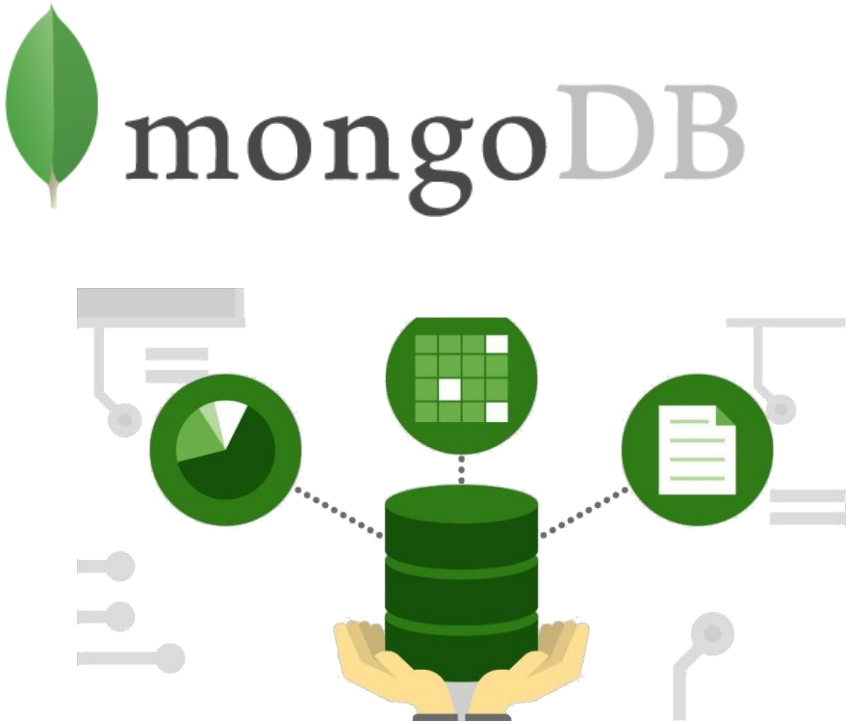
SESIÓN 04

GALAXY
TRAINING

ÍNDICE

Infraestructura de persistencia – NoSQL (CosmoDB).

MongoDB



- MongoDB es una base de datos de documentos que ofrece una gran escalabilidad y flexibilidad, y un modelo de consultas e indexación avanzado.
- MongoDB almacena datos en documentos flexibles similares a JSON, por lo que los campos pueden variar entre documentos y la estructura de datos puede cambiarse con el tiempo
- El modelo de documento se asigna a los objetos en el código de su aplicación para facilitar el trabajo con los datos
- MongoDB es una base de datos distribuida en su núcleo

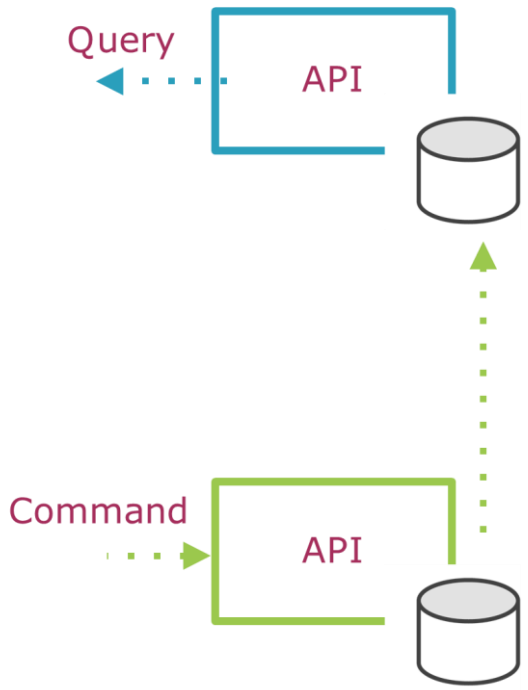
MongoDB / Cosmo DB

- Azure Cosmos DB es un servicio de base de datos con varios modelos distribuido de forma global de Microsoft. Con tan solo un clic, Cosmos DB permite escalar de forma elástica e individual el rendimiento y el almacenamiento en cualquier número de regiones de Azure a nivel mundial.
- Puede escalar de forma elástica el rendimiento y almacenamiento, y sacar provecho del rápido acceso a datos (menos de 10 milisegundos) mediante la API que prefiera, entre las que se incluyen SQL, MongoDB, Cassandra, Tables o Gremlin

Infraestructura de persistencia - Entity Framework Core, MSSQL, NoSQL (CosmoDB)

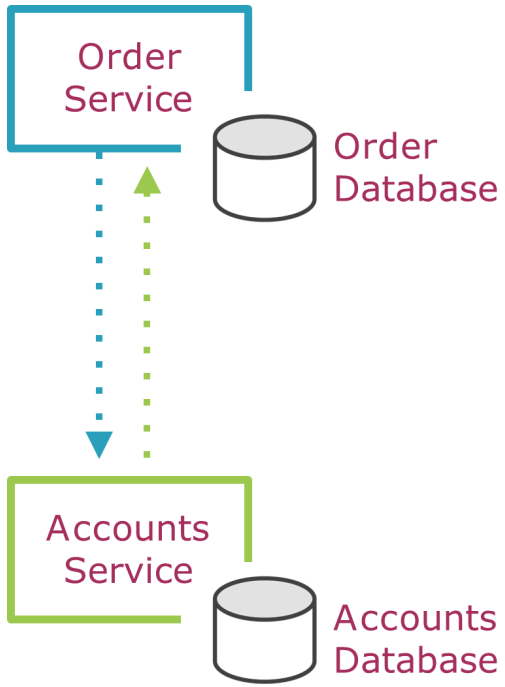
Implementando el patrón CQRS a un microservicio DDD.

Command Query Response Segregation



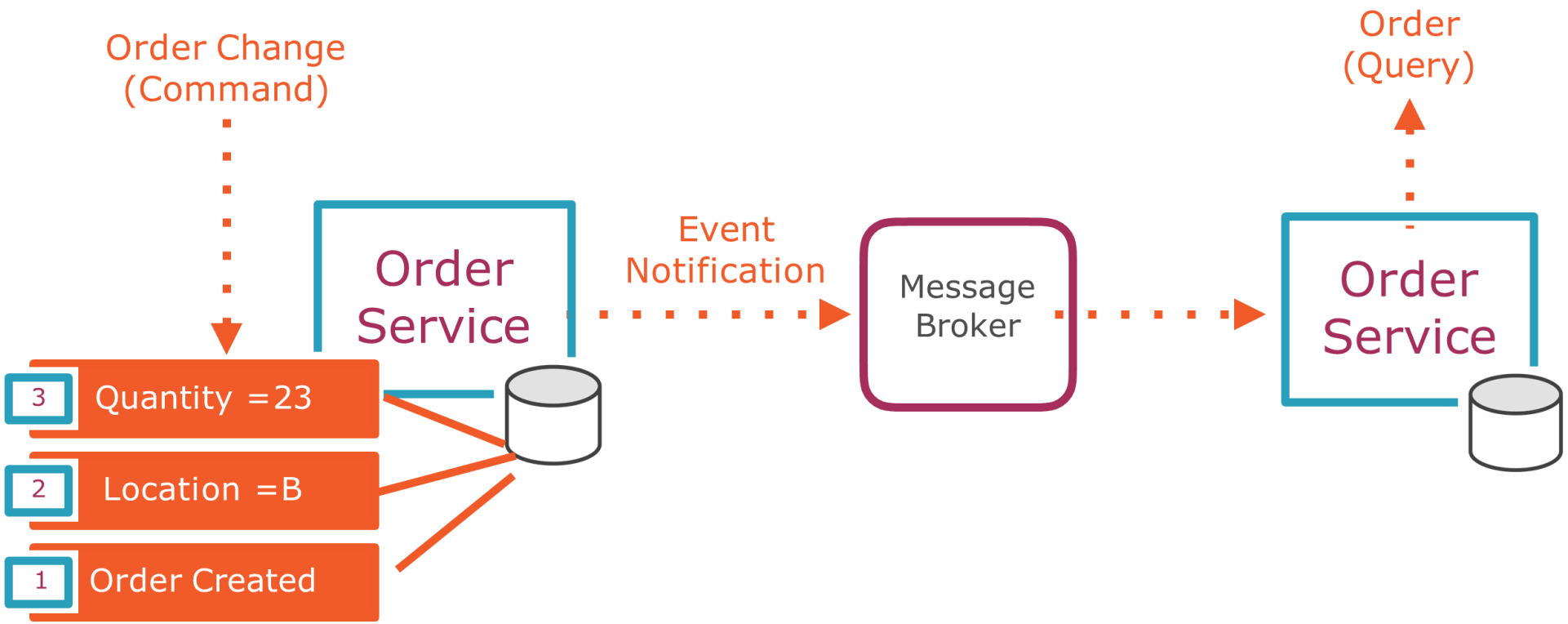
- CQRS
  - Modelos de comando (**Command**) y/o servicios
  - Modelos de consulta (**Query**) y/o servicios
- ¿Por qué?
  - Separación de responsabilidades.
    - Notificaciones de eventos manejadas por comando (**Escritura**)
    - Informes/funciones manejadas por consulta (**Lectura**)
  - Separación de tecnologías.
    - Servicio y almacenamiento
- Desafíos
  - Comando y consulta de sincronización de bases de datos

CQRS ¿Cómo?

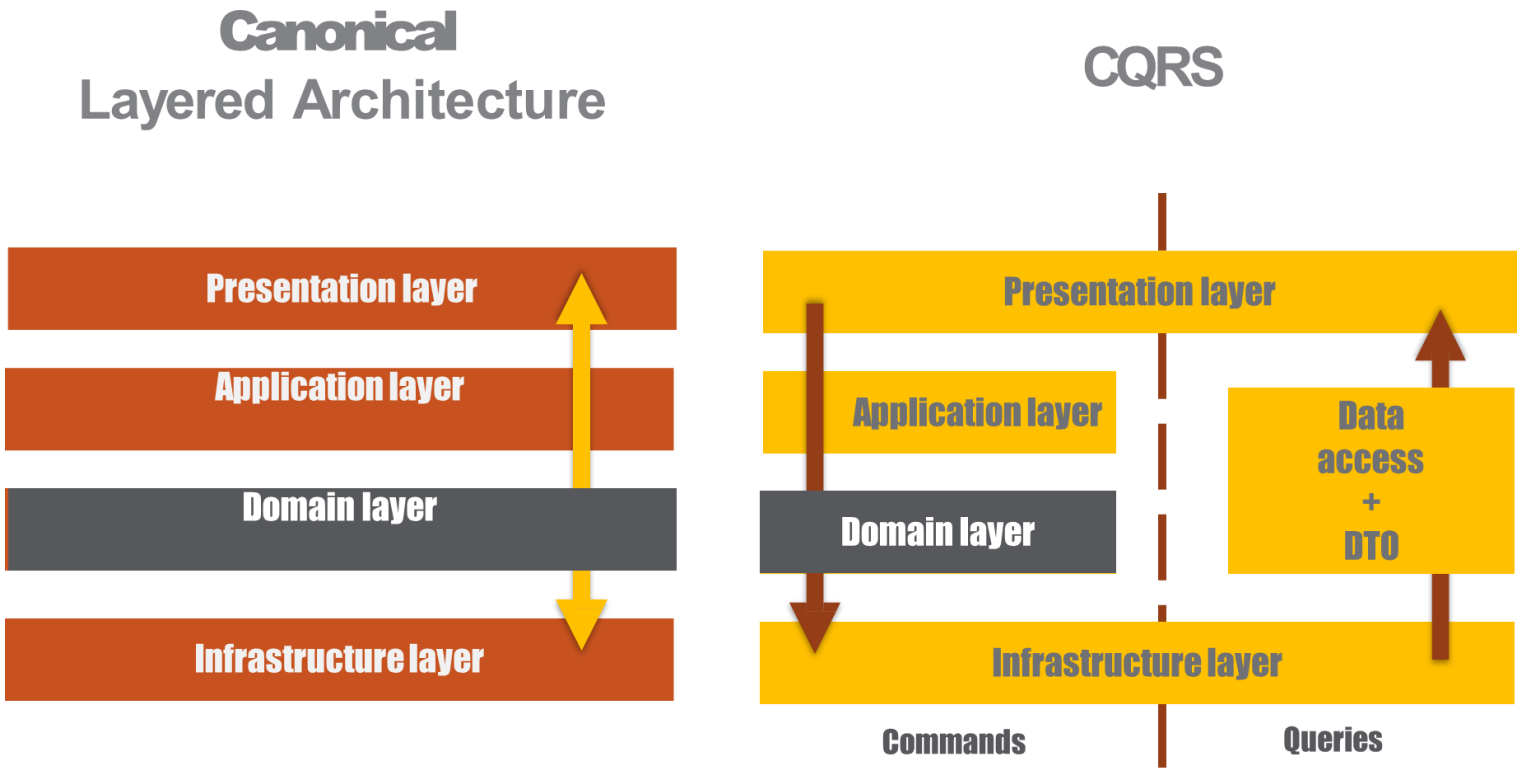


- Command microservice, recibe eventos
  - Se suscribe a una cola
  - Múltiples microservicios de suscripción
  - Uso de modelo de comandos y eventos del repositorio.
- Query microservices, para datos
  - En forma de una función
  - En forma de recuperación de datos
- Proceso para sincronizar el repositorio de comandos con el repositorio de consultas.
  - Podría recibir la misma notificación de evento
  - Podría hacerse a nivel de tecnologías de base de datos.

CQRS

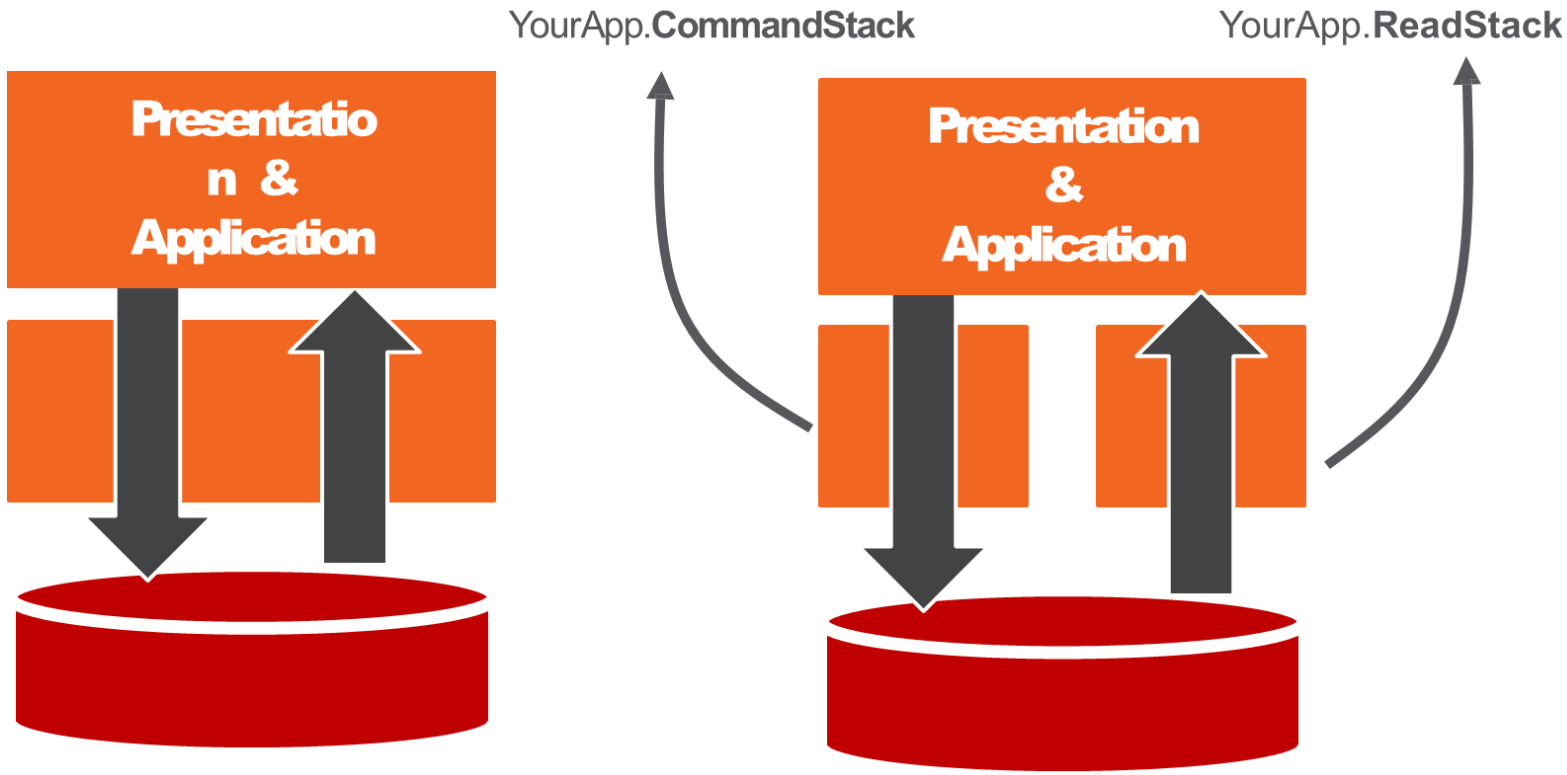

Order Change
(Command)
Order
Service
3
Quantity =23
2
Location =B
1
Order Created
Event
Notification
Message
Broker
Order
(Query)
Order
Service

CQRS

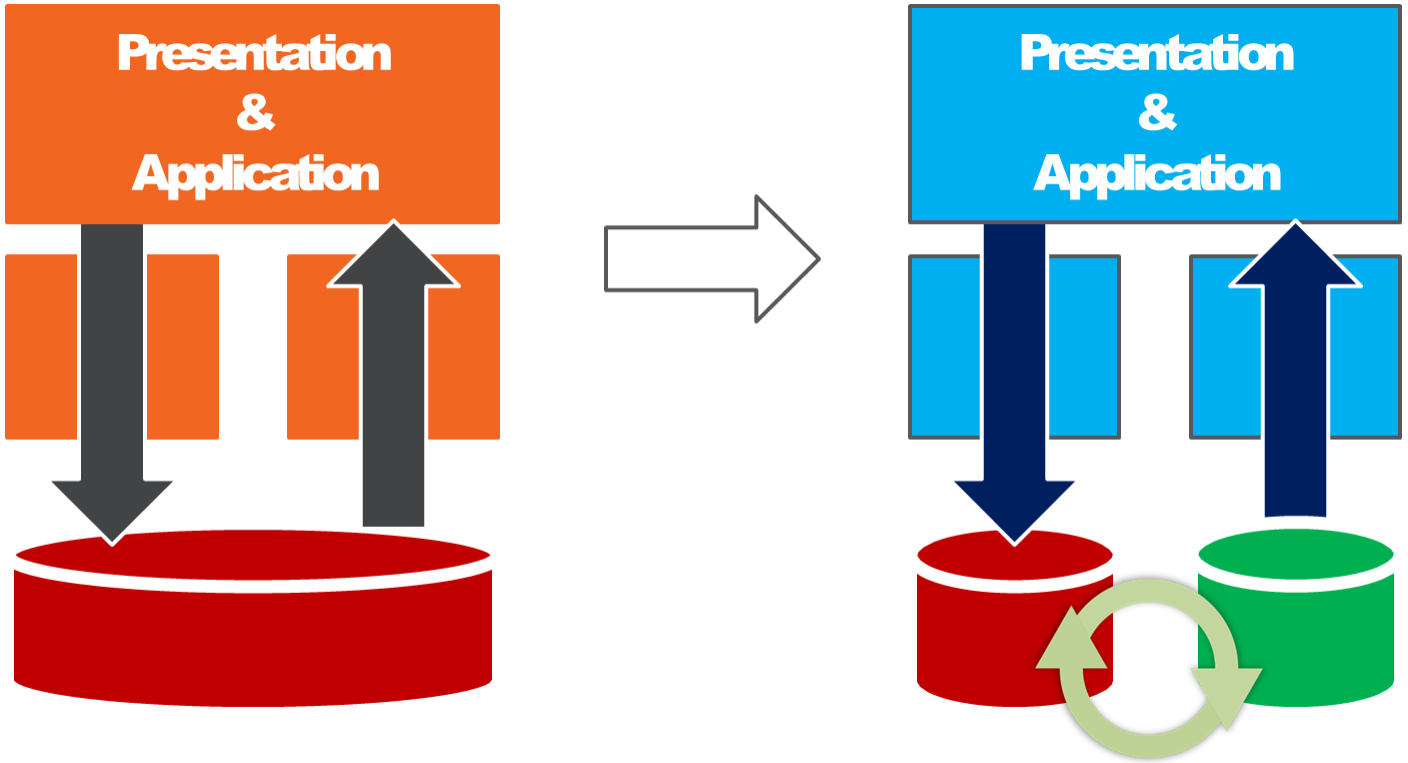

Canonical
Layered Architecture
Presentation layer
Application layer
Domain layer
Infrastructure layer
CQRS
Presentation layer
Application layer
Domain layer
Data
access
+
DTO
Infrastructure layer
Commands
Queries

Sabores de CQRS

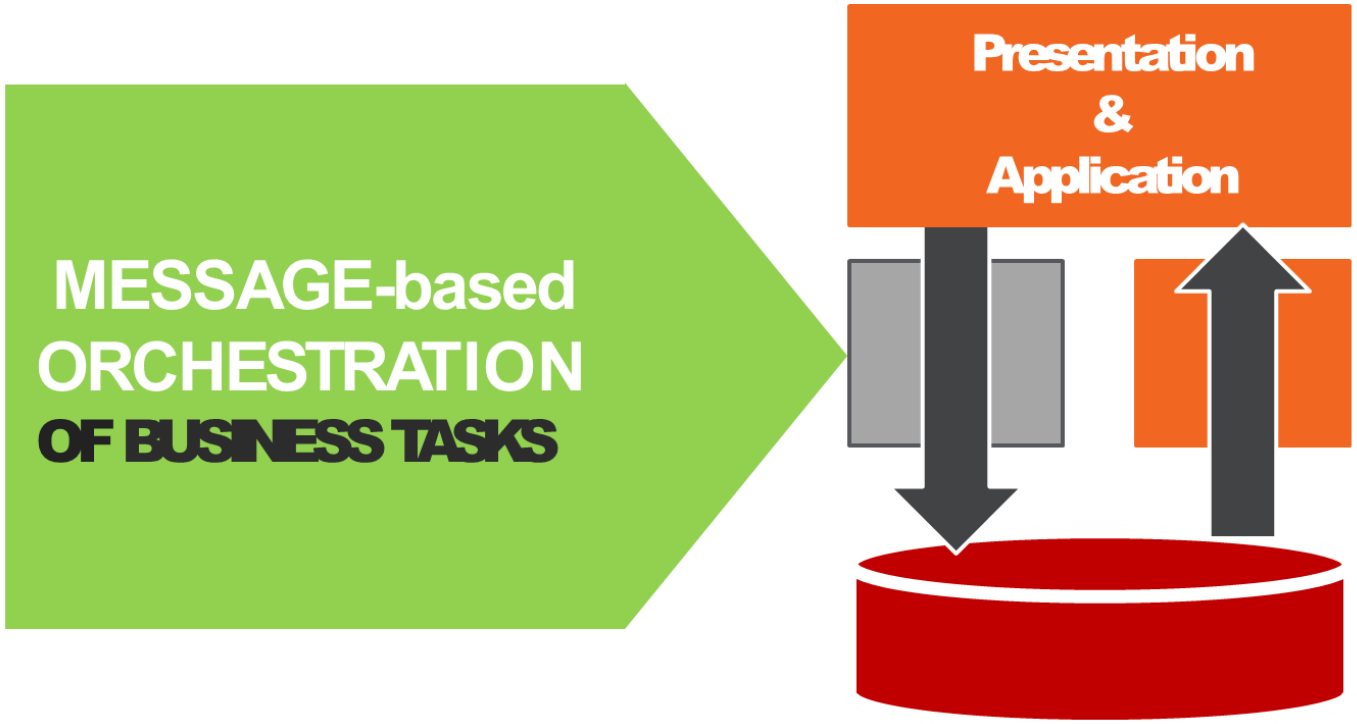
CQRS para aplicaciones CRUD simples

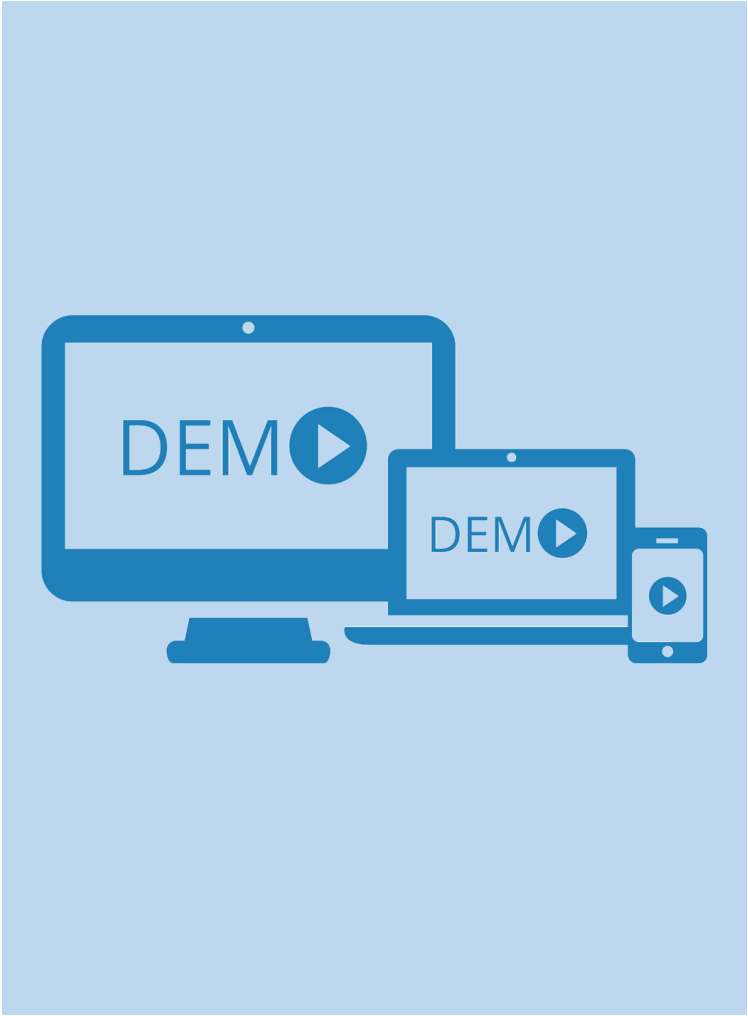


CQRS Premium



CQRS Deluxe


MESSAGE-based
ORCHESTRATION
OF BUSINESS TASKS
Presentation
&
Application



CQRS

Inyección de Dependencias (DI .Net Core).

Un conjunto de principios y patrones de diseño de software que nos permiten desarrollar código débilmente acoplado.

Van Deursen and Seeman. Dependency Injection in .NET. Manning, 2018.

Beneficios del débil acoplamiento

- Fácil de extender
- Fácil de probar
- Fácil de mantener
- Facilita el desarrollo paralelo.
- Facilita el “Late Binding”

Tiempos de vida del servicio

Transient

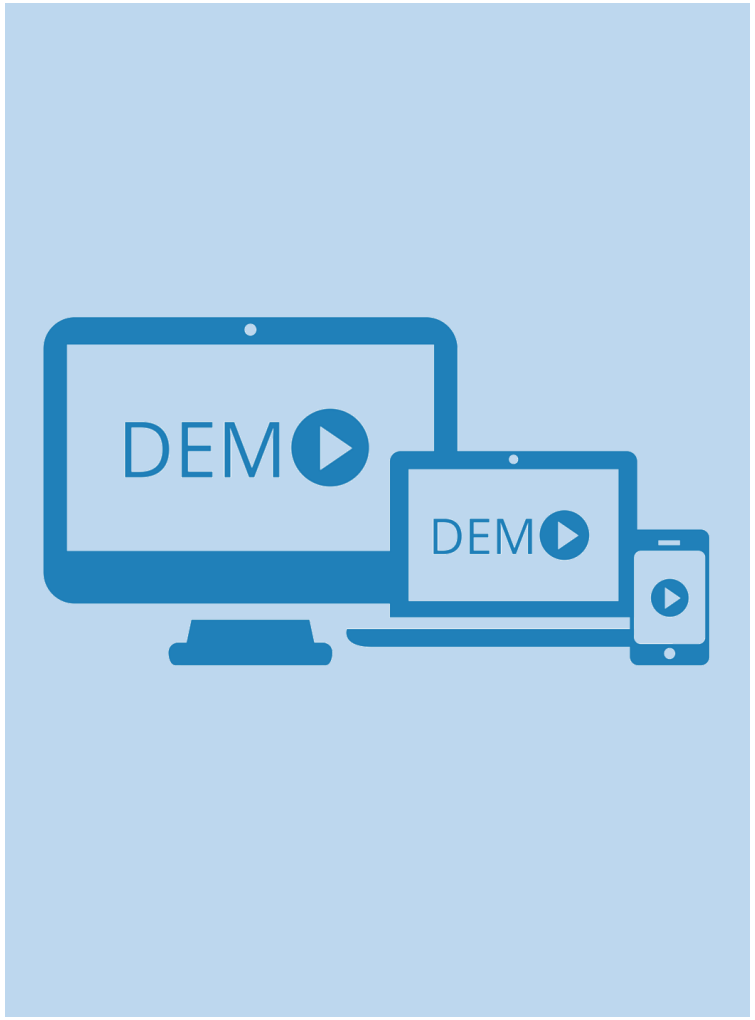
Creado cada vez que es solicitado

Singleton

Creado una vez durante la vida útil de la aplicación.

Scoped

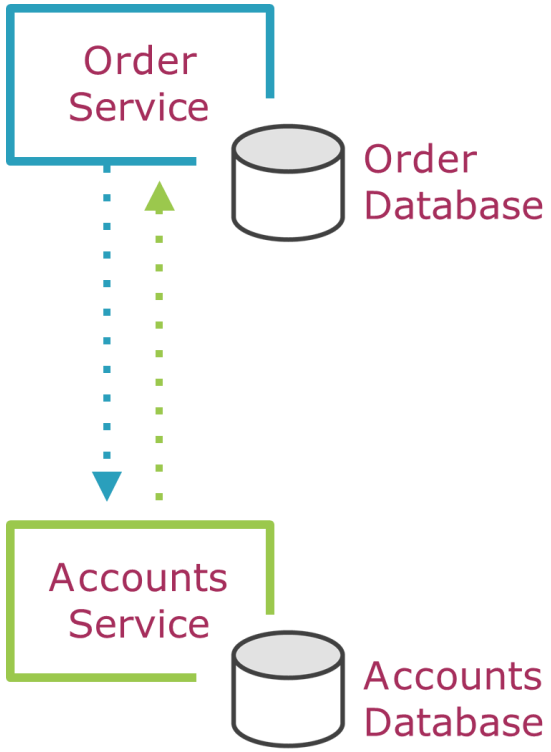
Creado una vez durante la vida del request



DI

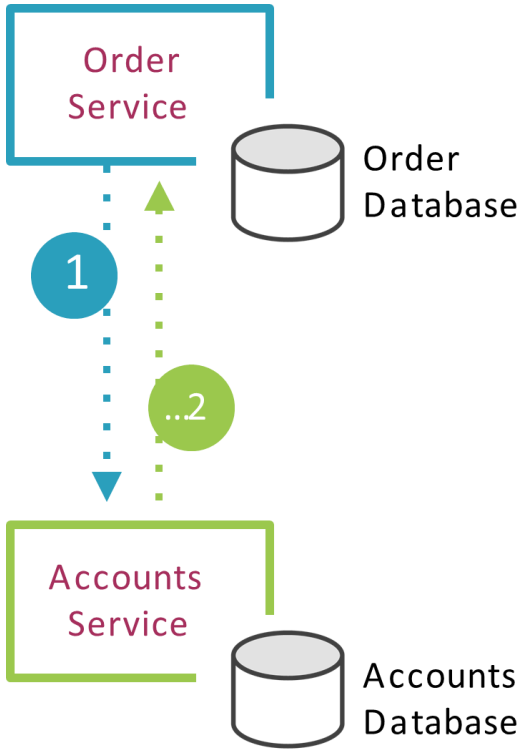
Cómo lograr la consistencia de datos a través de microservicios (consistencia eventual).

Cómo diseñar microservicios basados en API



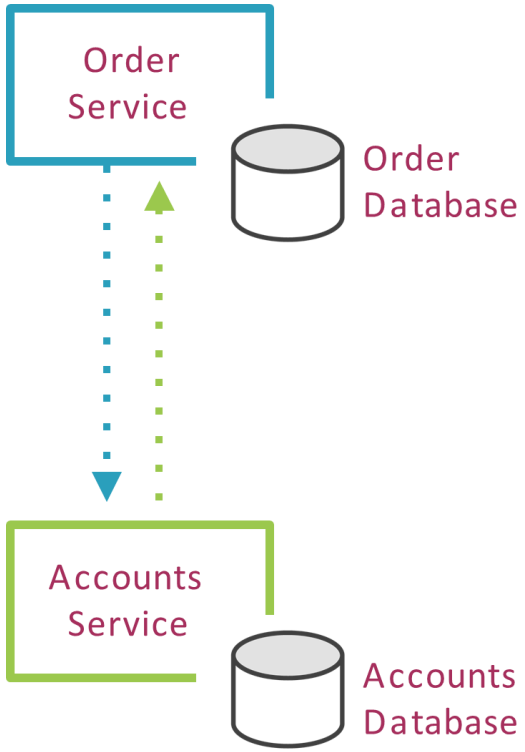
- Microservicios
  - API vs worker
- Arquitectura
  - API vs aplicación
- Como arquitectura
  - Requerimientos funcionales
  - Estilos de arquitectura
  - Patrones de arquitectura

Requerimientos Funcionales



- Principio de microservicios autónomos
  - Débilmente acoplado
  - Independientemente cambiable
  - Desplegable independientemente
  - Contratos de apoyo y honor
  - API agnóstica a la tecnología
  - API sin estado

Opciones de arquitectura



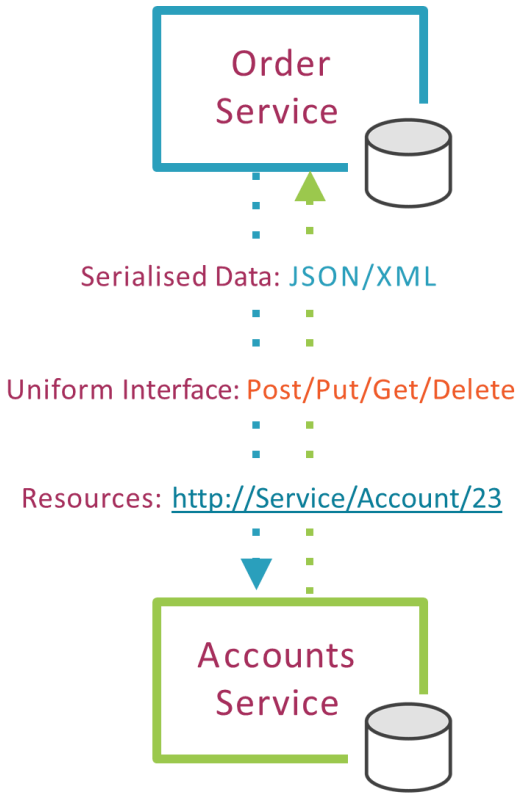
Estilo de arquitectura API

- Pragmatic REST
- HATEOS (True REST)
- RPC
- SOAP

Patrones de arquitectura API

- Facade pattern
- Proxy pattern
- Stateless service pattern

Estilo de arquitectura REST



Qué es REST?

- Es un estilo que define rersticciones
- Utiliza una infraestructura basada en HTTP
- Hereda las ventajas de la web
- Conceptos clave
  - JSON o XML
  - Endpoints de recursos
  - Interfaz de recursos uniforme

Consistencia Eventual

Existen sitios web que necesitan un poco de paciencia. Haces una actualización de algo, se actualiza la pantalla y falta la actualización. Esperas uno o dos minutos, pulsas Refresh, y ahí está.

Incoherencias como esta son lo suficientemente irritantes, pero pueden ser mucho más graves. La lógica empresarial puede terminar tomando decisiones sobre información inconsistente, cuando esto sucede puede ser extremadamente difícil diagnosticar lo que salió mal porque cualquier investigación se producirá mucho después de que se haya cerrado la ventana de incoherencia.

Los microservicios introducen problemas de coherencia eventuales debido a su loable insistencia en la administración descentralizada de datos. Con un monolito, puede actualizar un montón de cosas juntas en una sola transacción. Los microservicios requieren varios recursos para actualizar y las transacciones distribuidas se fruncen el ceño (por una buena razón). Por lo tanto, ahora, los desarrolladores deben ser conscientes de los problemas de coherencia y averiguar cómo detectar cuándo las cosas están fuera de sincronización antes de hacer cualquier cosa que el código se arrepentirá.

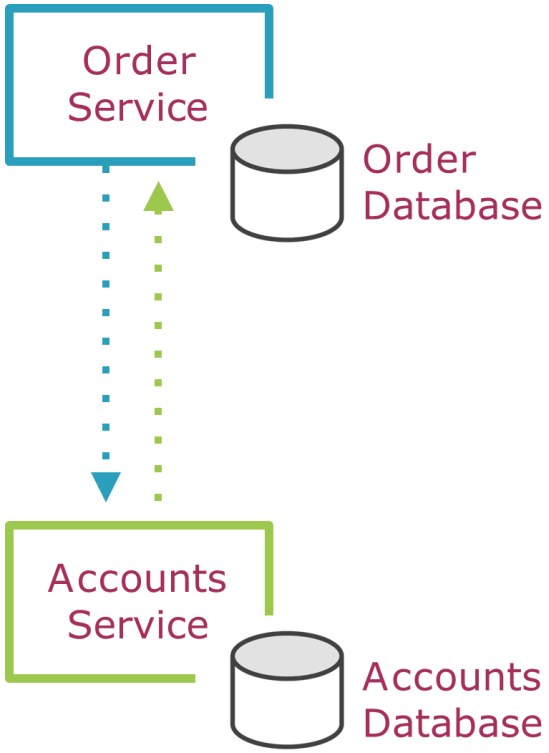
Consistencia Eventual

El mundo monolítico no está libre de estos problemas. A medida que los sistemas crecen, es más necesario usar el almacenamiento en caché para mejorar el rendimiento, y la invalidación de caché es el otro problema duro.

La mayoría de las aplicaciones necesitan bloqueos sin conexión para evitar transacciones de base de datos de larga duración. Los sistemas externos necesitan actualizaciones que no se puedan coordinar con un administrador de transacciones. Los procesos de negocio son a menudo más tolerantes a las incoherencias de lo que crees, porque las empresas a menudo valoran más la disponibilidad (los procesos de negocio han tenido durante mucho tiempo una comprensión instintiva del teorema de la PAC).

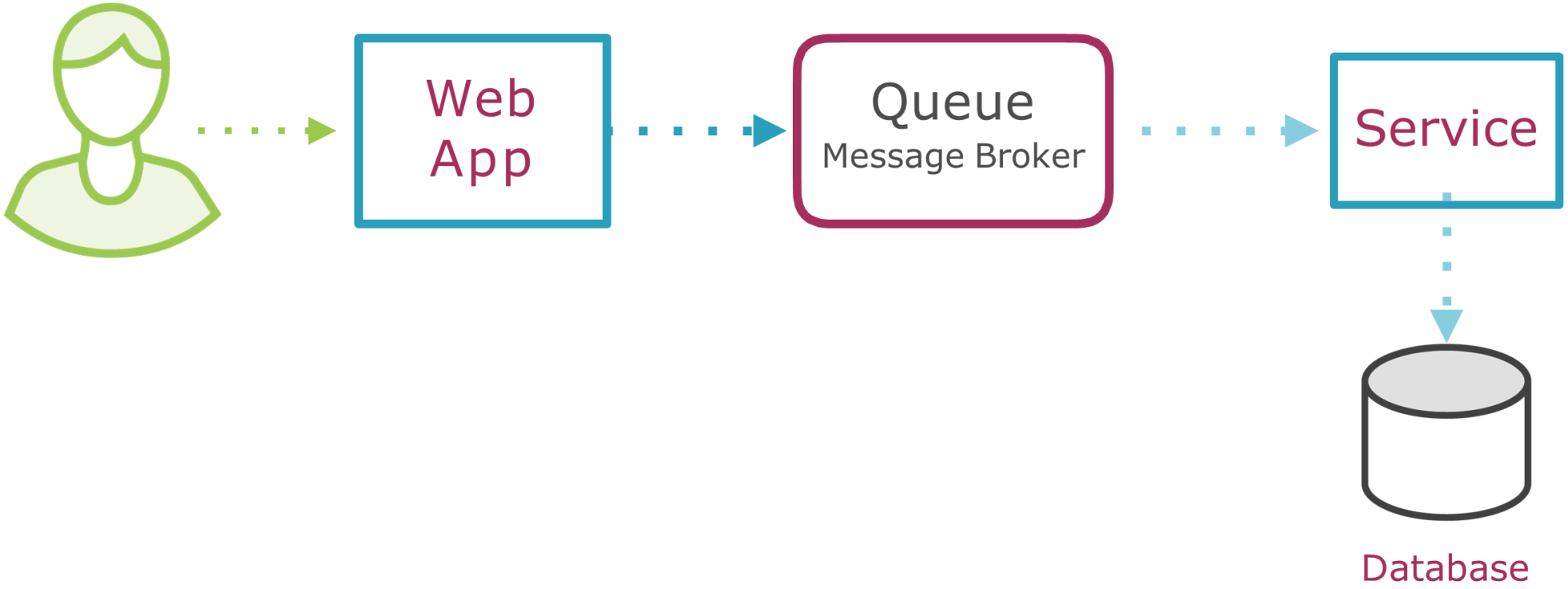
- **La consistencia (Consistency)**, es decir, cualquier lectura recibe como respuesta la escritura más reciente o un error.
- **La disponibilidad (Availability)**, es decir, cualqueir petición recibe una respuesta no errónea, pero sin la garantía de que contenga la escritura más reciente.
- **La tolerancia al particionado** (Partition Tolerance), es decir, el sistema sigue funcionado incluso si un número arbitrario de mensajes son descartados (o retrasados) entre nodos de la red.

Consistencia Eventual

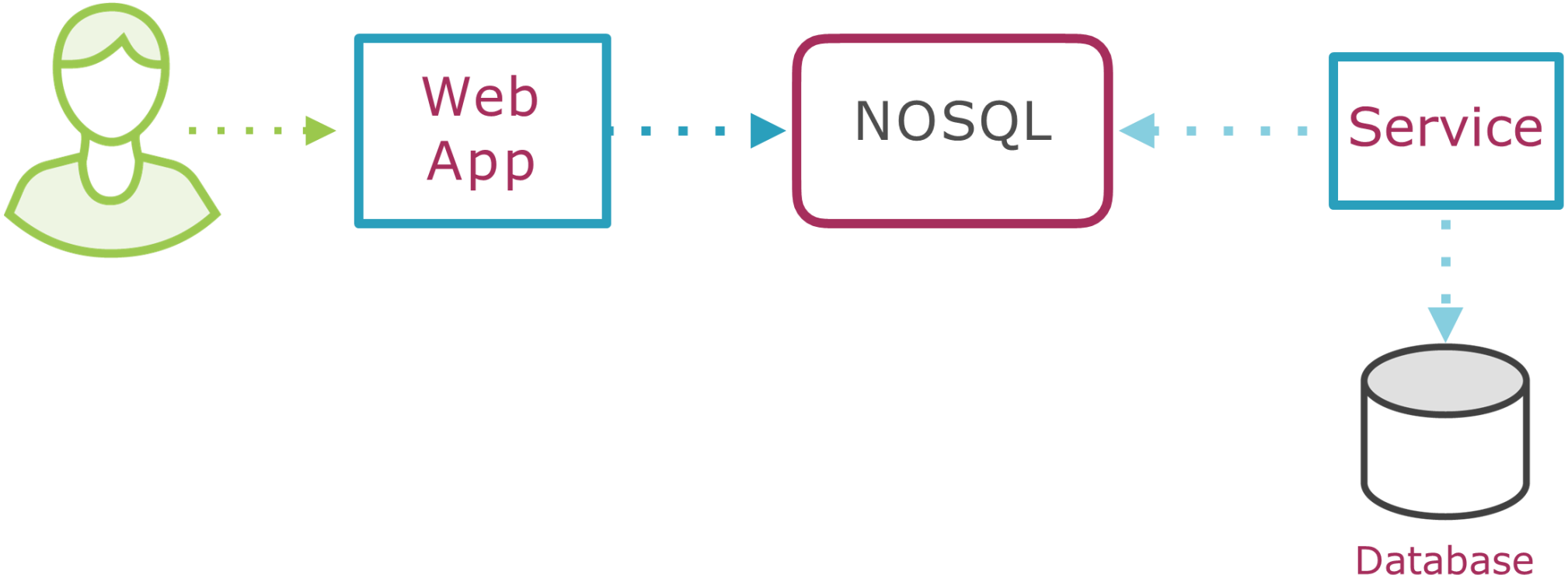


- Los datos eventualmente serán consistentes
  - BASE
  - BASE vs ACID
- Disponibilidad sobre consistencia
  - Evitar el bloqueo de recursos
  - Ideal para tareas de larga duración.
  - Preparado para inconsistencias
  - Condiciones de carrera
- Replicación de datos
- Basado en eventos
  - Transacción/acciones generadas como eventos
  - Mensajes usando message brokers

Consistencia Eventual

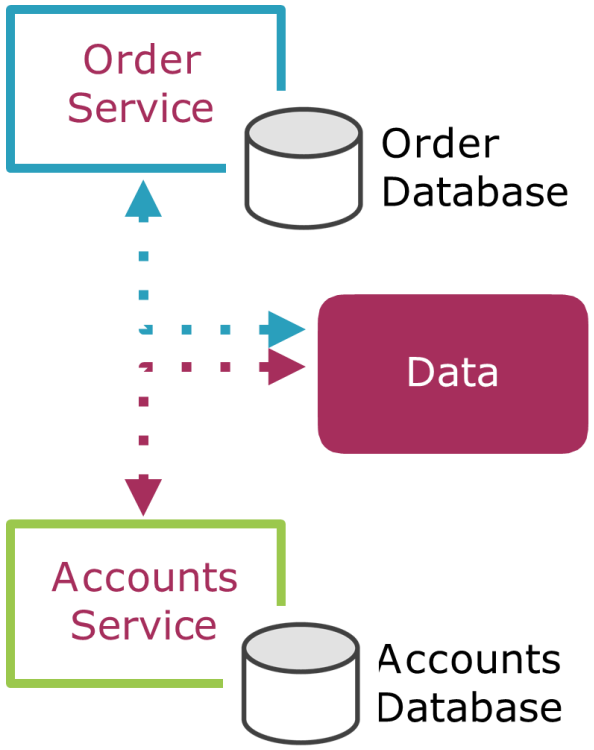


Consistencia Eventual



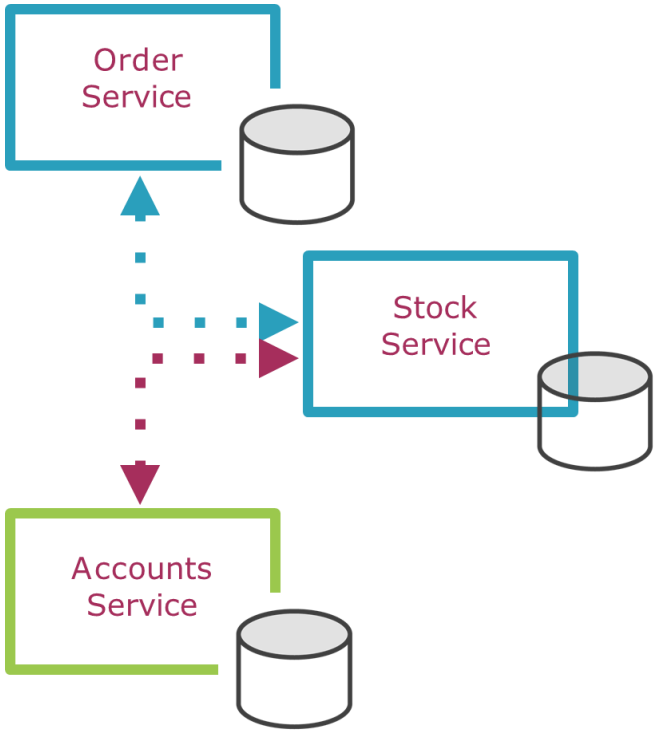
Implementación del patrón SAGA.

Introducción



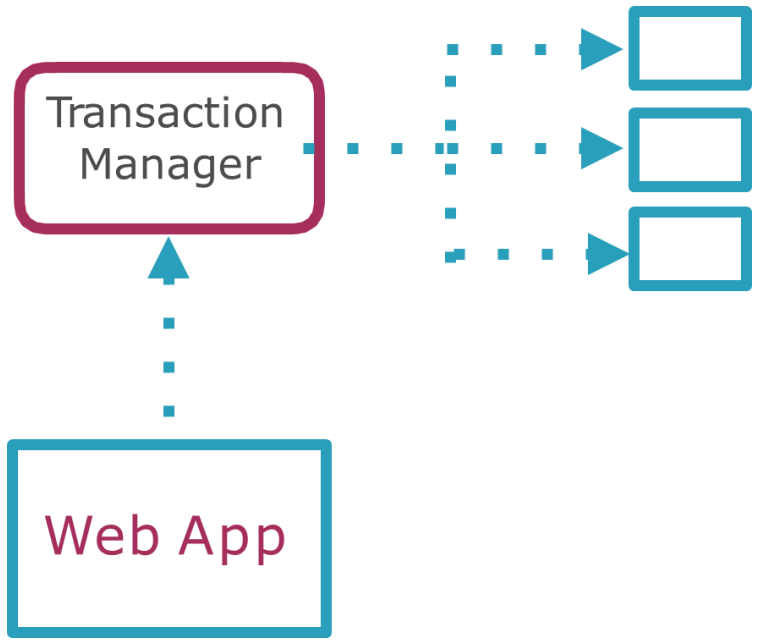
- Consistencia de los datos
  - Las transacciones son el enfoque tradicional.
- Transacciones del sistema monolítico
  - Base de datos única que es la única verdad
- Transacciones de microservicios
  - Arquitectura distribuida
  - Datos distribuidos
  - Transacciones distribuidas
- Teorema CAP
  - La falla de la red sucederá
  - Disponibilidad de datos o consistencia de datos?

Opciones



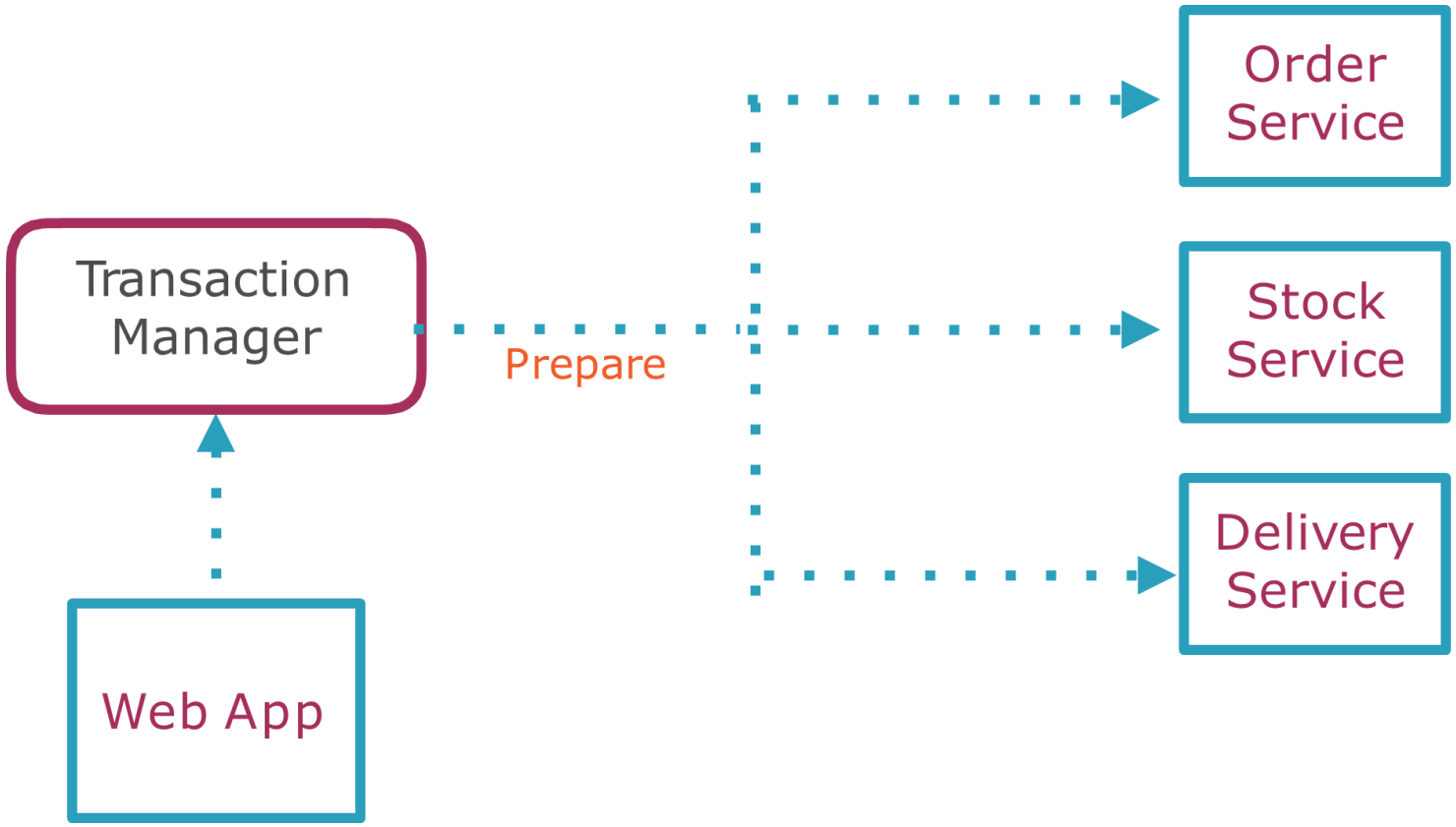
- Transacciones ACID tradicionales
  - Atomicidad, consistencia, aislamiento y durabilidad.
- Patrón de confirmación de dos fases (2PC)
  - ACID es obligatorio
  - Teorema CAP: elección de consistencia
- Patrón SAGA
  - Atomicidad por disponibilidad y consistencia
- Patrón de consistencia eventual
  - ACID
  - Teorema CAP: elección de disponibilidad

Confirmación de 2 fases (Two Phase Commit / 2PC)

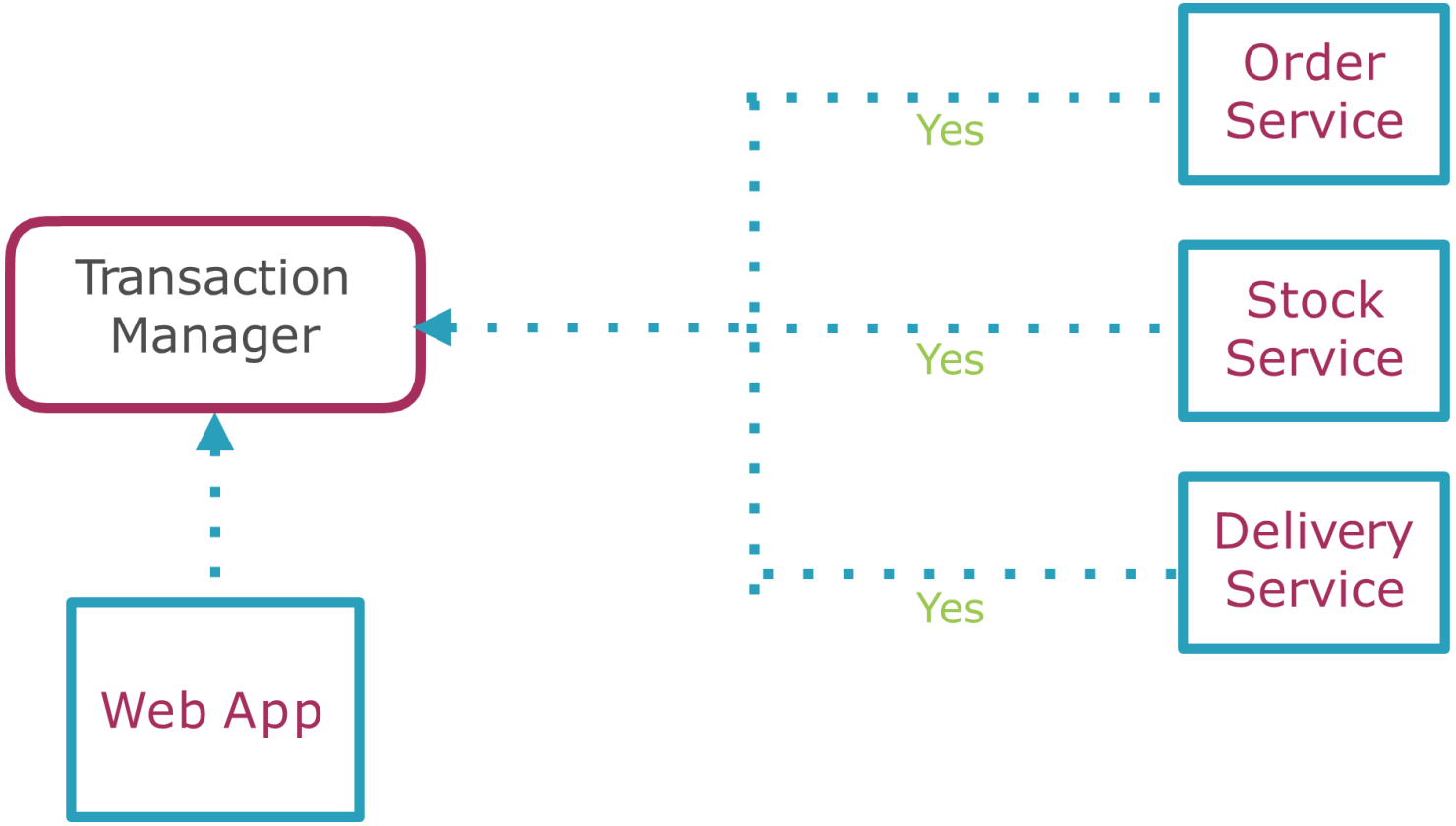


- Patrón para transacciones distribuidas
  - El administrador de transacciones maneja las transacciones
- Fase de preparación
  - El gestor de transacciones notifica el inicio de preparación.
- Fase de confirmación
  - El gestor de transacciones recibe las confirmaciones.
  - Gestor de transacciones
    - Emite un Commit en caso todos hayan confirmado.
    - Emite un Rollback en caso uno no haya confirmado.

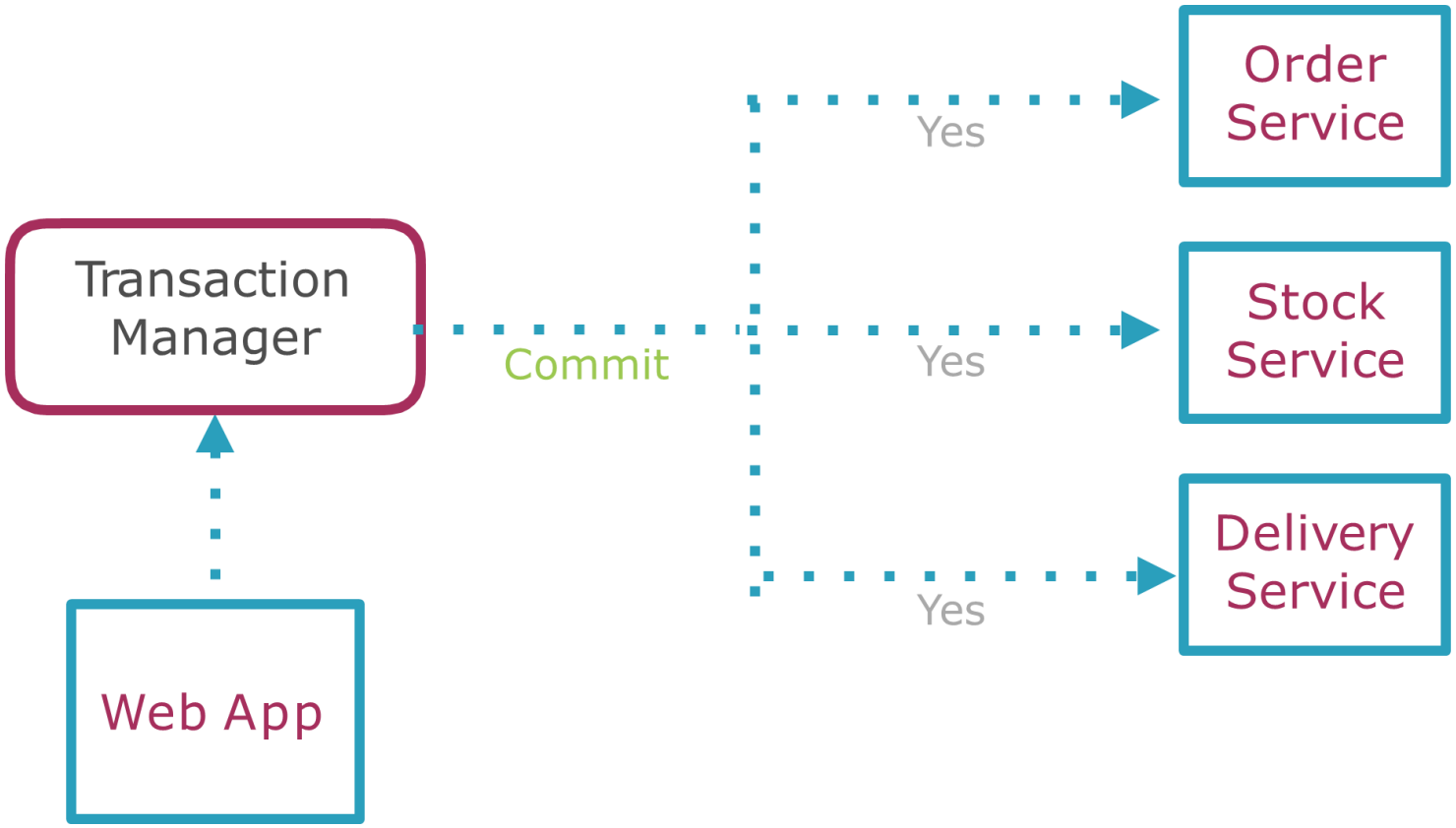
Confirmación de 2 fases – Fase de preparación



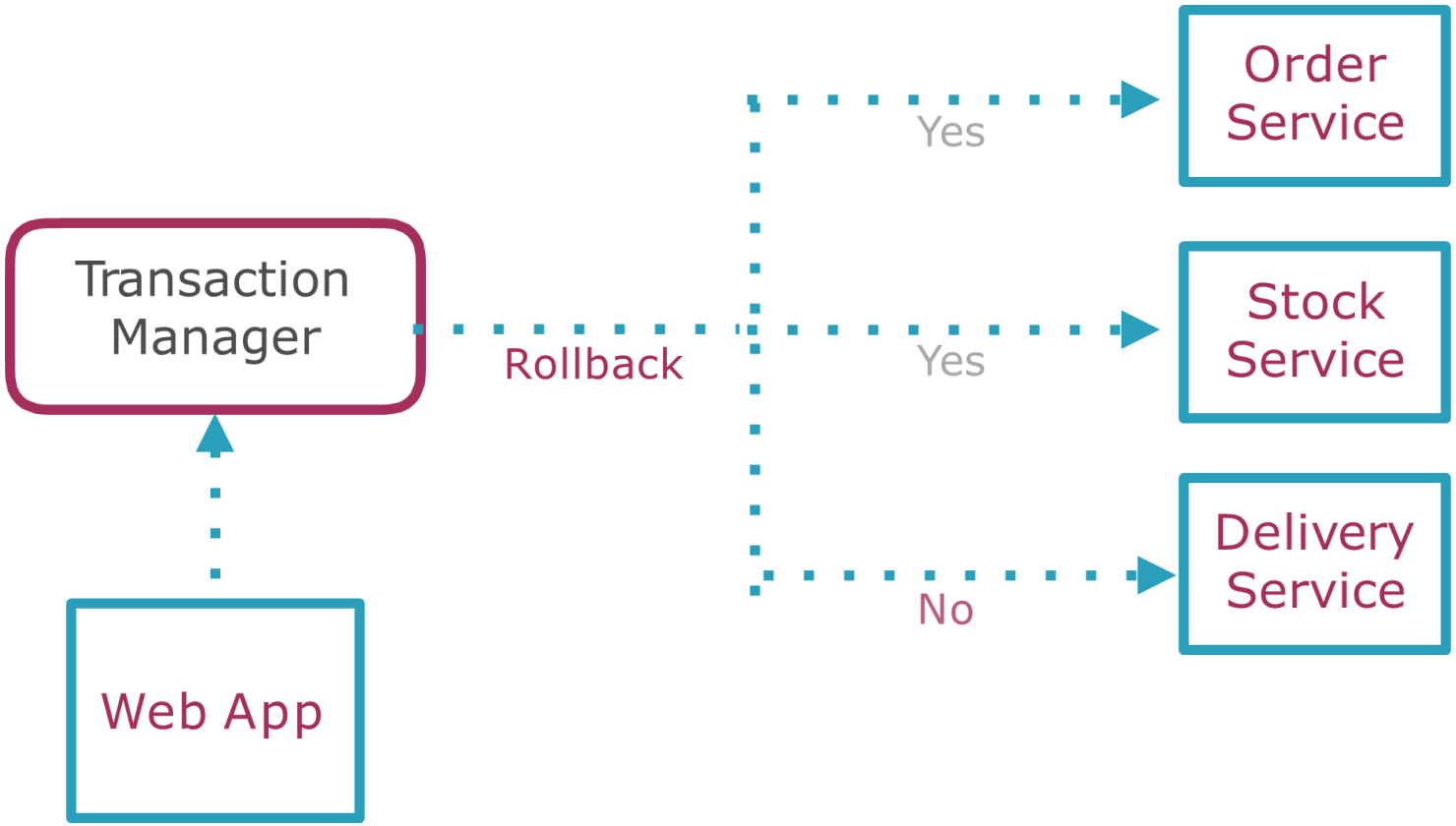
Confirmación de 2 fases – Fase de Confirmación



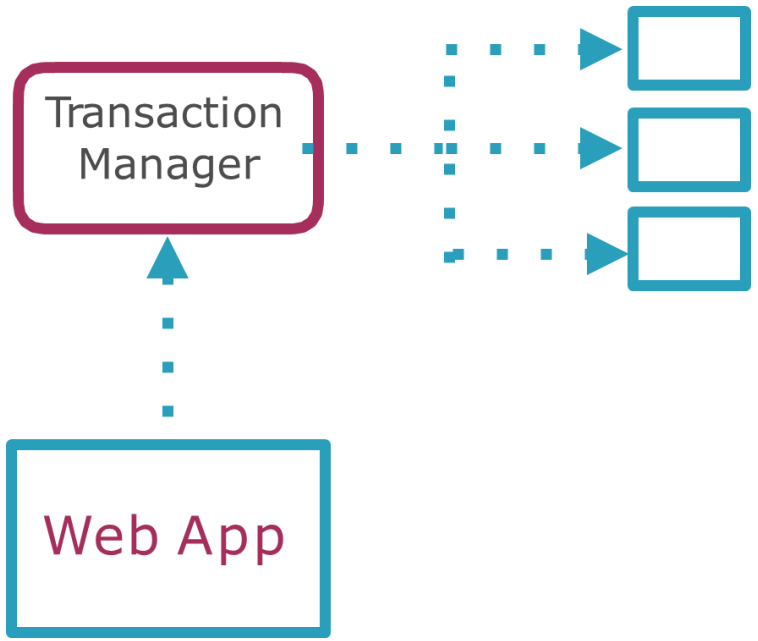
Confirmación de 2 fases – Commit



Confirmación de 2 fases – Rollback



Confirmación de 2 fases (Two Phase Commit / 2PC)



- Advertencias
  - Confianza en un administrador de transacciones
  - Sin respuesta de confirmación
  - Fallar después de una confirmación
  - Las transacciones pendientes bloquean recursos
  - Evitar implementaciones personalizadas
  - Tiene problemas de escalado
  - Rendimiento reducido
  - Antipatrón
- Considerar alternativas
  - Patrón de saga
  - Consistencia eventual

Saga - Contexto y problema

Una transacción es una única unidad de lógica o de trabajo, que a veces se compone de varias operaciones. Dentro de una transacción, un evento es un cambio de estado que se produce en una entidad y un comando encapsula toda la información necesaria para realizar una acción o desencadenar un evento posterior.

Cada transacción debe ser atómica, coherente, aislada y durable (**ACID**) . Las transacciones dentro de un único servicio son ACID, pero la coherencia de los datos entre servicios requiere una estrategia de administración de transacciones entre servicios.

En arquitecturas de multiservicio:

- La **atomicidad** es un conjunto de operaciones indivisibles e irreducible que deben realizarse todas o ninguna.
- La **coherencia** significa que la transacción coloca los datos solo de un estado válido a otro estado válido.
- El **aislamiento** garantiza que las transacciones simultáneas produzcan el mismo estado de datos que habrían producido transacciones ejecutadas secuencialmente.
- La **durabilidad** garantiza que las transacciones confirmadas permanezcan confirmadas incluso en caso de error del sistema o de interrupción del suministro eléctrico.

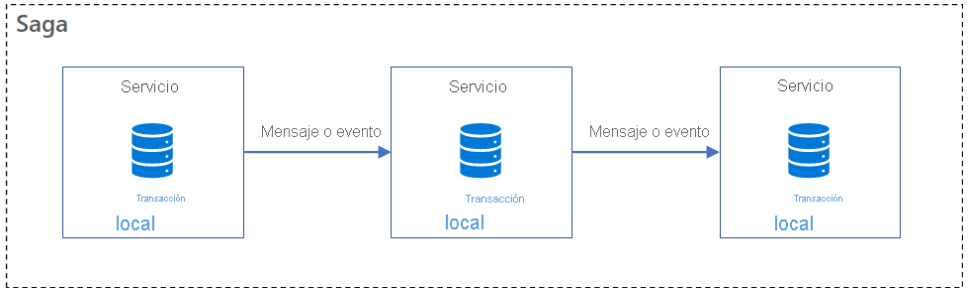
Saga - Contexto y problema

Un modelo de base de datos por microservicio proporciona muchas ventajas para las arquitecturas de microservicios. La encapsulación de los datos del dominio permite a cada servicio usar su mejor tipo de almacén de datos y esquema, escalar su propio almacén de datos según sea necesario y aislarse de los errores de otros servicios. Sin embargo, garantizar la coherencia de los datos entre las bases de datos específicas del servicio plantea desafíos.

Las transacciones distribuidas, como el protocolo de confirmación en dos fases (2PC), requieren que todos los participantes de una transacción la confirmen o reviertan antes de que la transacción pueda continuar. No obstante, algunas implementaciones de participantes, como las bases de datos NoSQL y la administración de mensajes, no admiten este modelo.

Otra limitación de las transacciones distribuidas es la sincronicidad y la disponibilidad de la comunicación entre procesos (IPC). La IPC proporcionada por el sistema operativo permite que procesos independientes compartan datos. Para que se confirmen las transacciones distribuidas, todos los servicios participantes deben estar disponibles, lo que puede reducir la disponibilidad general del sistema. Las implementaciones arquitectónicas con limitaciones de transacciones o IPC son candidatas para el patrón de saga.

Saga - Solución

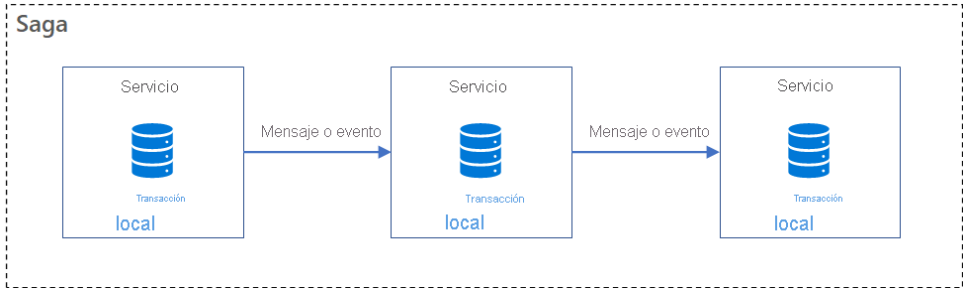


El patrón de saga proporciona la administración de transacciones mediante una secuencia de transacciones locales.

Una transacción local es el esfuerzo del trabajo atómico realizado por un participante de la saga.

Cada transacción local actualiza la base de datos y publica un mensaje o evento para desencadenar la siguiente transacción local en la saga. Si se produce un error en una transacción local, la saga ejecuta una serie de transacciones de compensación que deshacen los cambios realizados por las transacciones locales anteriores.

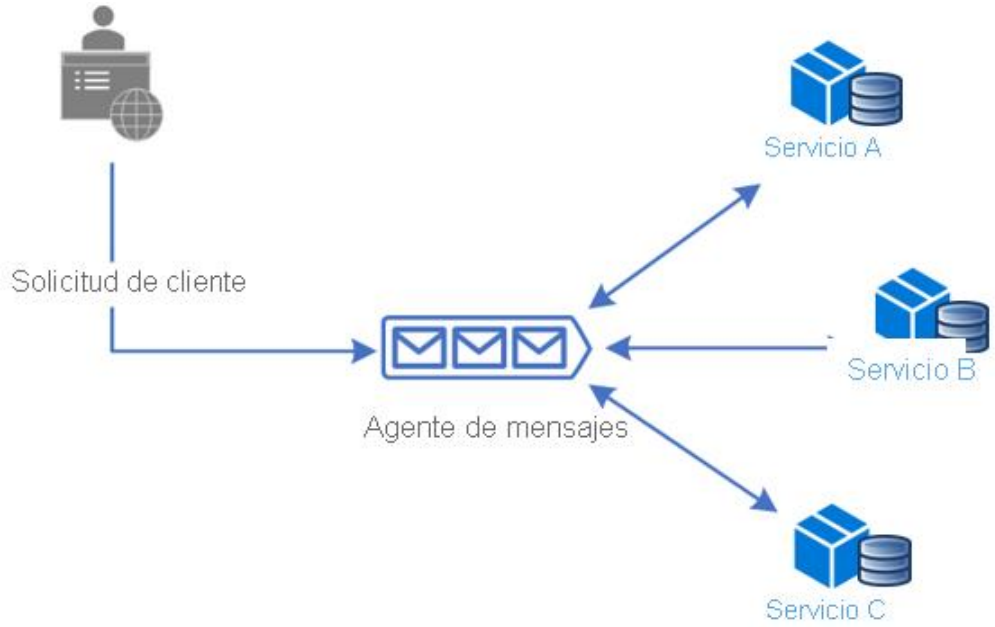
Saga - Solución



En patrones de saga:

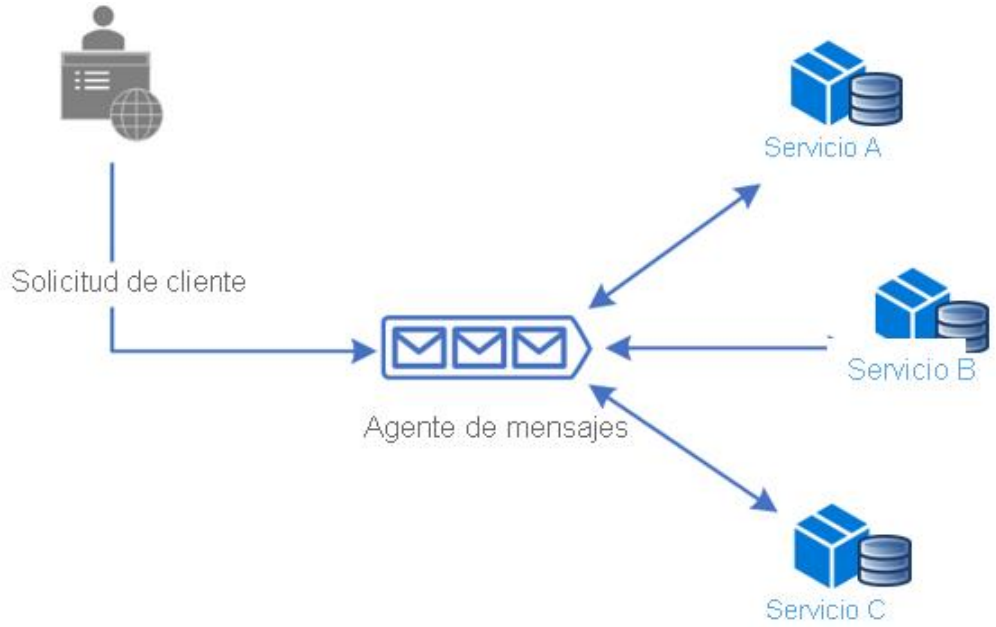
- Las transacciones compensables son transacciones que se pueden invertir procesando otra transacción con el efecto opuesto.
- Una transacción dinámica es el punto en el que se debe continuar o no continuar en una saga. Si se confirma la transacción dinámica, la saga se ejecuta hasta su finalización. Una transacción dinámica puede ser una transacción que no es compensable ni reintentable, o puede ser la última transacción compensable o la primera transacción reintentable de la saga.
- Las transacciones reintentables son transacciones que siguen a la transacción dinámica y que está garantizado que se realizarán correctamente.

Saga - Corografía



La coreografía es una manera de coordinar sagas en la que los participantes intercambian eventos sin un punto de control centralizado. Con la coreografía, cada transacción local publica eventos de dominio del dominio que desencadenan transacciones locales en otros servicios.

Saga - Corografía



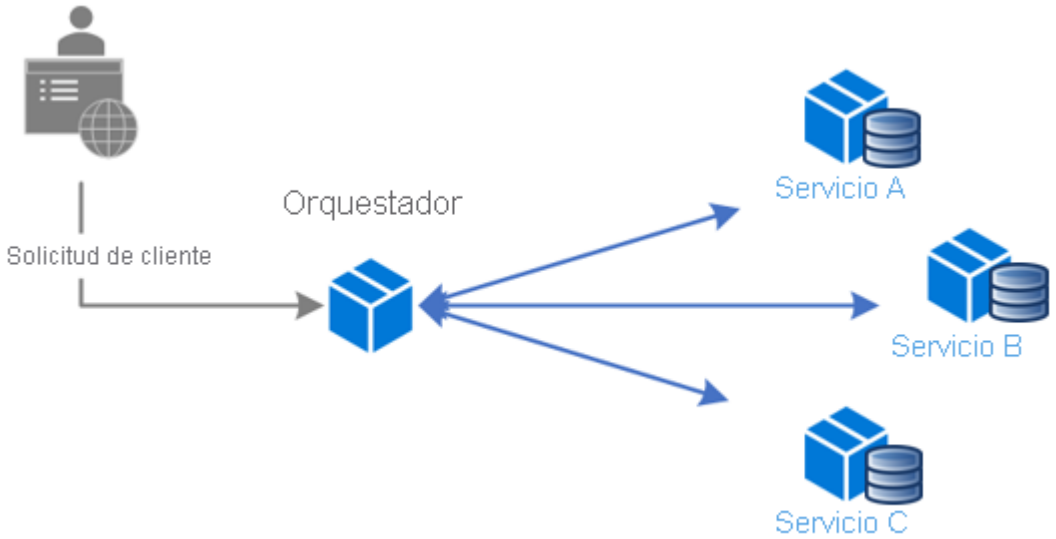
Ventajas

- Adecuada para flujos de trabajo sencillos que requieren pocos participantes y no necesitan una lógica de coordinación.
- No requiere implementación y mantenimiento adicionales del servicio.
- No introduce un único punto de error, ya que las responsabilidades se distribuyen entre los participantes de la saga.

Inconvenientes

- El flujo de trabajo puede resultar confuso al agregar nuevos pasos, ya que es difícil realizar un seguimiento de qué participantes de la saga escuchan y qué comandos.
- Existe un riesgo de dependencia cíclica entre los participantes de la saga porque tienen que usar los comandos de los demás.
- Las pruebas de integración son difíciles porque todos los servicios deben estar en ejecución para simular una transacción.

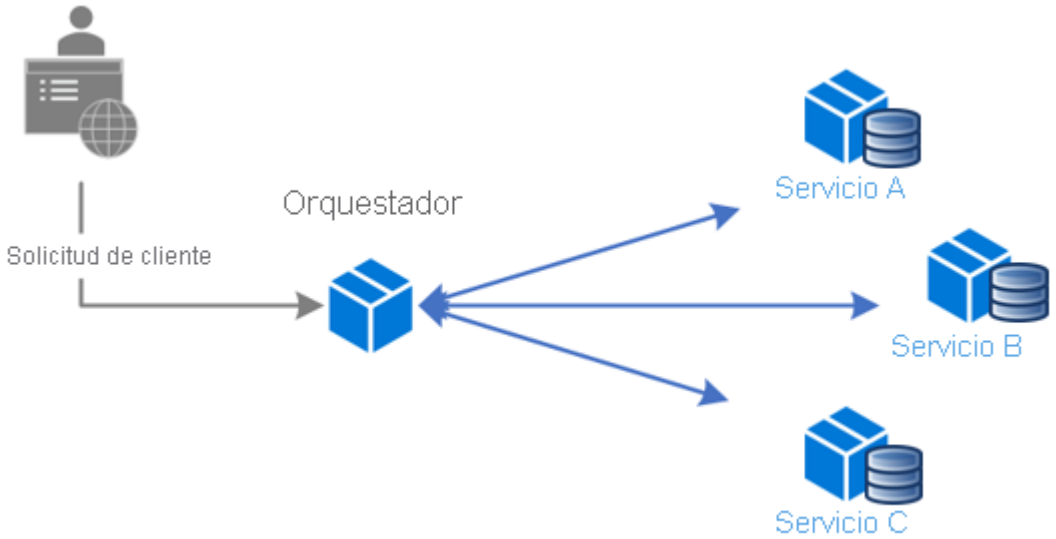
Saga - Orquestación



La orquestación es una manera de coordinar sagas donde un controlador centralizado indica a los participantes de la saga qué transacciones locales se deben ejecutar.

El orquestador de la saga administra todas las transacciones e indica a los participantes qué operación deben realizar en función de los eventos. El orquestador ejecuta solicitudes de saga, almacena e interpreta los estados de cada tarea y controla la recuperación de errores con transacciones de compensación.

Saga - Orquestación



Ventajas

- Válida para flujos de trabajo complejos que impliquen a muchos participantes o la adición de nuevos participantes a lo largo del tiempo.
- Adecuada cuando se controla a cada participante del proceso y se controla el flujo de actividades.
- No introduce dependencias cíclicas, ya que el orquestador depende unilateralmente de los participantes de la saga.
- No es necesario que los participantes de la saga conozcan los comandos de otros participantes. La separación clara de los problemas simplifica la lógica empresarial.

Inconvenientes

- La complejidad del diseño adicional requiere una implementación de una lógica de coordinación.
- Existe un punto de error adicional, ya que el orquestador administra todo el flujo de trabajo.

Saga – Problemas y consideraciones

Tenga en cuenta los puntos siguientes al implementar el patrón de saga:

- El patrón de saga inicialmente puede ser desafiante, ya que requiere una nueva forma de pensar sobre cómo coordinar una transacción y mantener la coherencia de los datos para un proceso empresarial que abarque varios microservicios.
- El patrón de saga es especialmente difícil de depurar y la complejidad crece a medida que los participantes aumentan.
- No se pueden revertir los datos porque los participantes de la saga confirman los cambios en sus bases de datos locales.
- La implementación debe ser capaz de controlar un conjunto de posibles errores transitorios y proporcionar idempotencia para reducir los efectos secundarios y garantizar la coherencia de los datos. La idempotencia significa que la misma operación se puede repetir varias veces sin cambiar el resultado inicial.
- Es mejor implementar la observabilidad para supervisar y realizar un seguimiento del flujo de trabajo de la saga.
- La falta de aislamiento de los datos de los participantes impone desafíos de durabilidad. La implementación de la saga debe incluir contramedidas para reducir las anomalías.

Saga – Problemas y consideraciones

Pueden producirse las siguientes anomalías si no se toman las medidas adecuadas:

- **Pérdida de actualizaciones**, cuando un patrón de saga escribe sin leer los cambios realizados por otro patrón de saga.
- **Lecturas de datos sucios**, cuando una transacción o una saga lee las actualizaciones realizadas por una saga que todavía no ha completado dichas actualizaciones.
- **Lecturas aproximadas/no repetibles**, cuando diferentes pasos de la saga leen datos diferentes porque se produce una actualización de datos entre las lecturas.

Saga – Problemas y consideraciones

Entre las contramedidas sugeridas para reducir o evitar anomalías se incluyen:

- El **bloqueo semántico**, un bloqueo de nivel de aplicación en el que una transacción de compensable de una saga utiliza un semáforo para indicar que hay una actualización en curso.
- Las **actualizaciones conmutativas** que se pueden ejecutar en cualquier orden y generar el mismo resultado.
- **Visión pesimista**: Es posible que una saga lea datos sucios mientras otra saga ejecuta una transacción compensable para revertir la operación. La visión pesimista reordena la saga para que los datos subyacentes se actualicen en una transacción reintentable, lo que elimina la posibilidad de una lectura de datos sucios.
- El **valor de relectura** comprueba que los datos no se hayan modificado y, a continuación, actualiza el registro. Si el registro ha cambiado, los pasos se anulan y la saga puede reiniciarse.
- Un **archivo de versión** registra las operaciones en un registro a medida que llegan y las ejecuta en el orden correcto.
- **Por valor** usa el riesgo empresarial de cada solicitud para seleccionar dinámicamente el mecanismo de simultaneidad. Las solicitudes de bajo riesgo favorecen a las sagas, mientras que las solicitudes de alto riesgo favorecen a las transacciones distribuidas.

Cuando usar SAGA

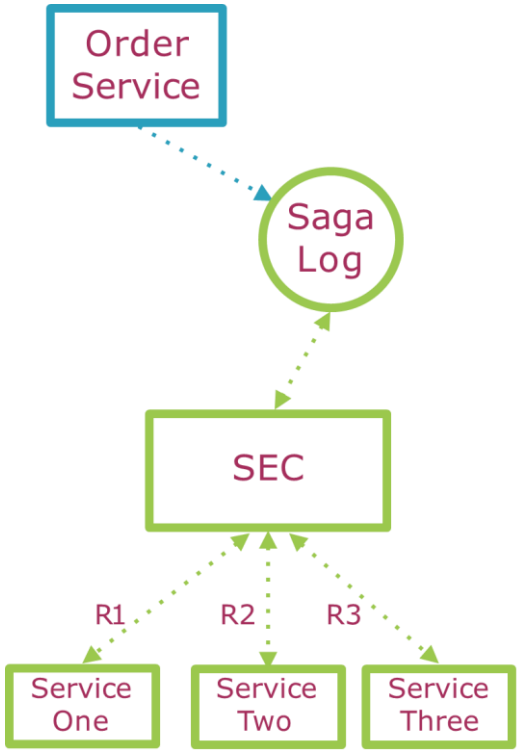
Use el patrón de saga cuando necesite:

- Garantizar la coherencia de los datos en un sistema distribuido sin acoplamiento estricto.
- Revertir o compensar si se produce un error en una de las operaciones de la secuencia.

El patrón de saga es menos adecuado para:

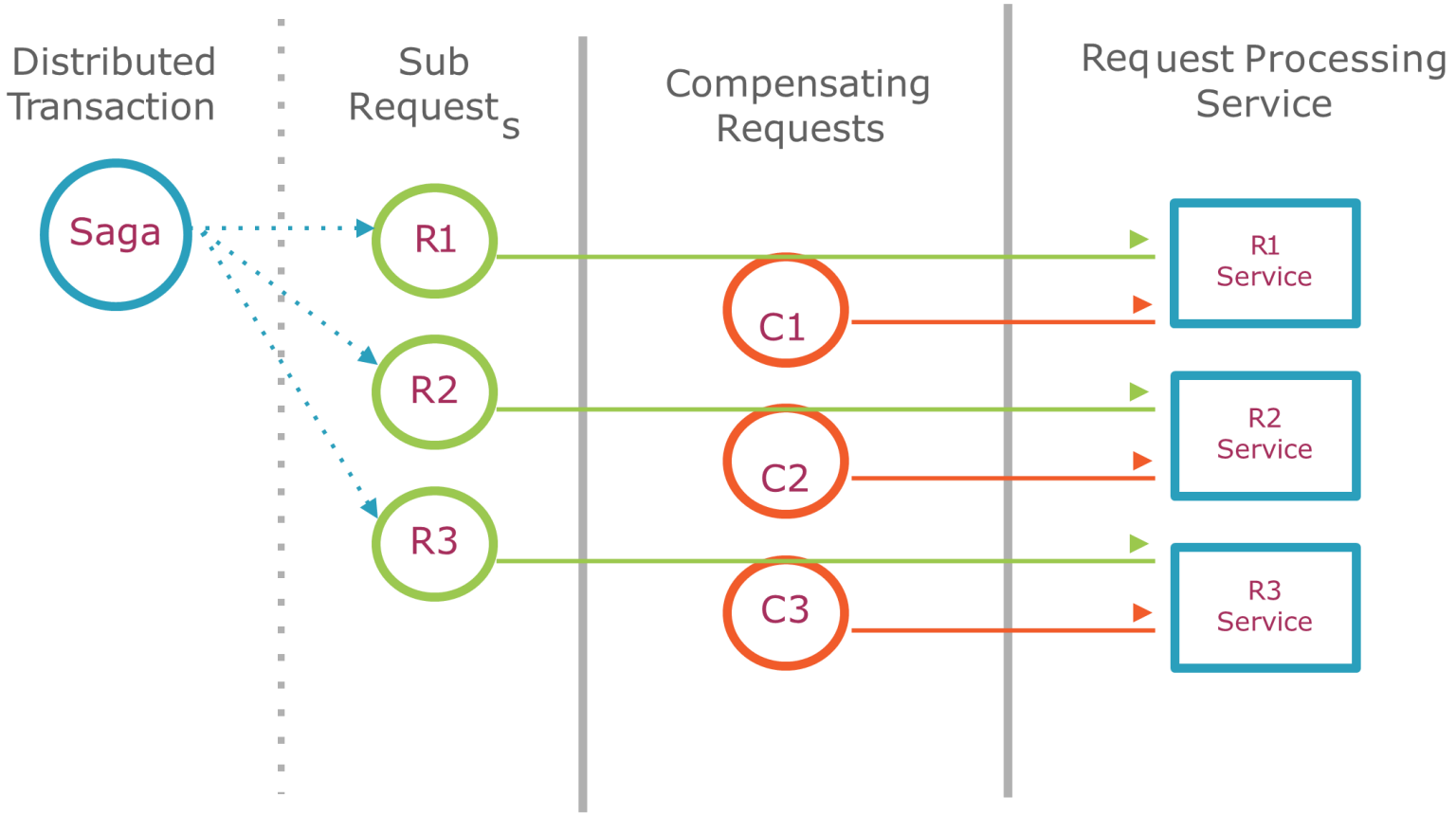
- Transacciones estrechamente acopladas.
- Transacciones de compensación que se producen en participantes anteriores.
- Dependencias cíclicas.

Patrón Saga

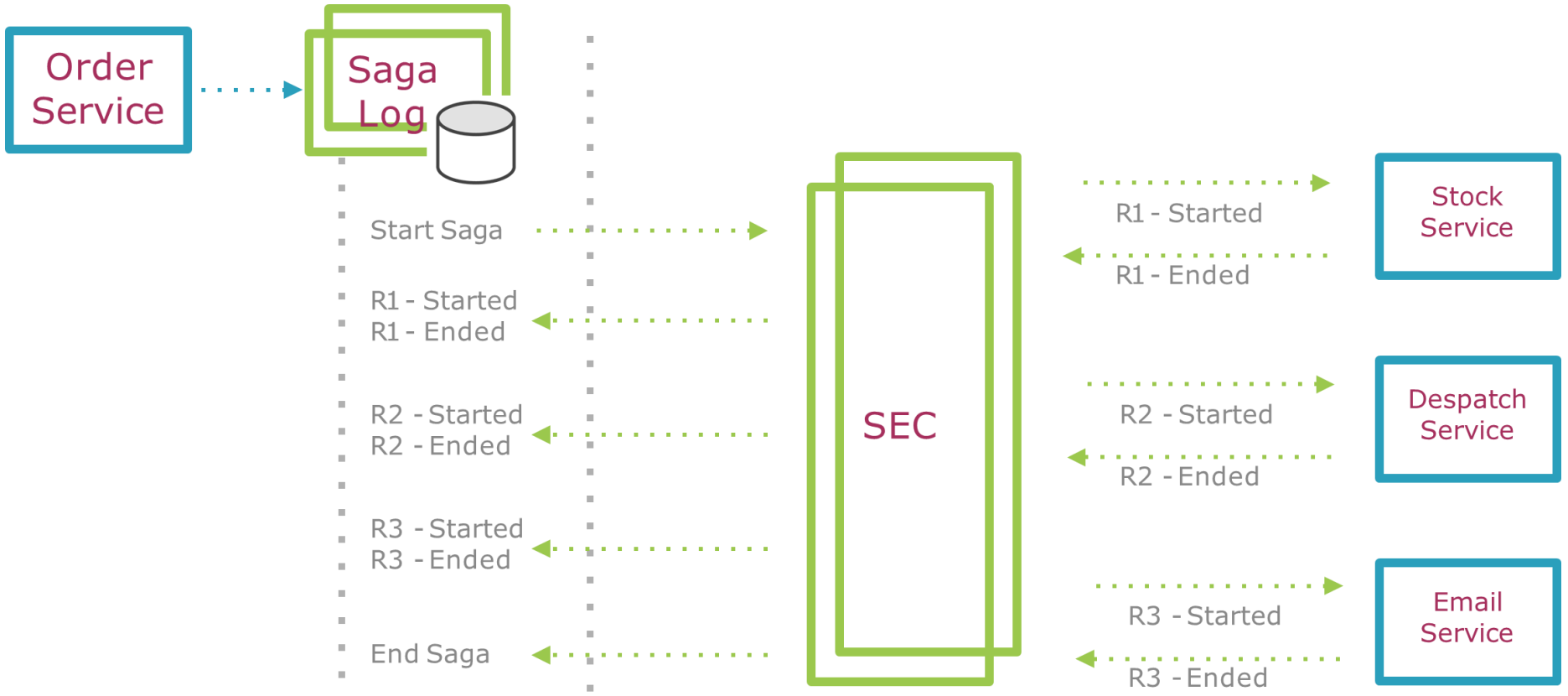


- Reemplaza una transacción distribuida con una saga
  - Divide la transacción en muchas solicitudes
  - Rastrea cada solicitud
  - ACID: compromiso de atomicidad
  - Descrito por primera vez en 1987
- También es un patrón de gestión de fallas
  - Qué hacer cuando falla un servicio
  - Compensar solicitudes
- Implementación
  - SAGA Log
  - Coordinador de ejecución de saga (SEC)
  - Solicitudes y compensación de solicitudes

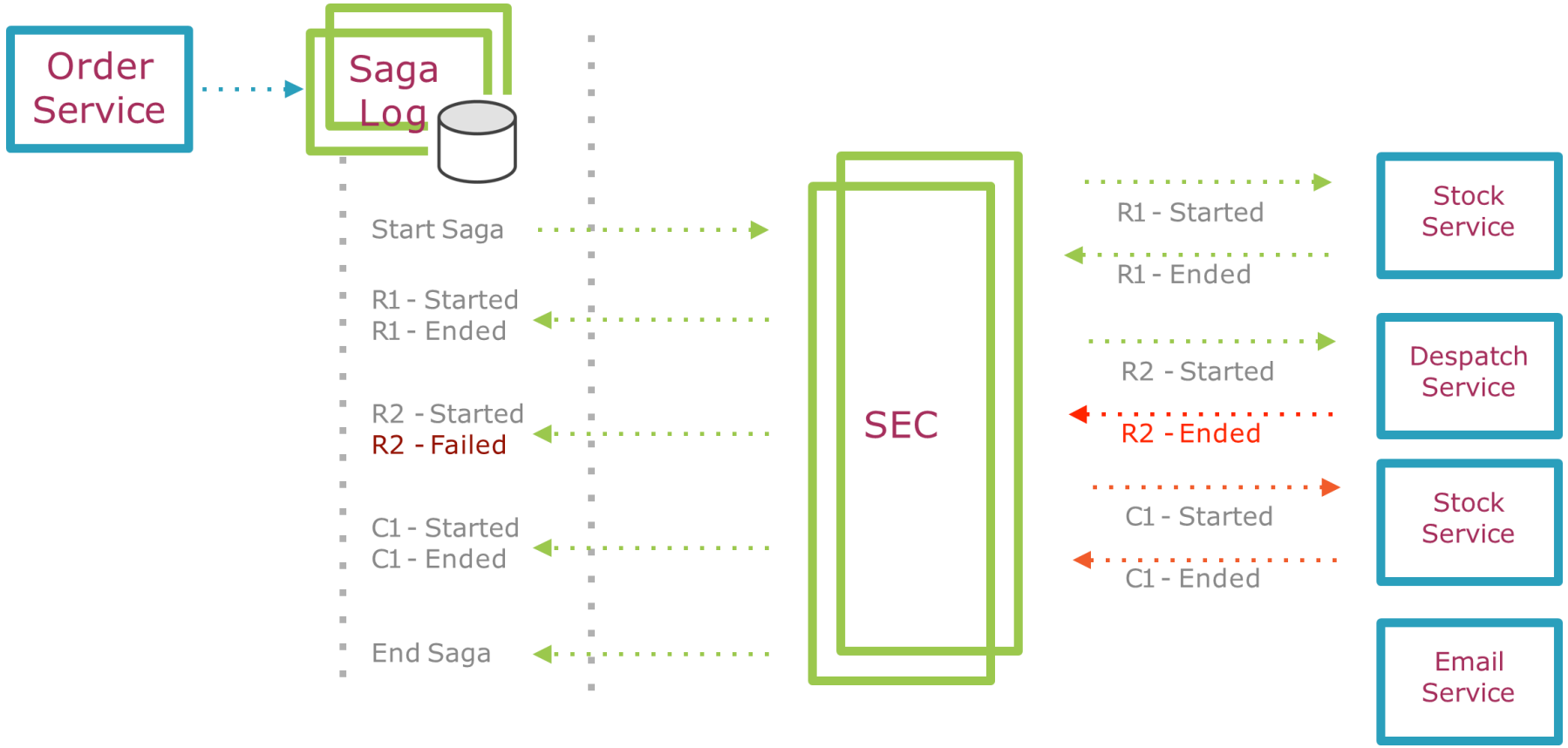
Patrón Saga


Distributed Transaction
Sub Requests
Compensating Requests
Request Processing Service
Saga
R1
R2
R3
C1
C2
C3
R1 Service
R2 Service
R3 Service

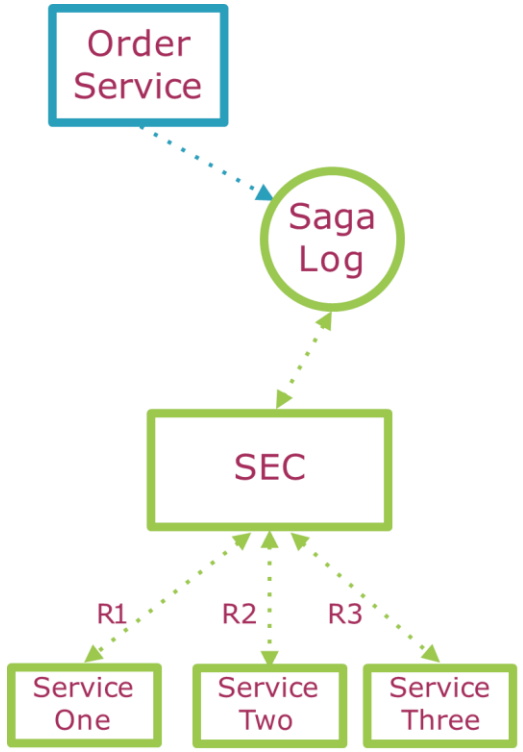
Saga exitoso



Saga no exitoso



Patrón Saga implementación



- Servicio que inicia la saga.
  - Envía la solicitud de saga al registro de saga
- Saga Log
  - Servicio con una base de datos
- SEC
  - Interpreta y escribe en el registro.
  - Envía solicitudes de saga
  - Envía solicitudes de compensación de saga
  - Recuperación: estado seguro vs estado inseguro
- Solicitudes de Compensación
  - Enviar en caso de error para todas las solicitudes completadas
  - Idempotente (fácil con REST)
  - Cada uno se envía de cero a muchas veces

GRACIAS

POR SU PREFERENCIA

GALAXY
TRAINING