PREPÁRATE
PARA SER EL
MEJOR

+ ENTREMIENTO
EXPERIENCIA

BIENVENIDOS.

ENTRENAMIENTO
EXPERIENCIA

GALAXY
TRAINING

Microservicios con .NET Core: Arquitectura, Contenerización y Orquestación

Sesión 02

Ing. Erick Arostegui Cunza
Instructor

earostegui@galaxy.edu.pe

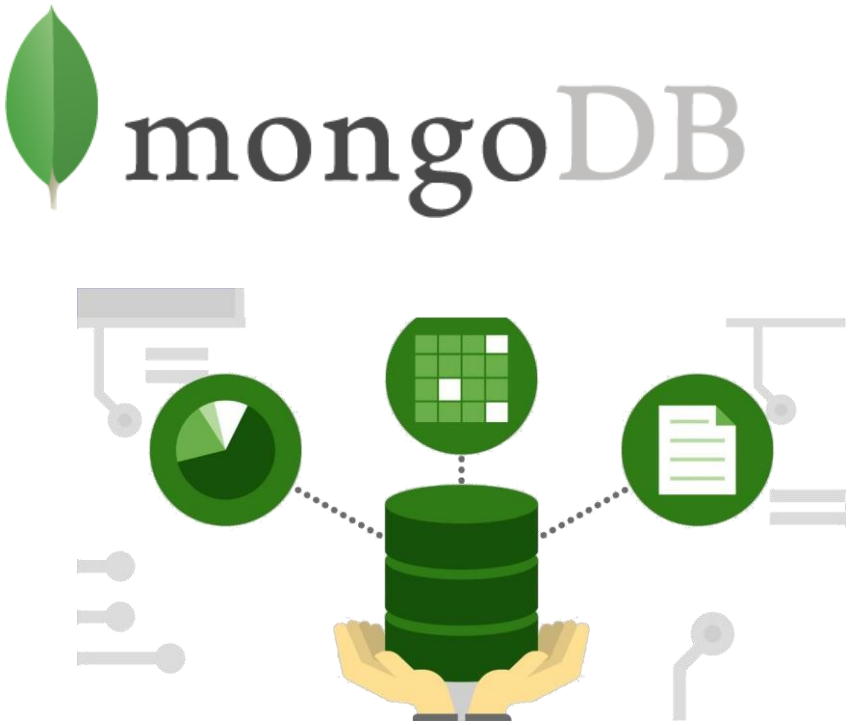
AGENDA

ARQ. DE MICROSERVICIOS (PERSISTENCIA Y CONSISTENCIA DE DATOS)

- Infraestructura de persistencia – NoSQL (CosmoDB).
- Implementando el patrón CQRS a un microservicio DDD.
- Inyección de Dependencias (DI .Net Core).
- Cómo lograr la consistencia de datos a través de microservicios (consistencia eventual).
- Implementación del patrón SAGA.

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

ARQUITECTURA DE MICROSERVICIOS (PERSISTENCIA Y CONSISTENCIA DE DATOS)

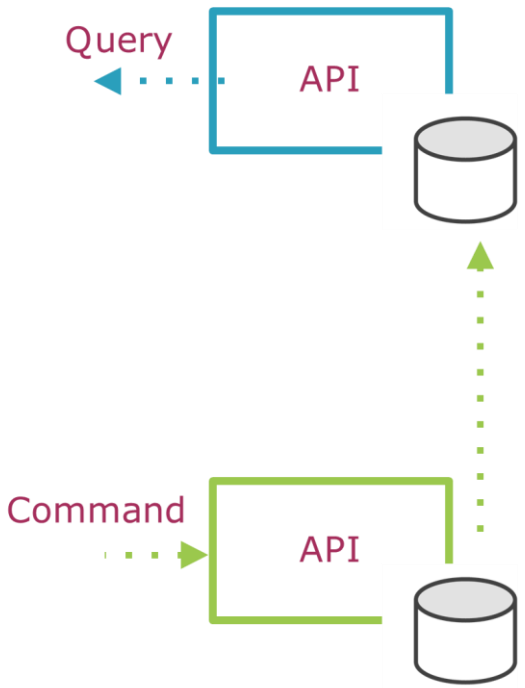
Intermedio

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

Ing. Erick Arostegui

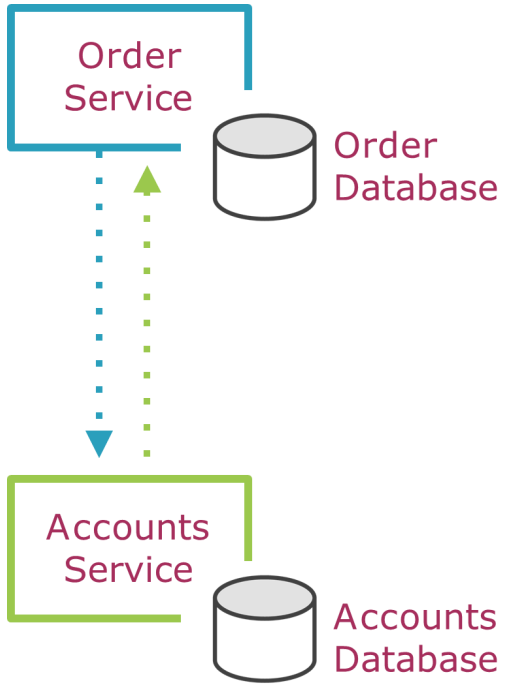
Implementando el patrón CQRS a un microservicio DDD.

Command Query Response Segregation



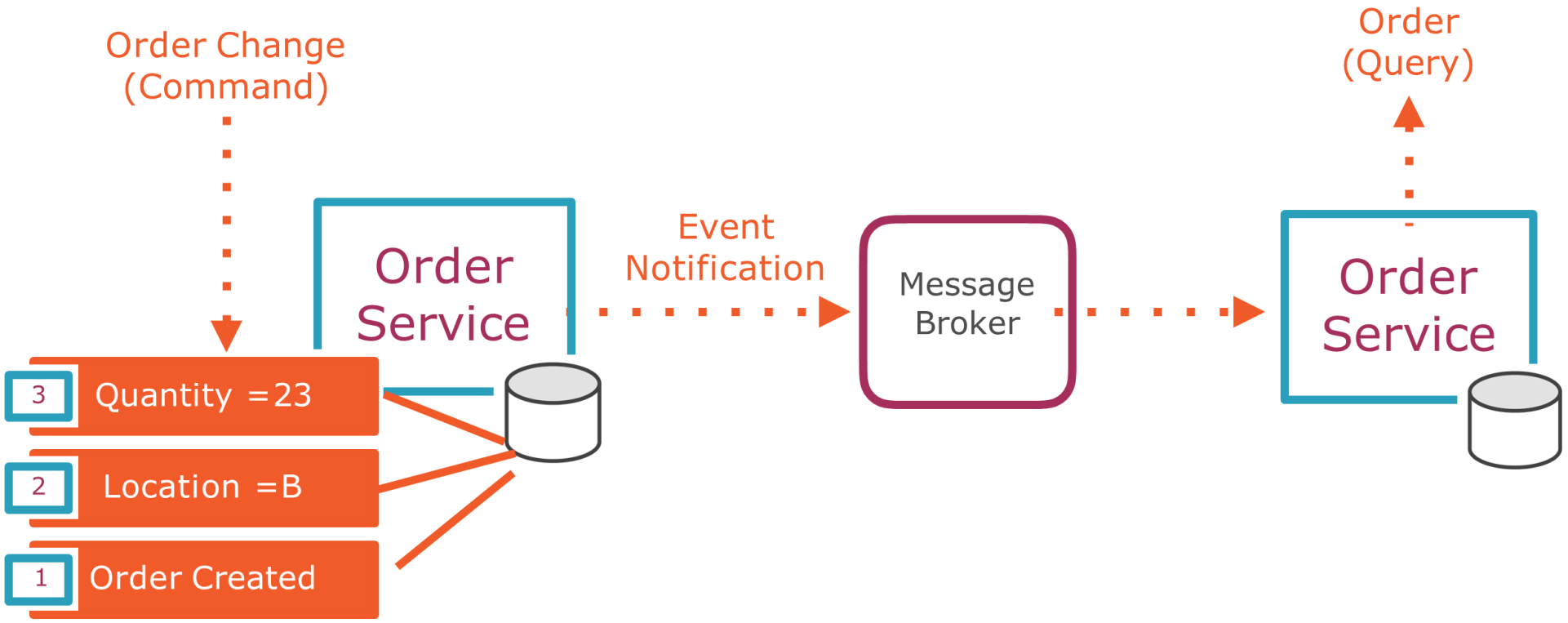
- CQRS
  - Modelos de comando (**Command**) y/o servicios
  - Modelos de consulta (**Query**) y/o servicios
- ¿Por qué?
  - Separación de responsabilidades.
    - Notificaciones de eventos manejadas por comando (**Escritura**)
    - Informes/funciones manejadas por consulta (**Lectura**)
  - Separación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

CQRS ¿Cómo?

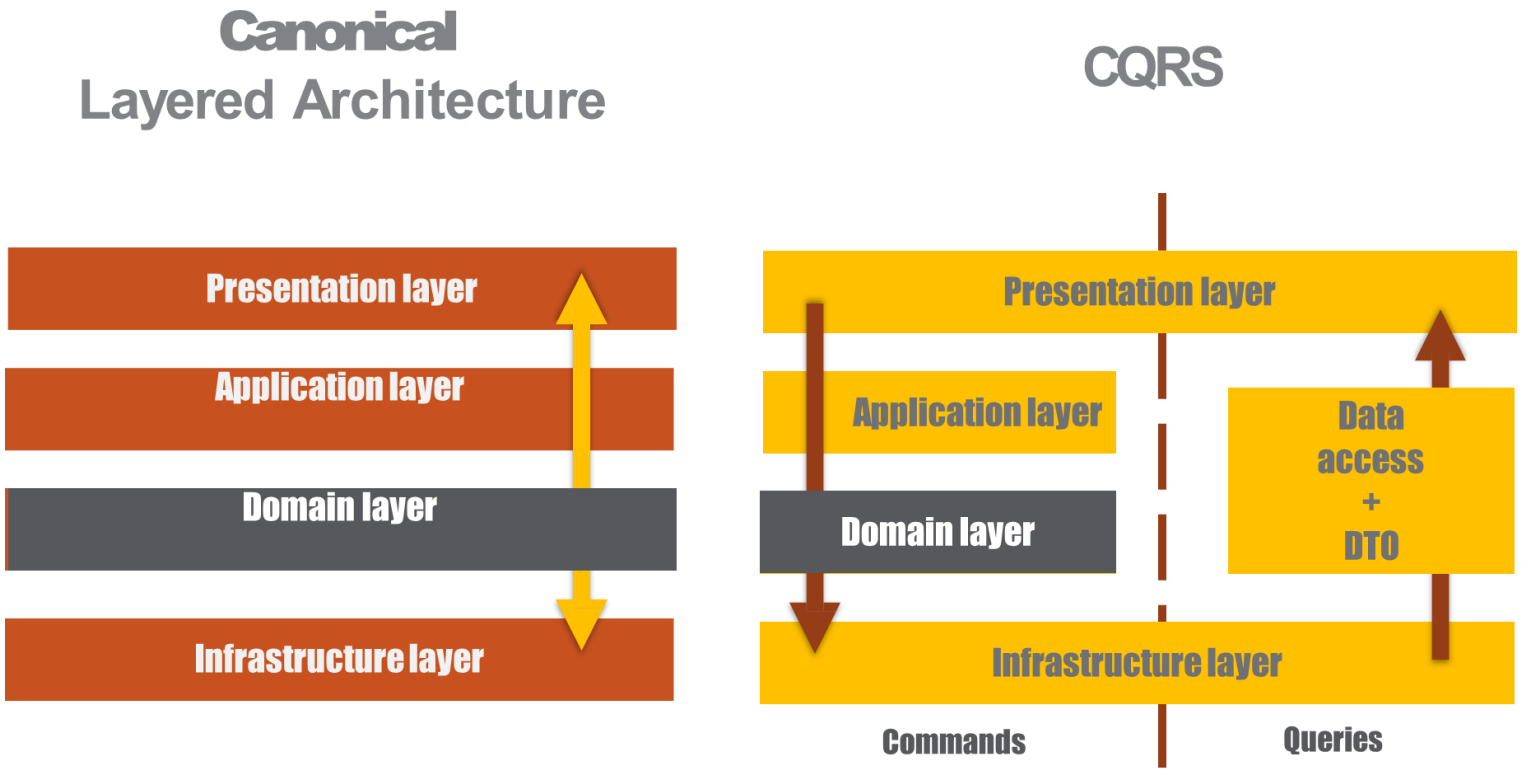


- Command microservice, recibe eventos
  - Se suscribe a una cola
  - Múltiples microservicios de suscripción
  - Uso de modelo de comandos y eventos del repositorio.
- Query microservices, para datos
  - En forma de una función
  - En forma de recuperación de datos
- Proceso para sincronizar el repositorio 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

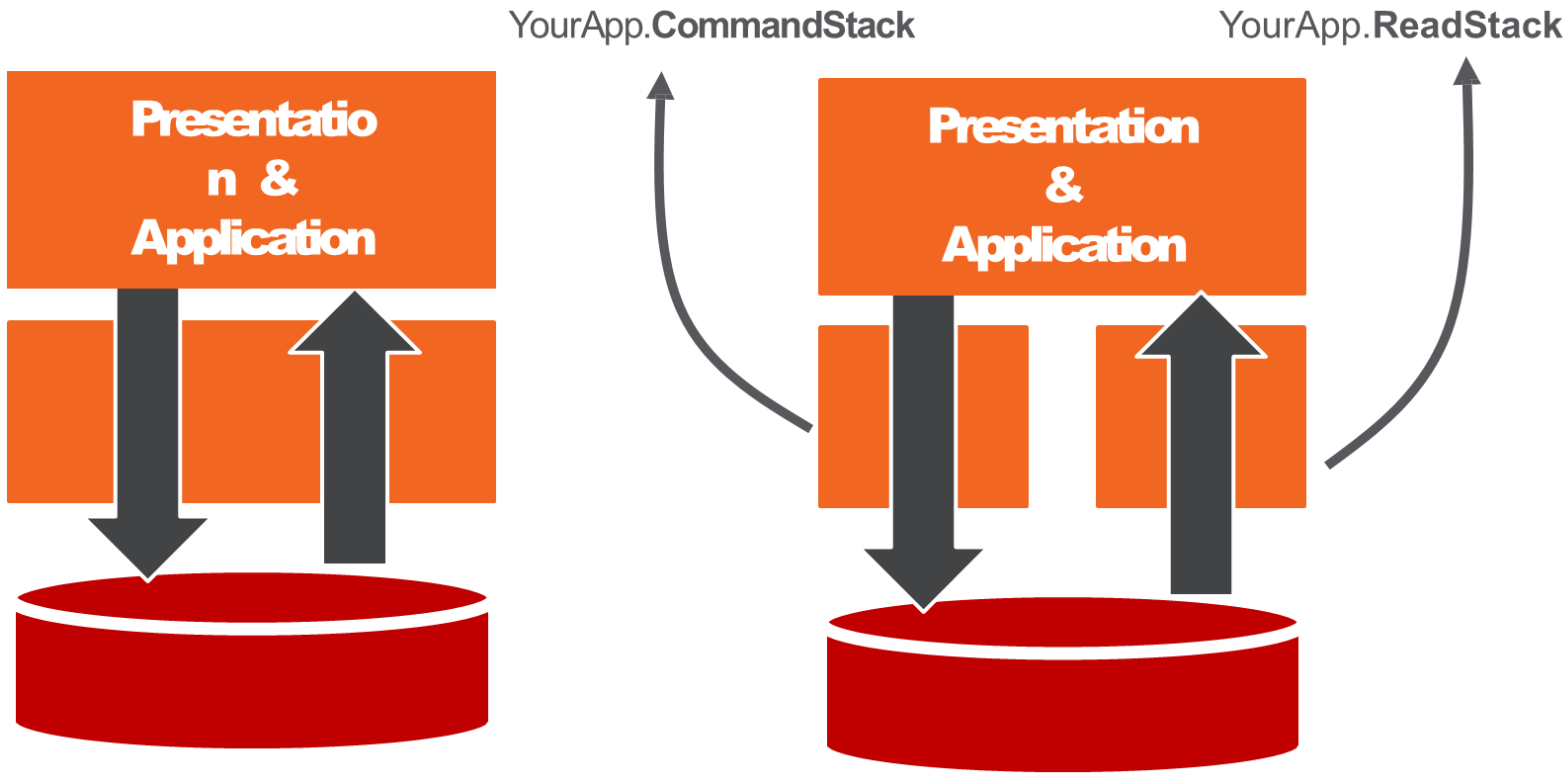

Order Change
(Command)
Order
Service
3
Quantity =23
2
Location =B
1
Order Created
Event
Notification
Message
Broker
Order
(Query)
Order
Service

CQRS

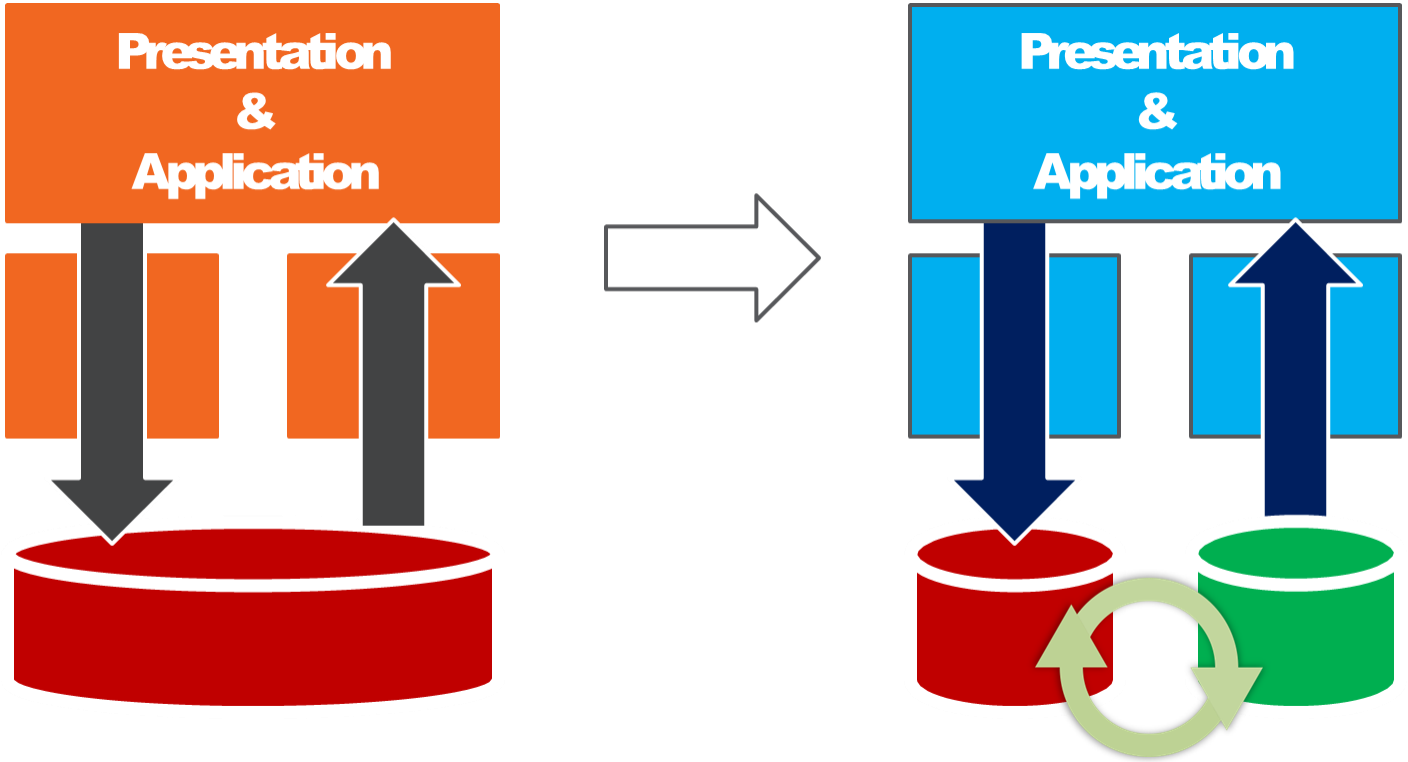

Canonical
Layered Architecture
Presentation layer
Application layer
Domain layer
Infrastructure layer
CQRS
Presentation layer
Application layer
Domain layer
Data
access
+
DTO
Infrastructure layer
Commands
Queries

Sabores de CQRS

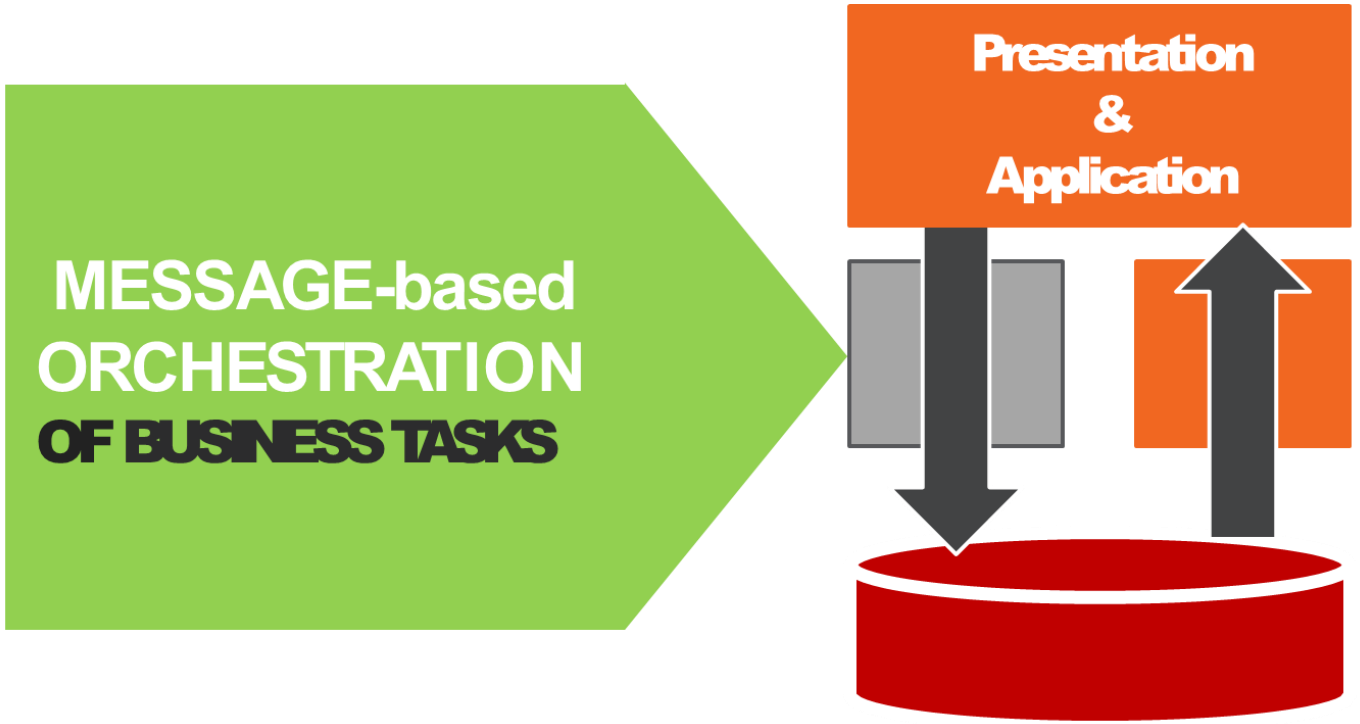
CQRS para aplicaciones CRUD simples



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

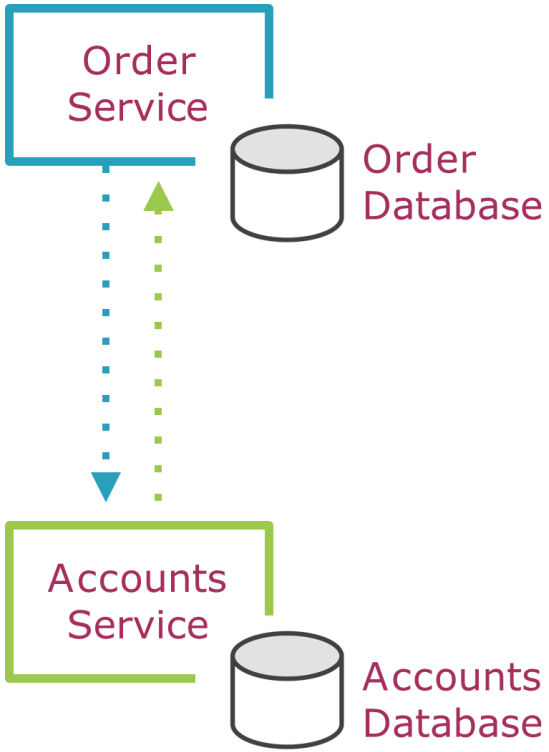
ARQUITECTURA DE MICROSERVICIOS (PERSISTENCIA Y CONSISTENCIA DE DATOS)

Intermedio

DI

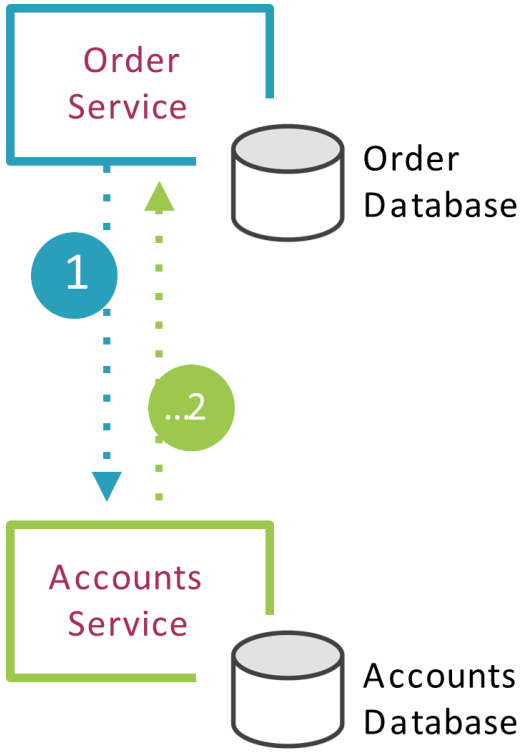
Cómo lograr la consistencia de datos a través de microservicios (consistencia eventual).

Cómo diseñar microservicios basados en API



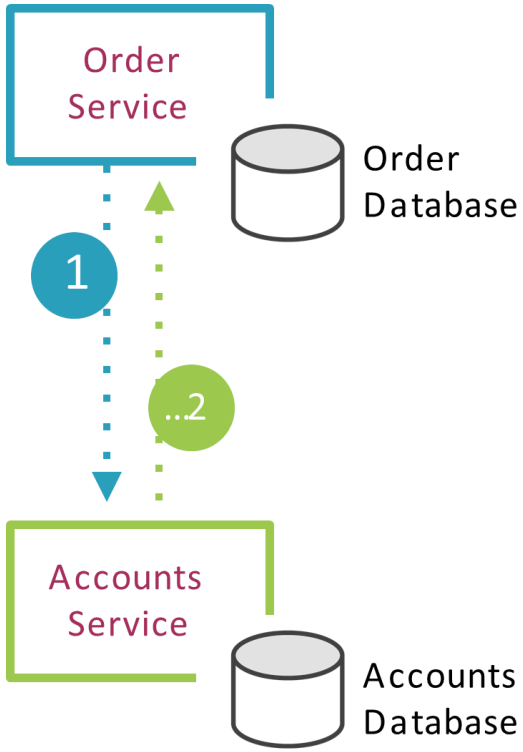
- Microservicios
  - API vs worker
- Arquitectura
  - API vs aplicación
- Como arquitectura
  - Requerimientos funcionales
  - Estilos de arquitectura
  - Patrones de arquitectura

Requerimientos Funcionales



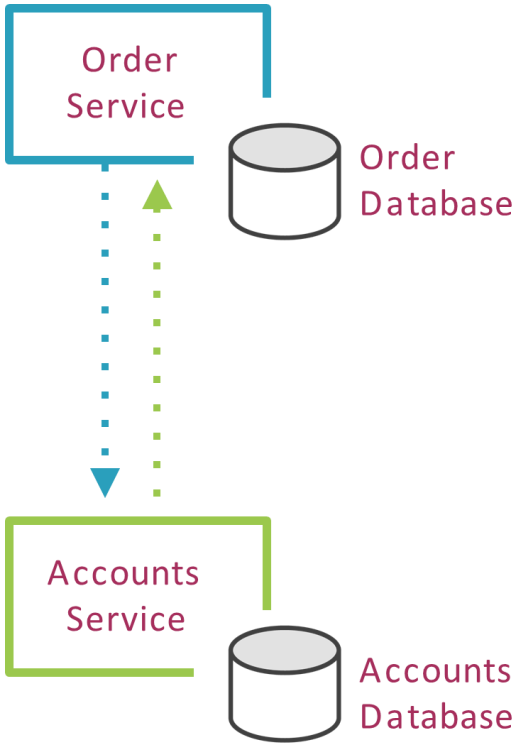
- Principio de microservicios autónomos
  - Débilmente acoplado
  - Independientemente cambiable
  - Desplegable independientemente
  - Contratos de apoyo y honor
  - API agnóstica a la tecnología
  - API sin estado

Opciones de arquitectura



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

Estilos de arquitectura



- Estilos arquitectónicos de API
  - REST Pragmático
  - HATEOS (verdadero REST)
  - RPC
  - SOAP
- Patrones arquitectónicos de API
  - Patrón de fachada
  - Patrón de proxy
  - Patrón de servicio sin estado

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.

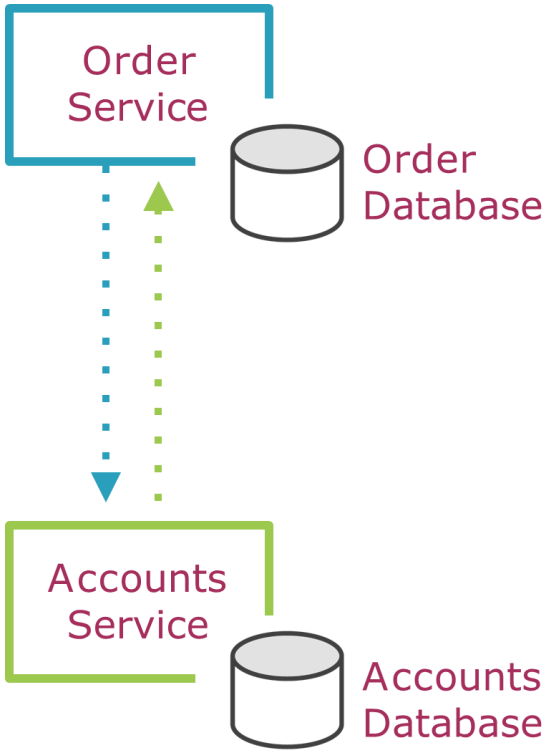
Consistencia Eventual

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).

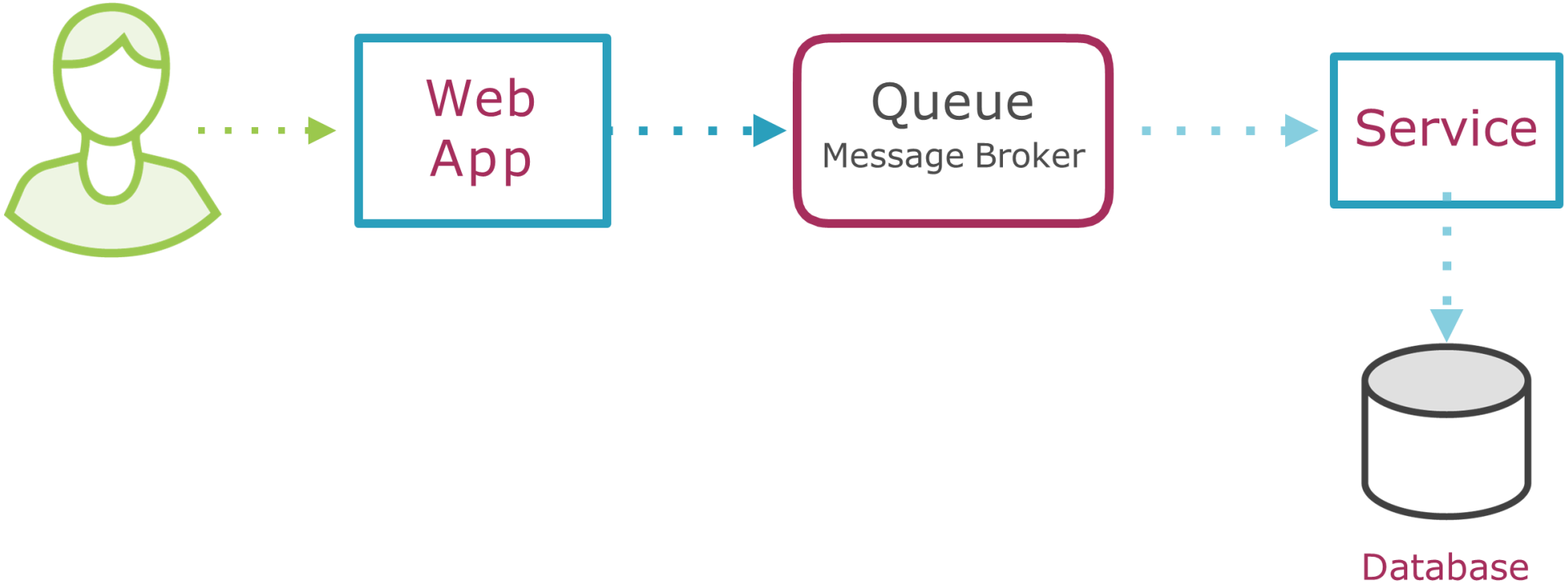
- **La consistencia (Consistency)**, es decir, cualquier lectura recibe como respuesta la escritura más reciente o un error.
- **La disponibilidad (Availability)**, es decir, cualqueir petición recibe una respuesta no errónea, pero sin la garantía de que contenga la escritura más reciente.
- **La tolerancia 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

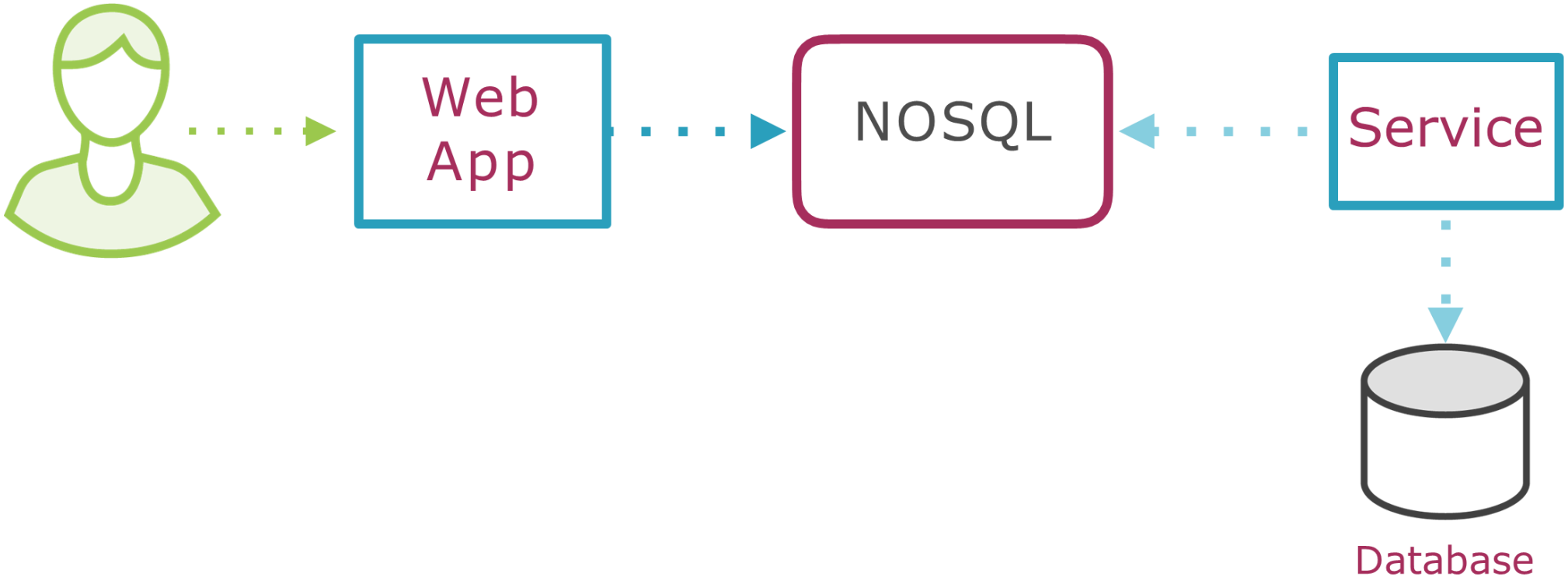


- Los datos eventualmente serán consistentes
  - BASE
  - BASE vs ACID
- Disponibilidad sobre consistencia
  - Evitar el bloqueo de recursos
  - Ideal para tareas de larga duración.
  - Preparado para inconsistencias
  - Condiciones de carrera
- Replicación de datos
- Basado en eventos
  - Transacción/acciones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

Consistencia Eventual



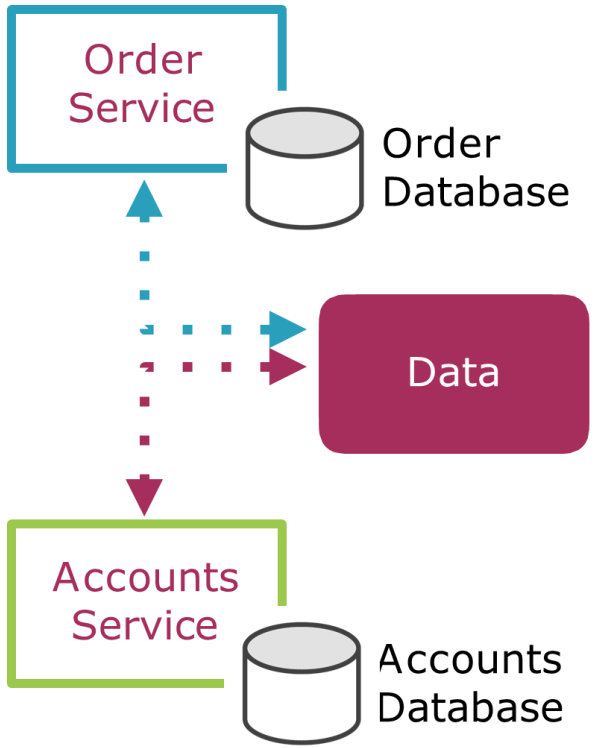
Consistencia Eventual



Implementación del patrón SAGA.

ARQUITECTURA DE MICROSERVICIOS (PERSISTENCIA Y CONSISTENCIA DE DATOS)

Intermedio

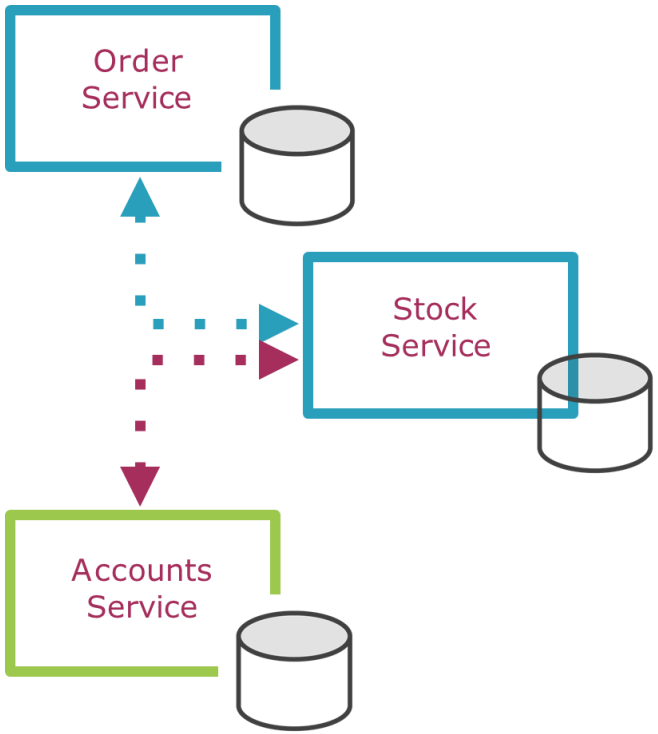


Introducción

- 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?

ARQUITECTURA DE MICROSERVICIOS (PERSISTENCIA Y CONSISTENCIA DE DATOS)

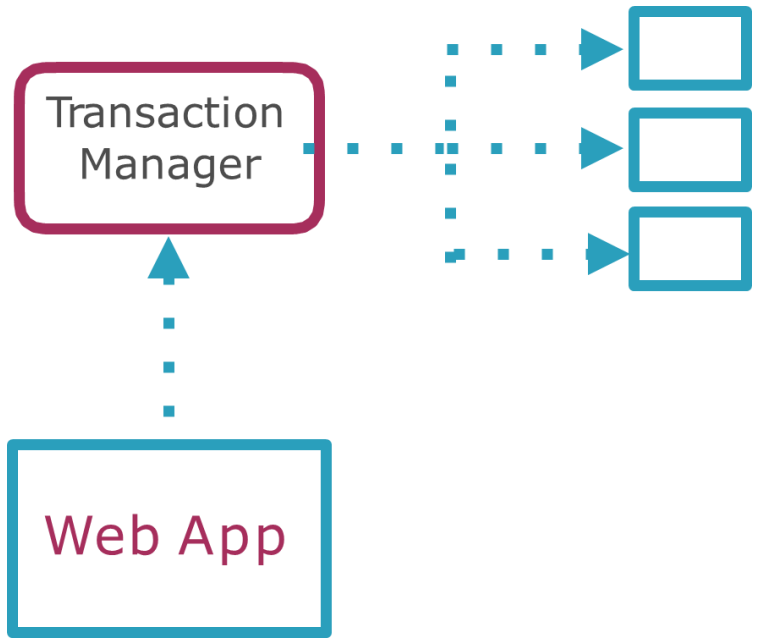
Intermedio



Opciones

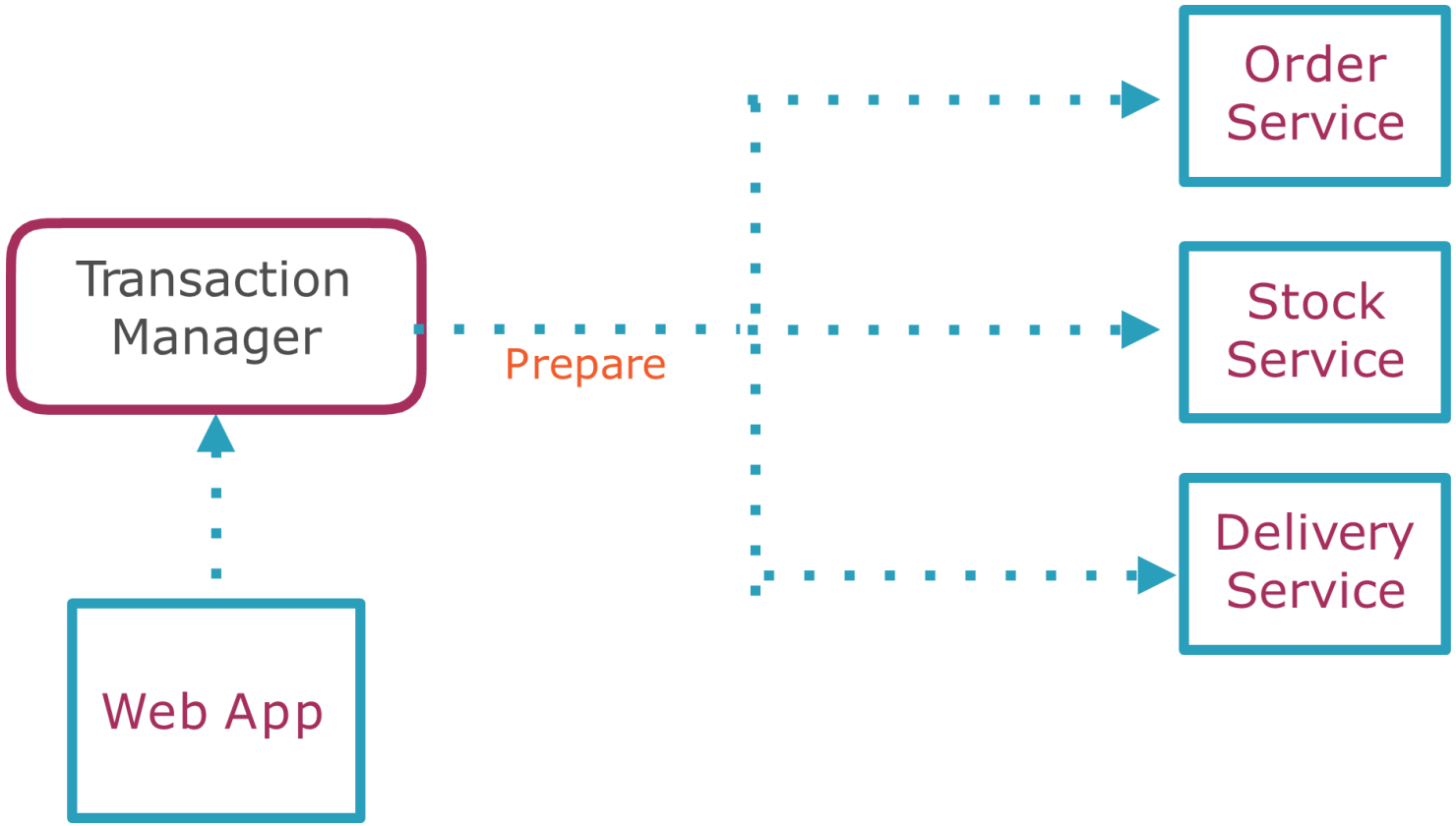
- Transacciones ACID tradicionales
  - Atomicidad, consistencia, aislamiento y durabilidad.
- Patrón de confirmación de dos fases (2PC)
  - ACID es obligatorio
  - Teorema CAP: elección de consistencia
- Patrón SAGA
  - Atomicidad por disponibilidad y consistencia
- Patrón de consistencia eventual
 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

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

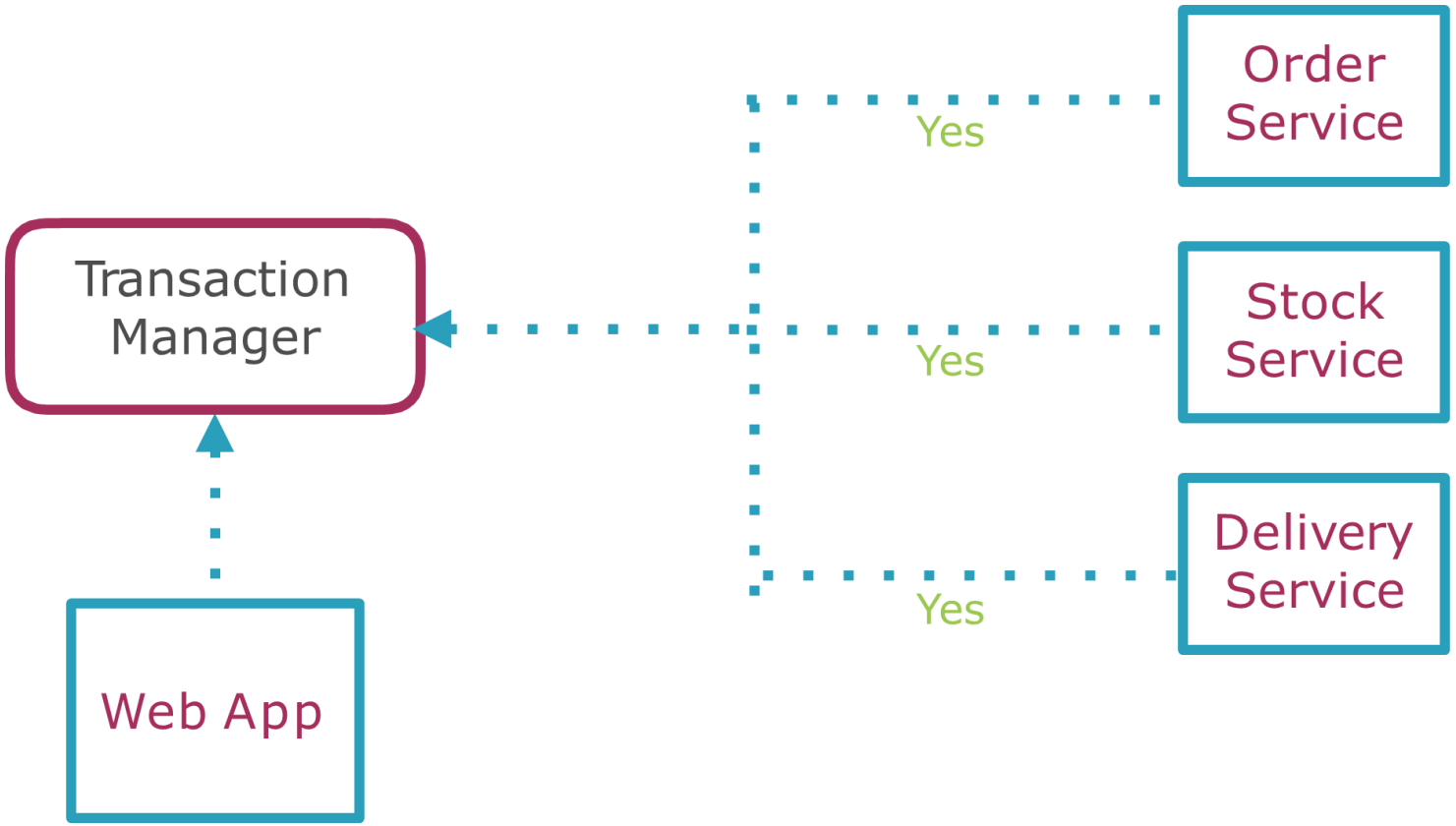


- Patrón para transacciones distribuidas
  - El administrador de transacciones maneja las transacciones
- Fase de preparación
  - El gestor de transacciones notifica el inicio de preparación.
- Fase de confirmación
  - El gestor de transacciones recibe las confirmaciones.
  - Gestor de transacciones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

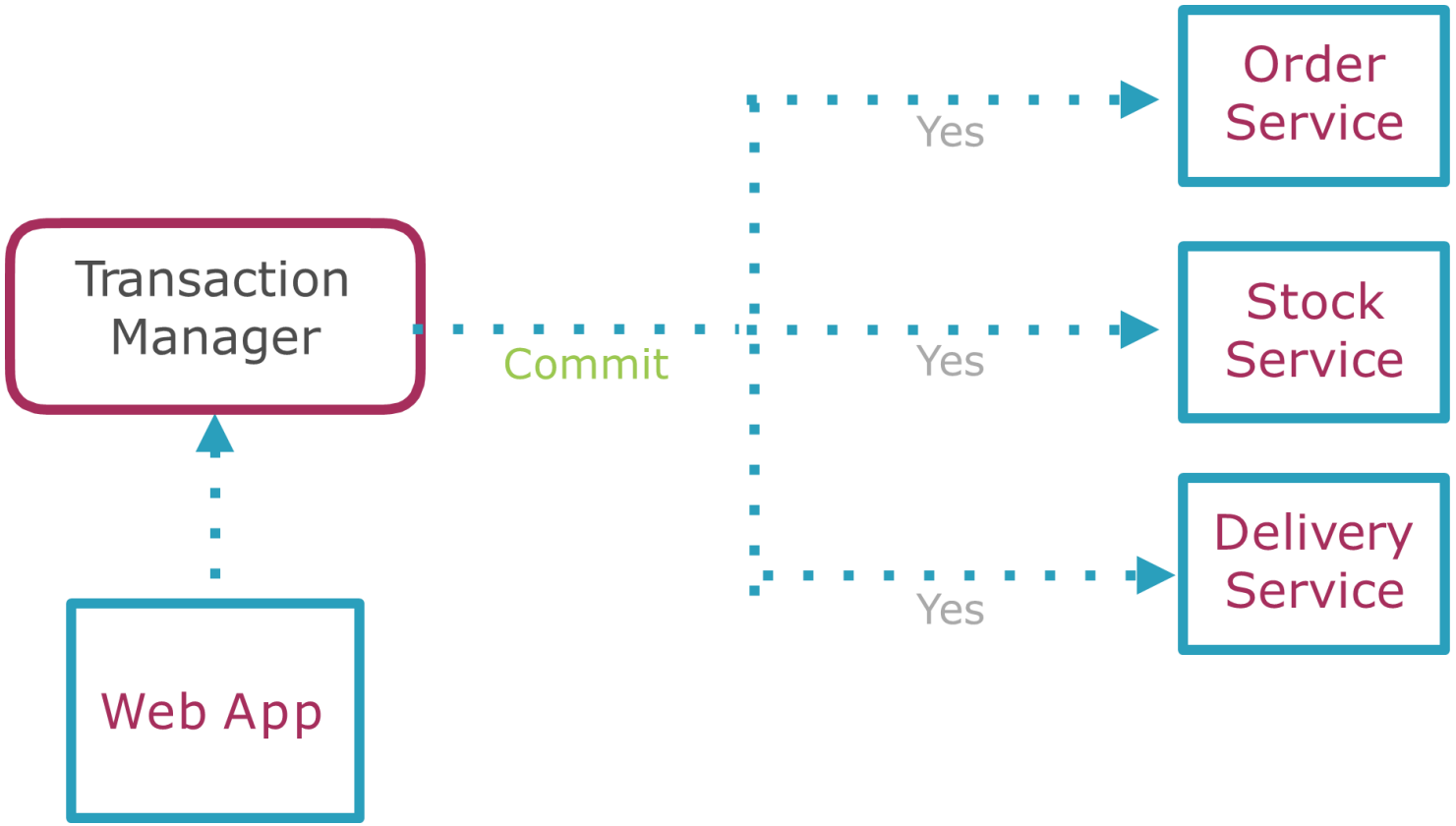
Confirmación de 2 fases – Fase de preparación



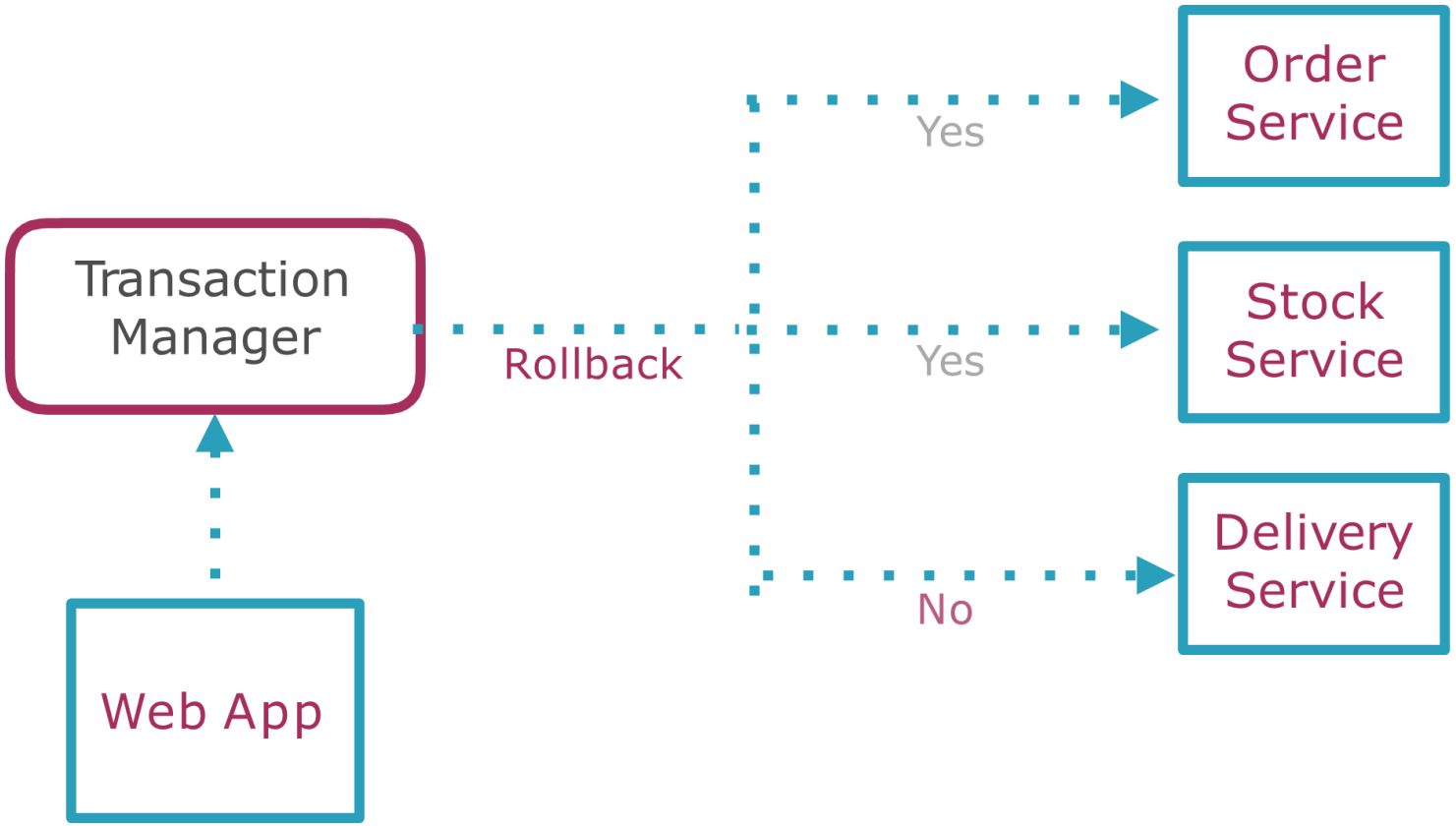
Confirmación de 2 fases – Fase de Confirmación



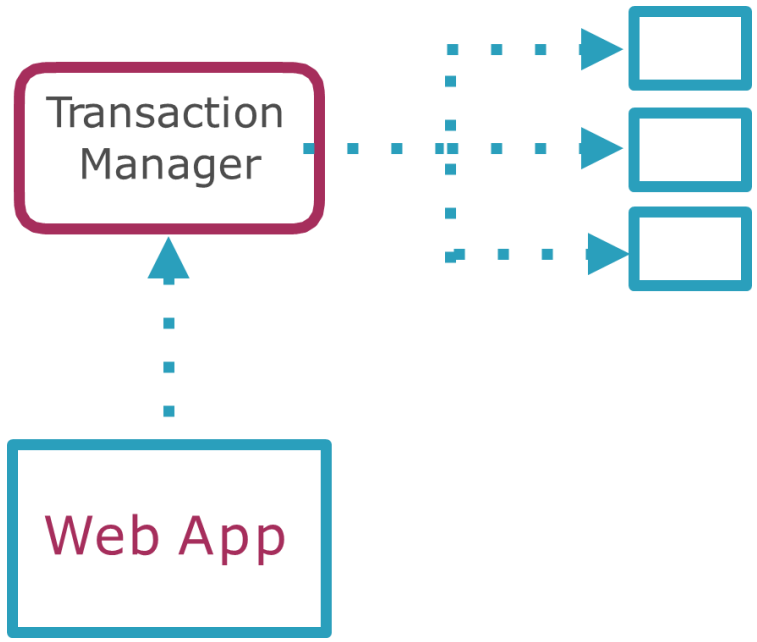
Confirmación de 2 fases – Commit



Confirmación de 2 fases – Rollback

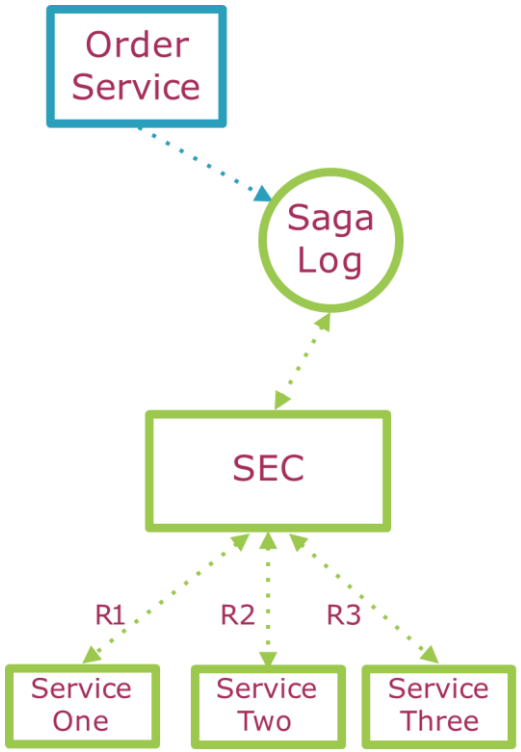


Confirmación de 2 fases (Two Phase Commit / 2PC)



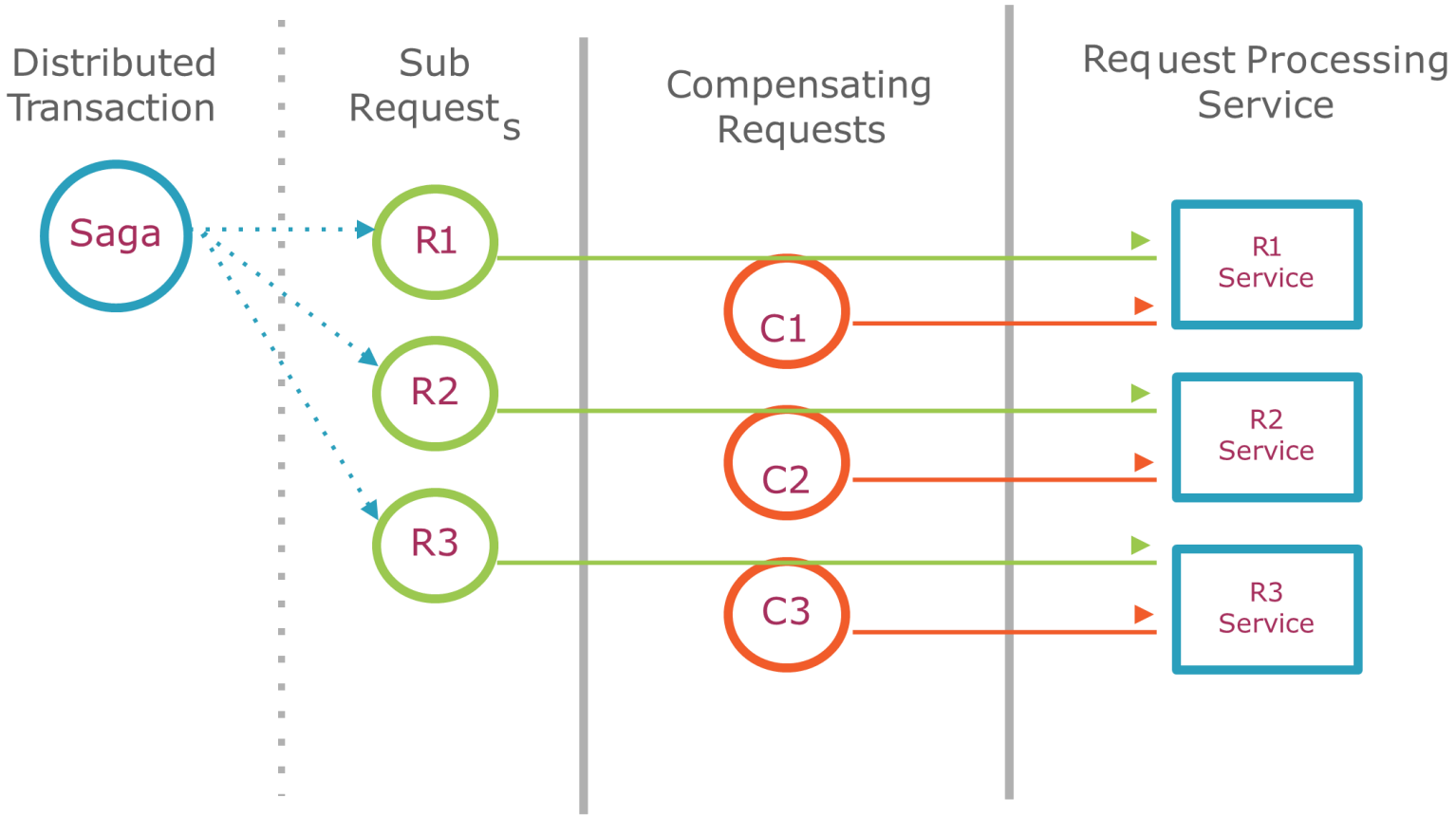
- Advertencias
  - Confianza en un administrador de transacciones
  - Sin respuesta de confirmación
  - Fallar después de una confirmación
  - Las transacciones pendientes bloquean recursos
  - Evitar implementaciones personalizadas
  - Tiene problemas de escalado
  - Rendimiento reducido
  - Antipatrón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

Patrón Saga

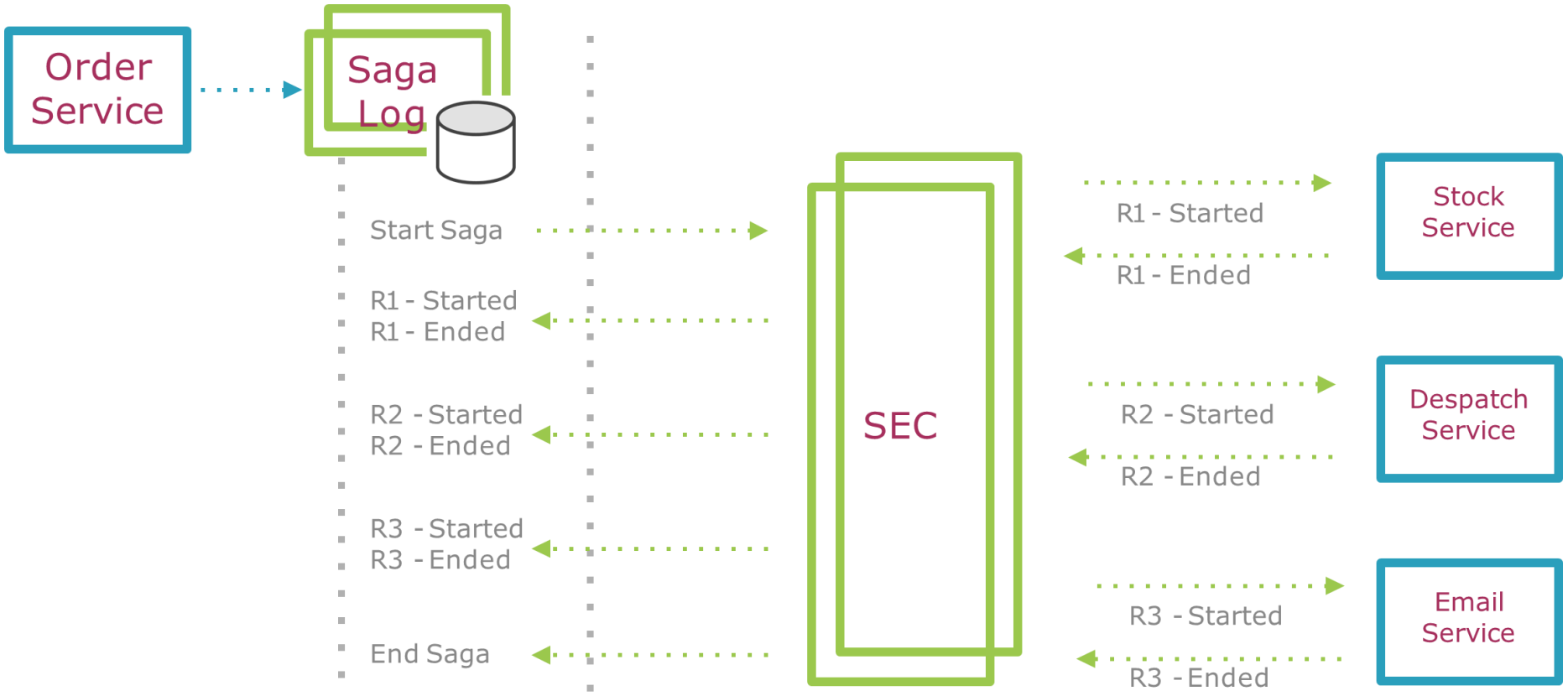


- Reemplaza una transacción distribuida con una saga
  - Divide la transacción en muchas solicitudes
  - Rastrea cada solicitud
  - ACID: compromiso de atomicidad
  - Descrito por primera vez en 1987
- También es un patrón de gestión de fallas
  - Qué hacer cuando falla un servicio
  - Compensar solicitudes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

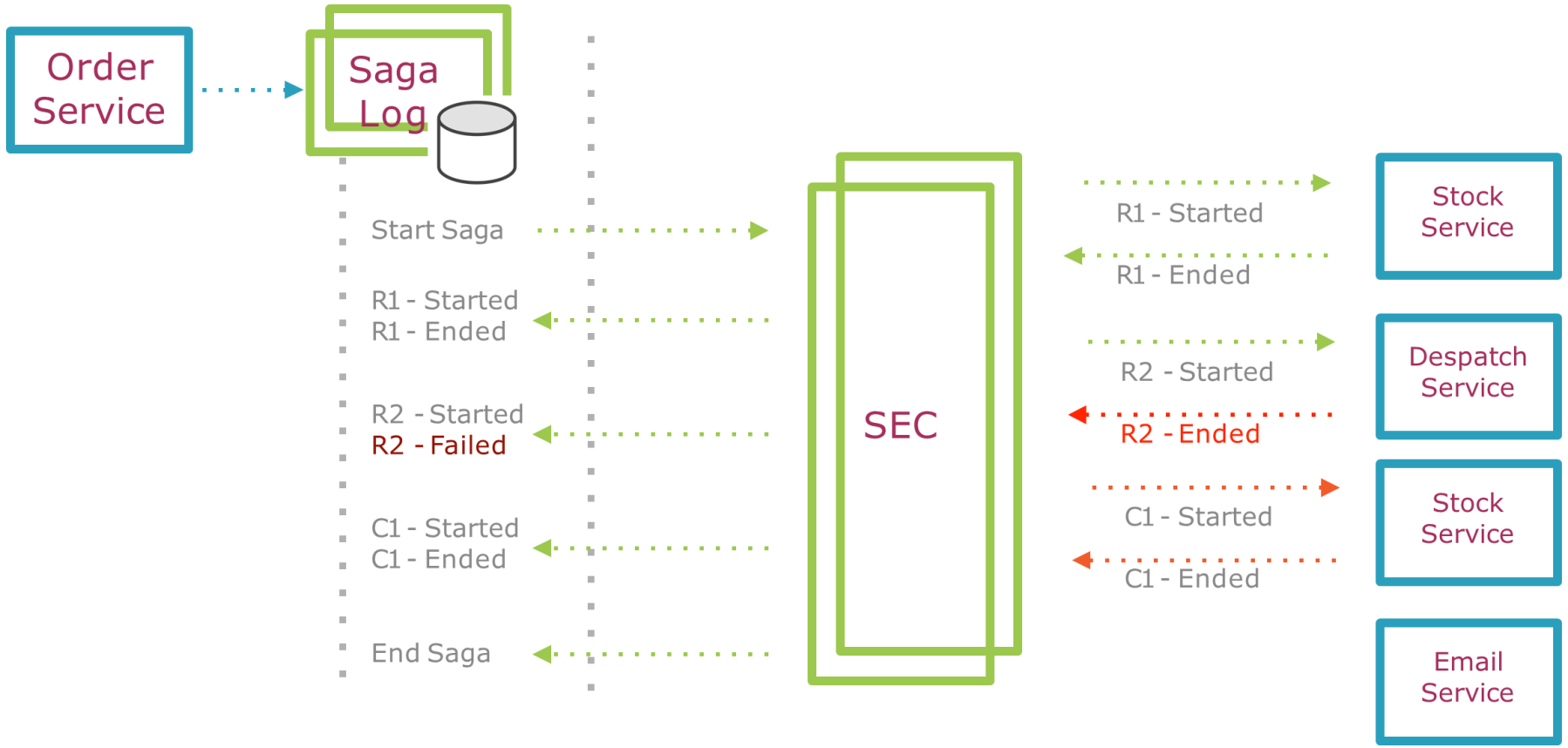
Patrón Saga


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

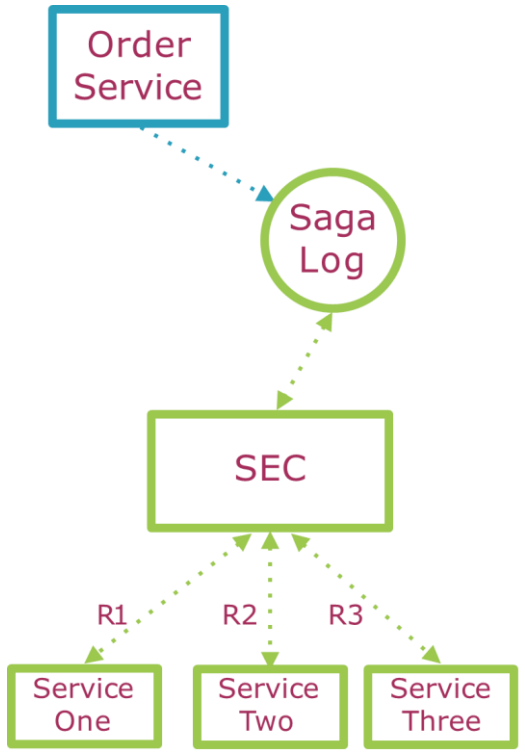
Saga exitoso



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


GALAXY
TRAINING