

Sesión 03

Persistencia y consistencia de datos – Parte I

Instructor:

ERICK ARÓSTEGUI

earostegui@galaxy.edu.pe



6 NET

MICROSERVICES
ARCHITECTURE

ÍNDICE

01

Patrones de gobierno de datos

02

Patrón CQRS a un microservicio

03

Consistencia eventual en microservicios

04

Implementando CQRS (SQL y NoSQL)

05

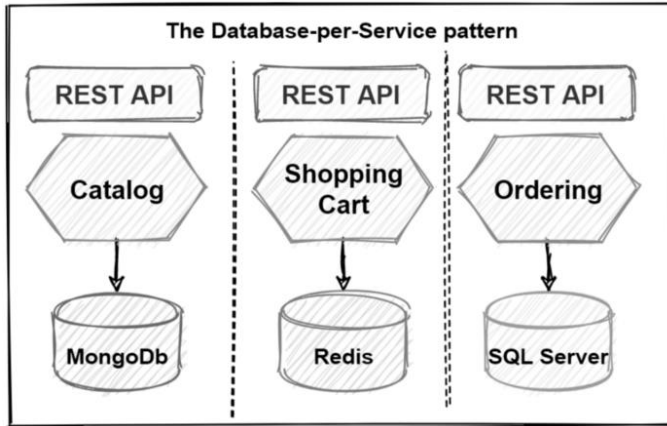
Recomendaciones para su implementación

01

Patrones de gobierno de datos

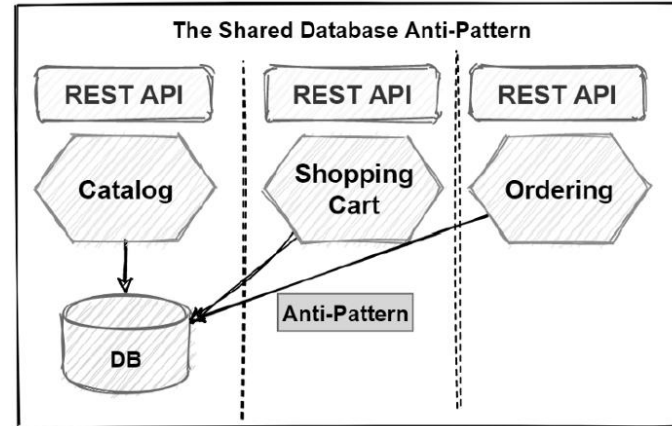


→ Patrones de gobierno de datos



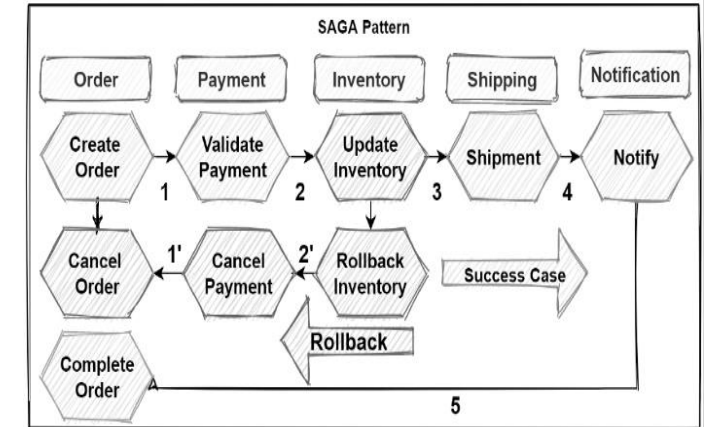
Base de datos por Servicio

Cada microservicio administra sus datos. Los desarrolladores usan API's bien definidas para facilitar la comunicación entre las bases de datos de dos o más microservicios.



Base de datos compartida

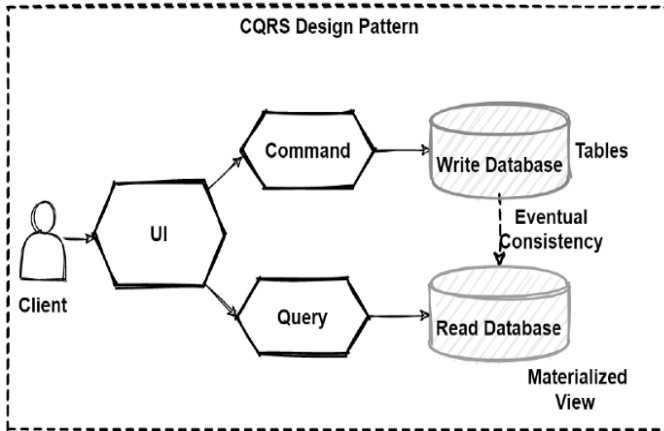
Permite que varios servicios accedan y almacenen datos en la misma base de datos. Si usa la base de datos compartida para varios microservicios, entonces caerá en un **antipatrón** y debe evitar estos enfoques.



Patrón Saga

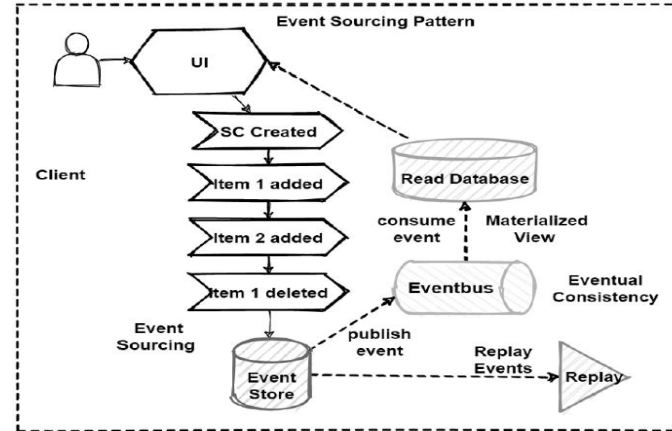
Saga es una secuencia de transacciones locales en la que el resultado de cada transacción depende de un evento anterior. Si alguna transacción falla, saga realiza una serie de transacciones de compensación.

→ Patrones de gobierno de datos



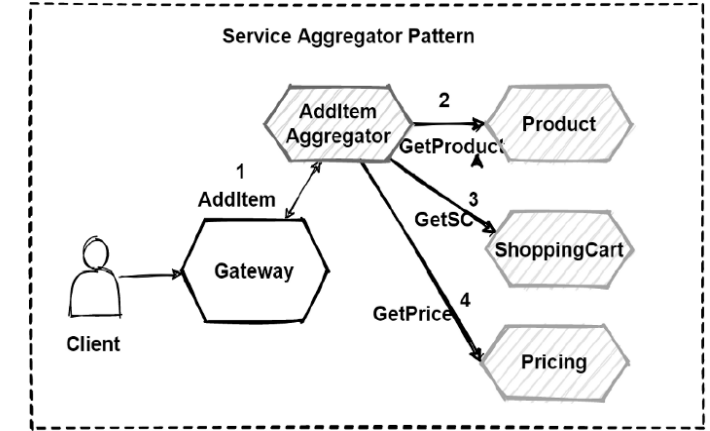
Patrón CQRS

Se proporciona a la base de datos de comandos y consultas separadas para realizar mejor la consulta de varios microservicios. CQRS ofrece un mayor rendimiento y una mejor escalabilidad de los microservicios.



Patrón Event Sourcing

Proporciona acumular eventos y agregarlos en una secuencia de eventos en bases de datos. De esta manera podemos reproducir a cierto punto de los eventos. Este patrón está muy bien usando con **cqrs** y patrones de **saga**.



Composición de APIs

Utiliza compositores de API para acceder a los conjuntos de datos de los servicios. Después de obtener los datos, el patrón utiliza una "unión en memoria" para emparejar dos servicios antes de enviarlos al consumidor.

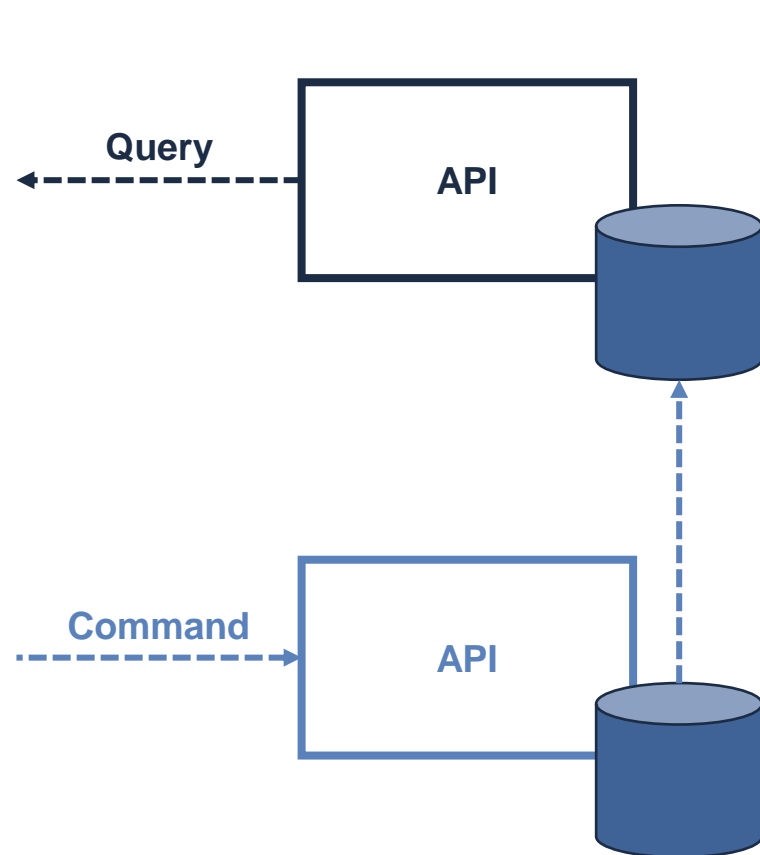
02



Patrón CQRS a un microservicio

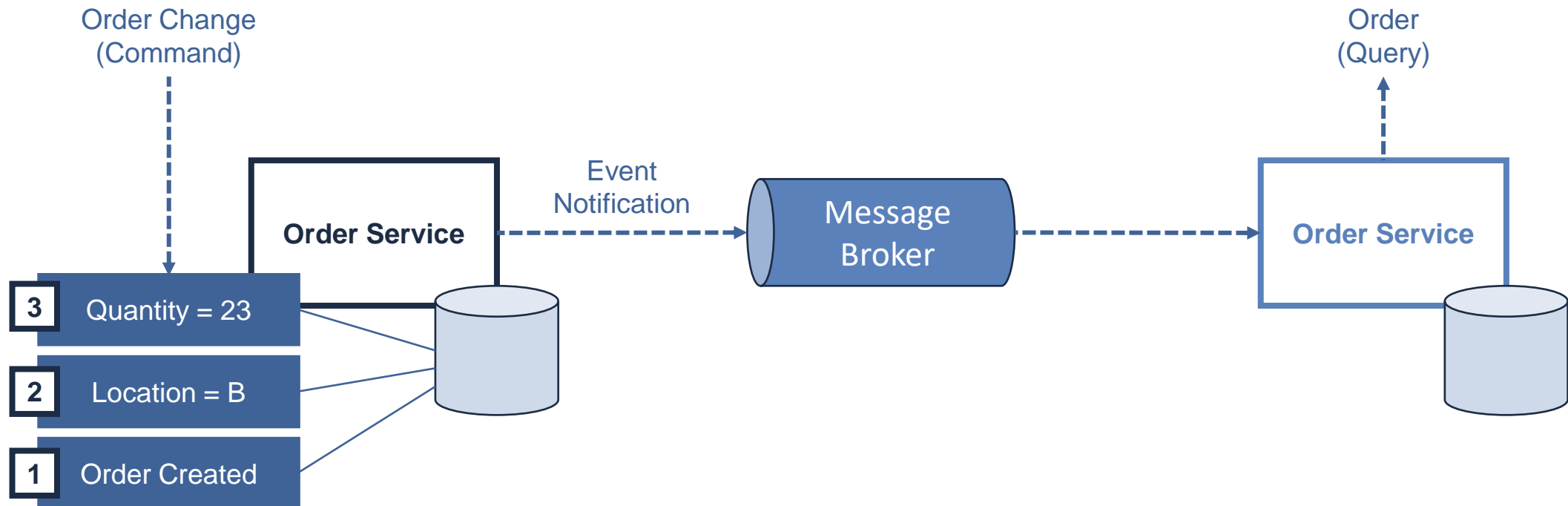
→ Patrón CQRS a un microservicio

Command Query Response Segregation

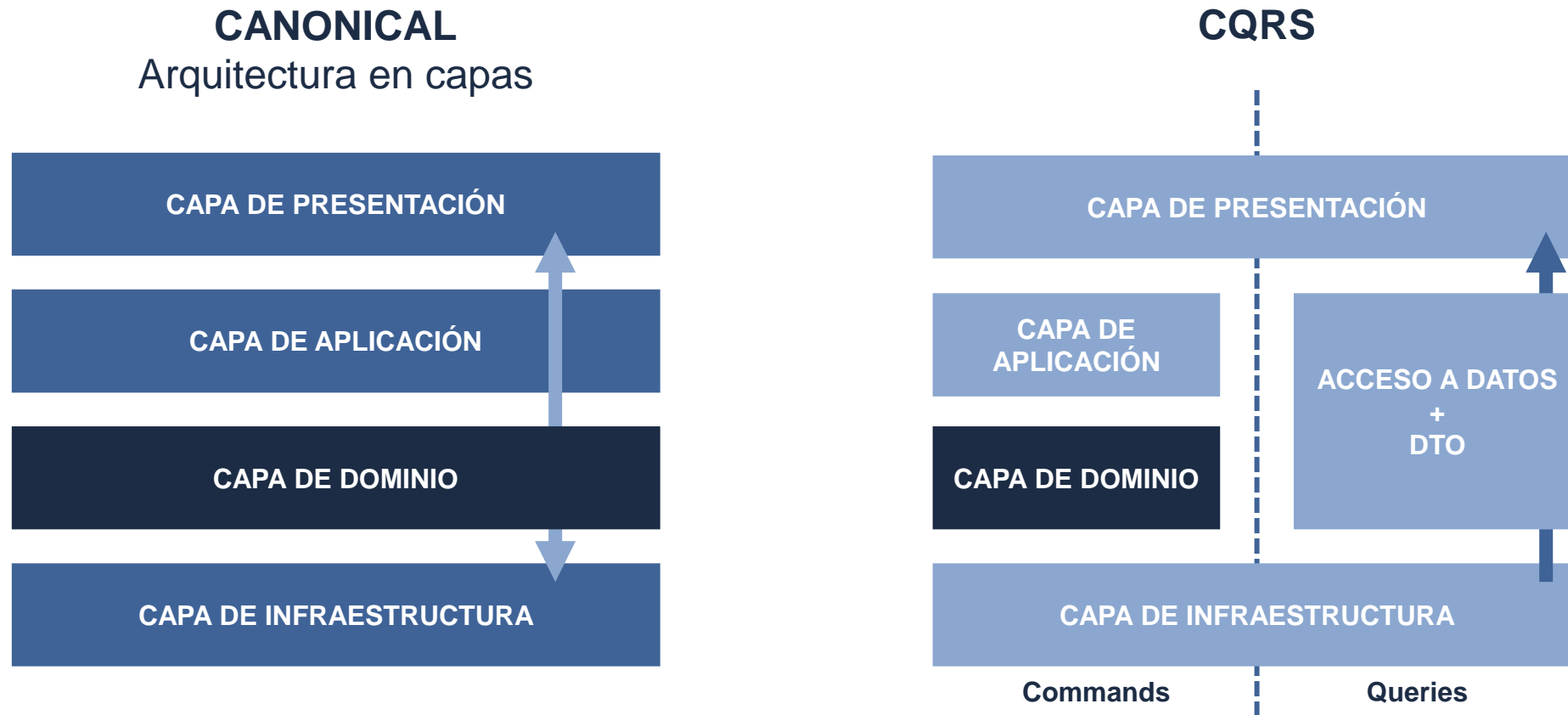


- CQRS
 - Modelos de comando (**Command**) y/o servicios
 - Modelos de consulta (**Query**) y/o servicios
- ¿Por qué?
 - Separación de responsabilidades.
 - Notificaciones de eventos manejadas por comando (**Escritura**)
 - Informes/funciones manejadas por consulta (**Lectura**)
 - Separación de tecnologías.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

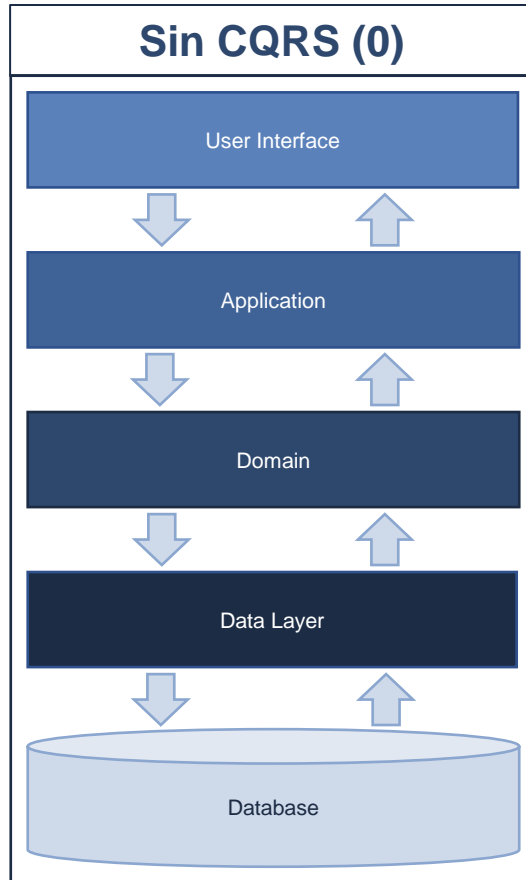
Command Query Response Segregation



Command Query Response Segregation

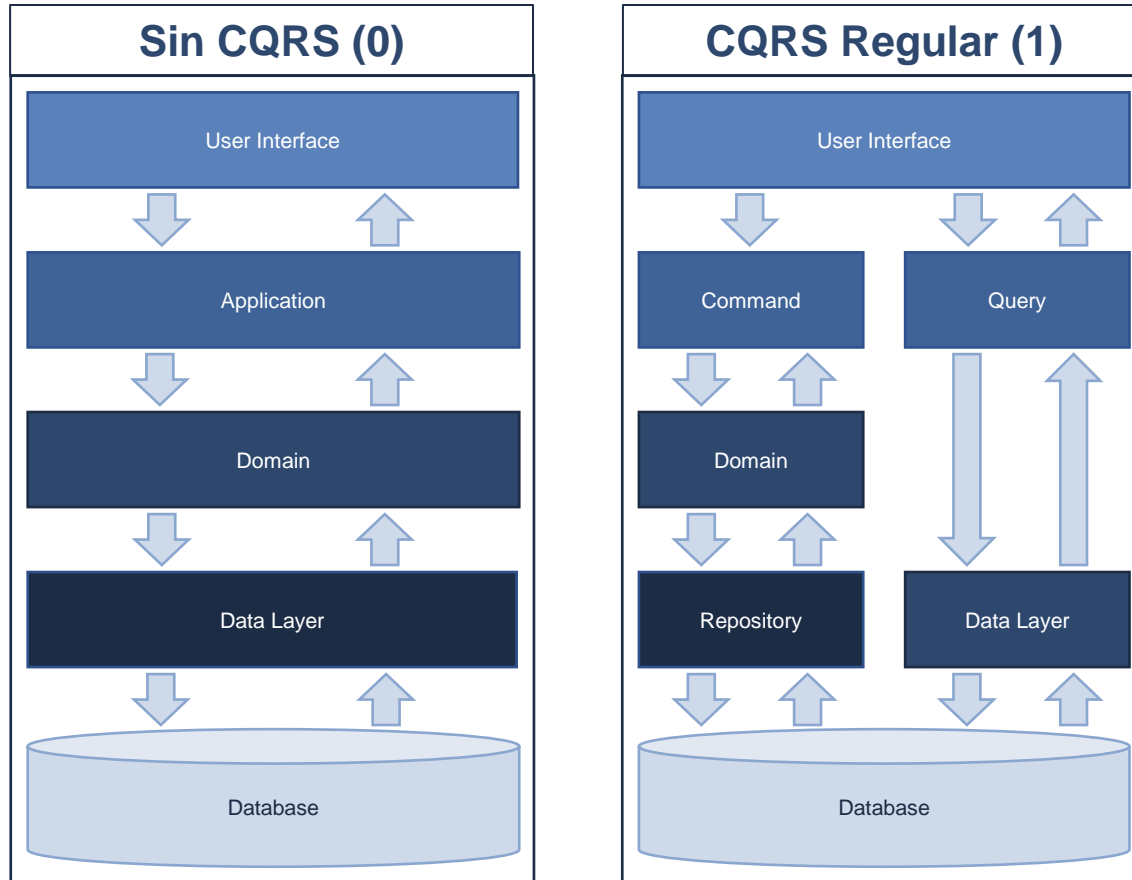


Tipos de CQRS



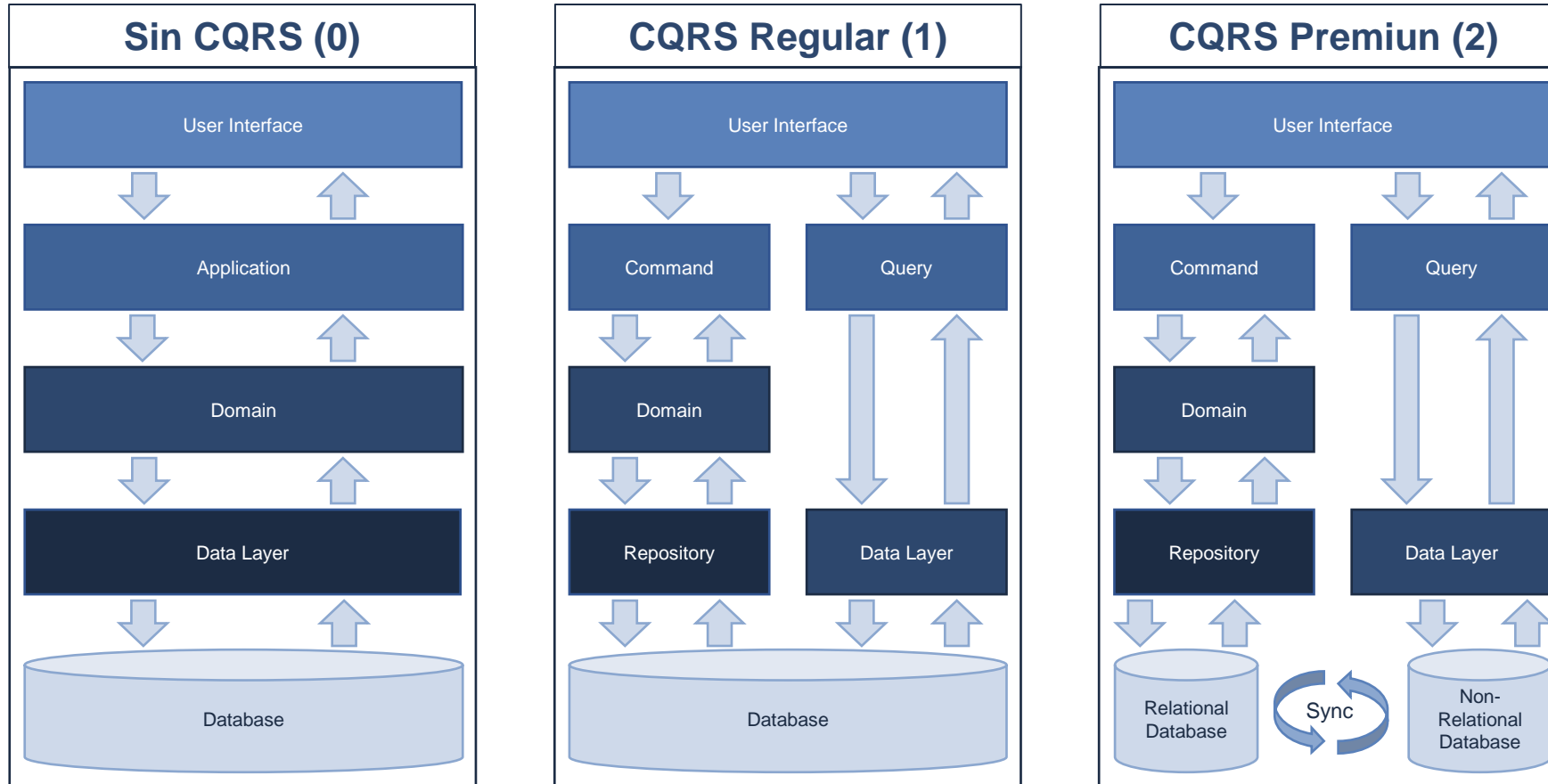
→ Patrón CQRS a un microservicio

Tipos de CQRS



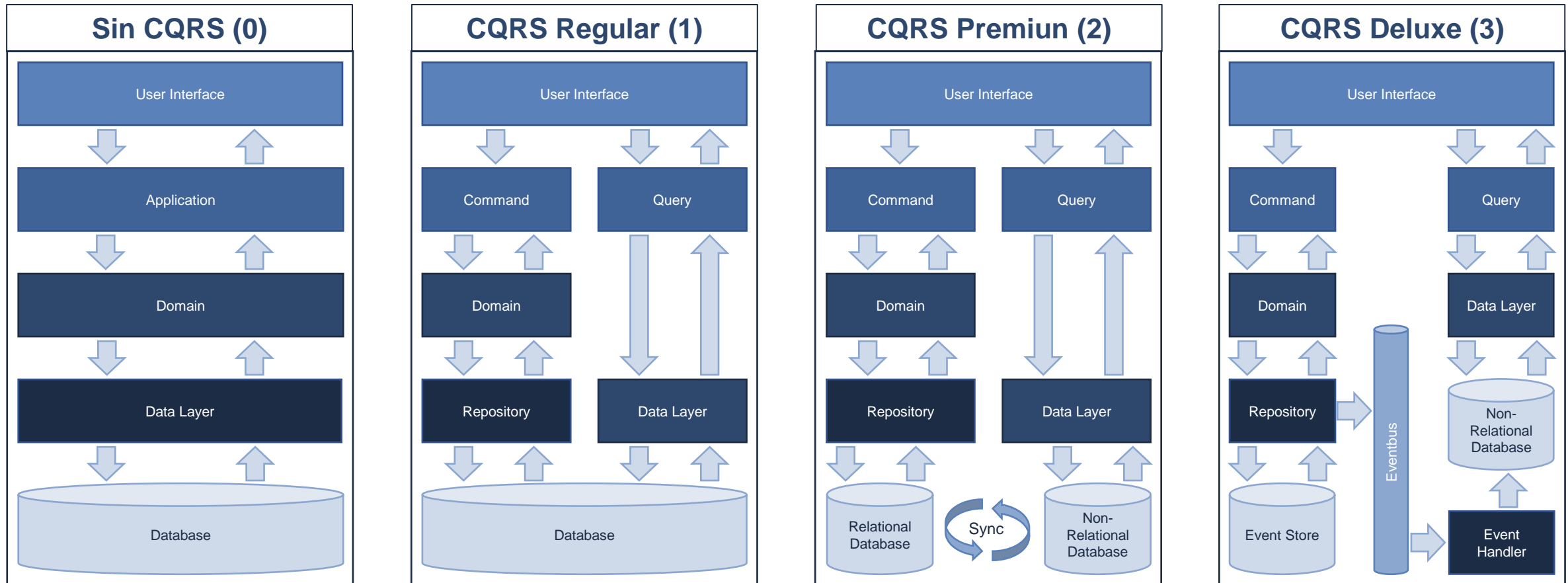
→ Patrón CQRS a un microservicio

Tipos de CQRS



→ Patrón CQRS a un microservicio

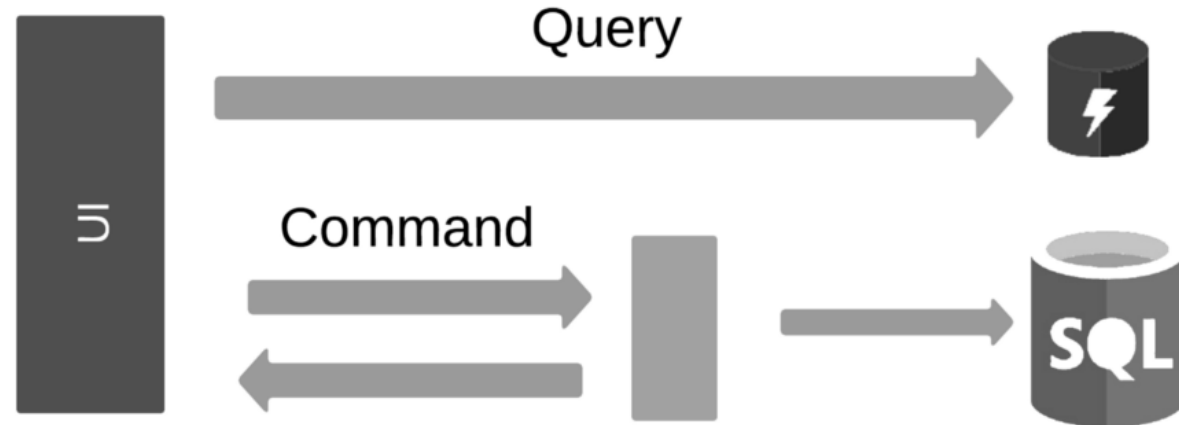
Tipos de CQRS



Patrón CQRS a un microservicio



DEMO



CQRS (Command Query Responsibility Segregation)

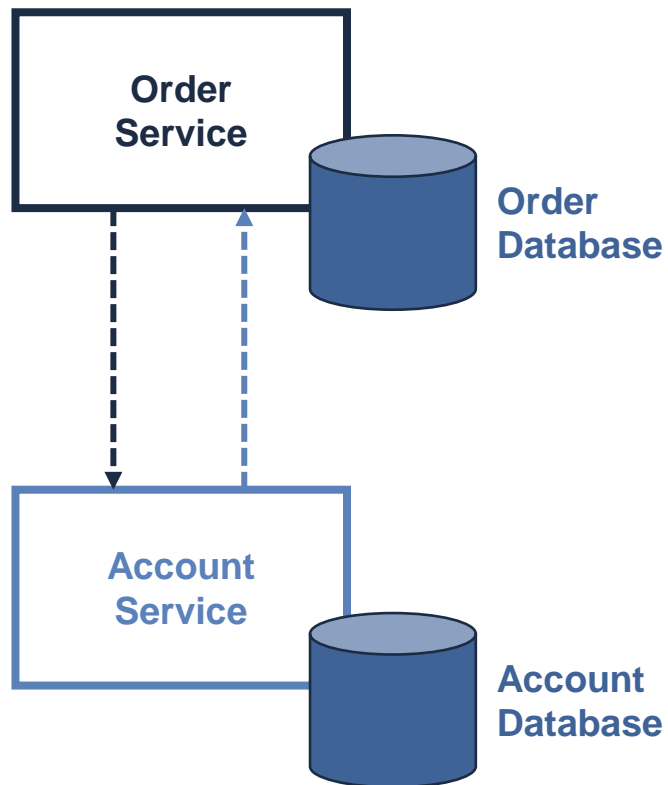
03



Consistencia eventual en microservicios

→ Consistencia eventual en microservicios

Cómo diseñar microservicios basados en API

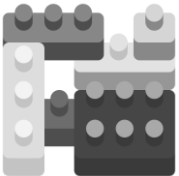


- Microservicios
 - **API** vs worker
- Arquitectura
 - **API** vs aplicación
- Como arquitectura
 - Requerimientos funcionales
 - **Estilos de arquitectura**
 - **Patrones de arquitectura**

Requerimientos Funcionales de MSA



Débilmente acoplado



Independientemente
cambiable



Desplegable
independientemente



Contratos de apoyo y
honor

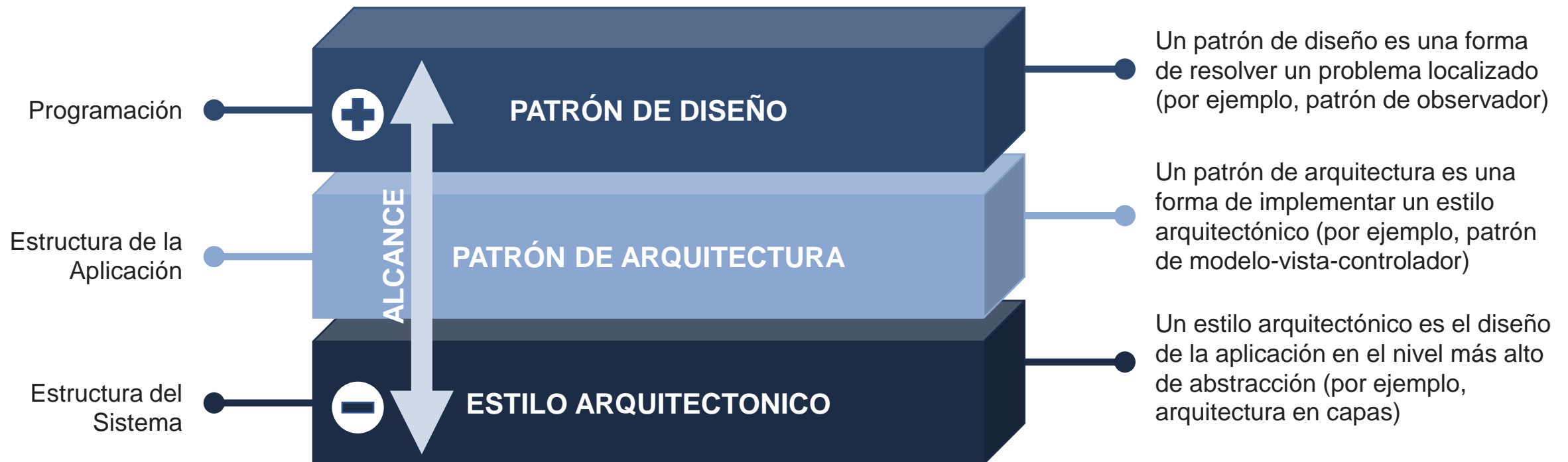


API agnóstica a la
tecnología



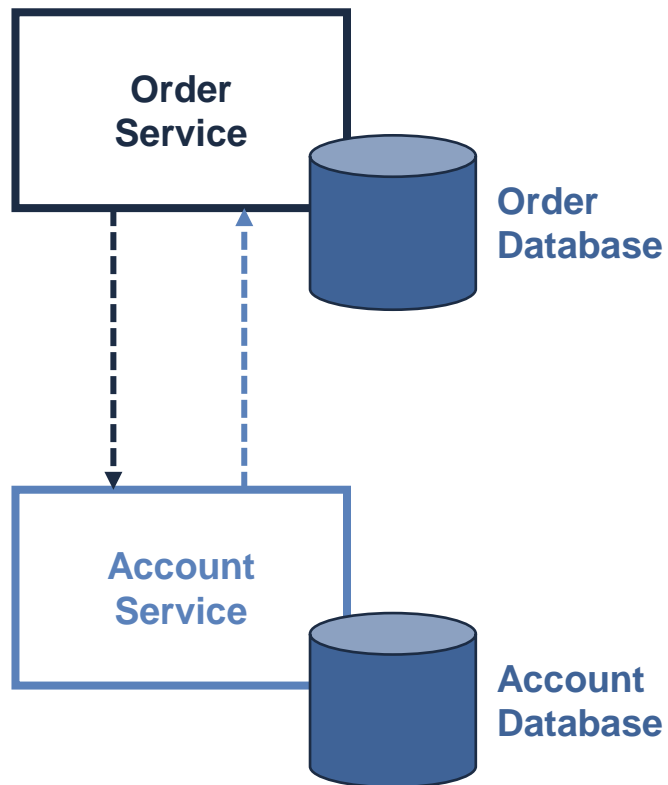
Diseño para fallos

Cómo diseñar microservicios basados en API



→ Consistencia eventual en microservicios

Opciones de arquitectura



Patrones de arquitectura API

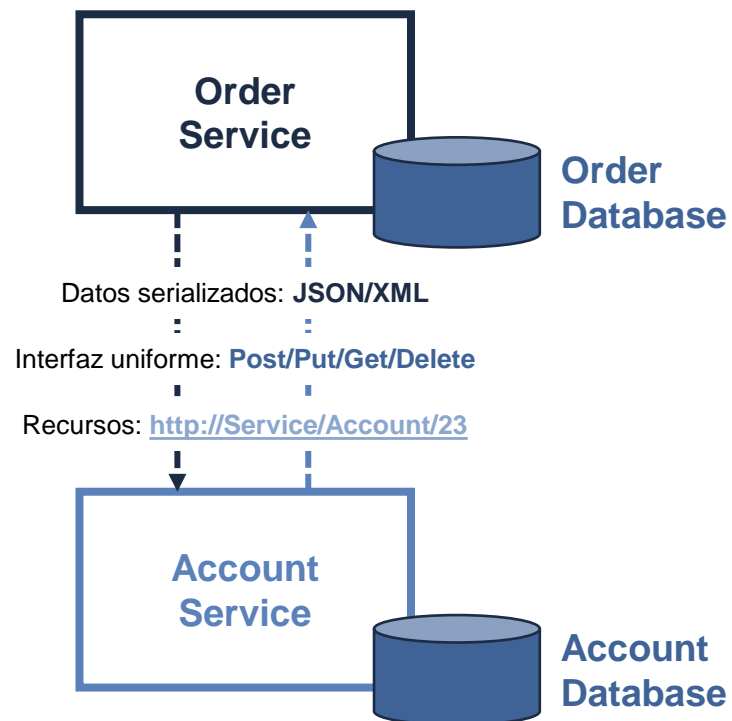
- Facade pattern
- Proxy pattern
- Stateless service pattern

Estilo de arquitectura API

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

→ Consistencia eventual en microservicios

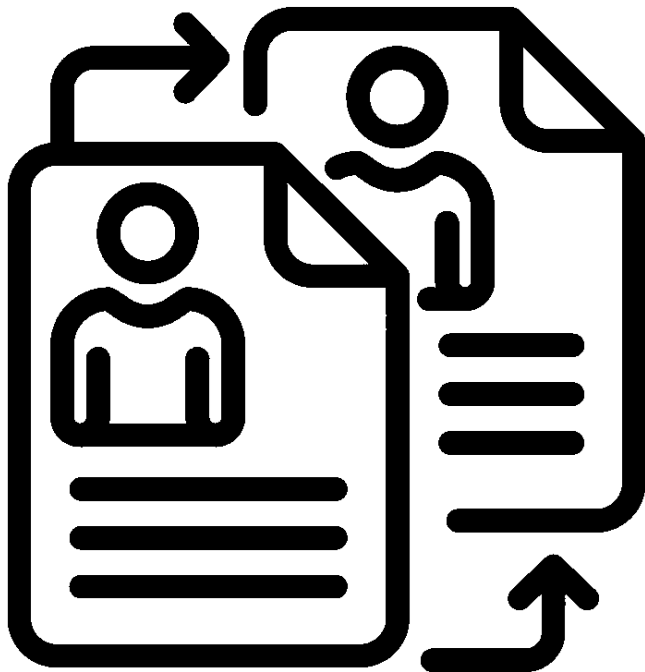
Estilo de arquitectura REST



Qué es REST?

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

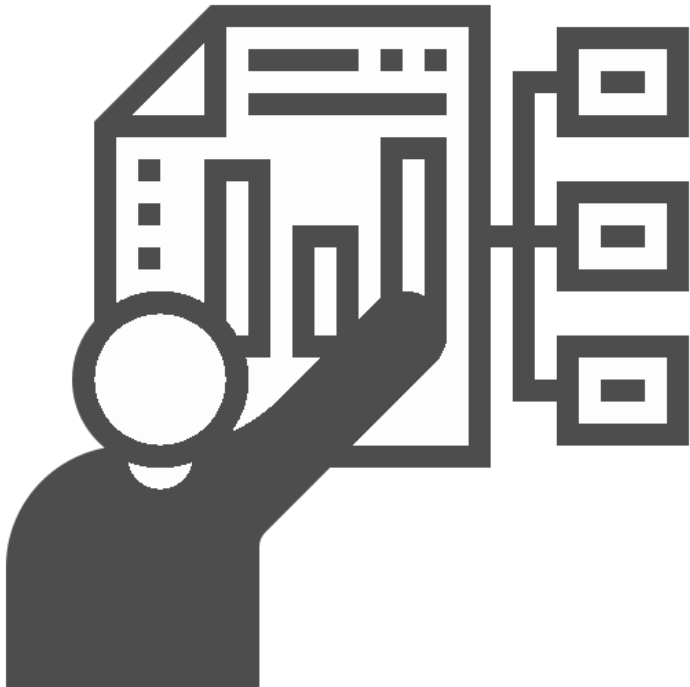
Consistencia Eventual



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.

Consistencia Eventual



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

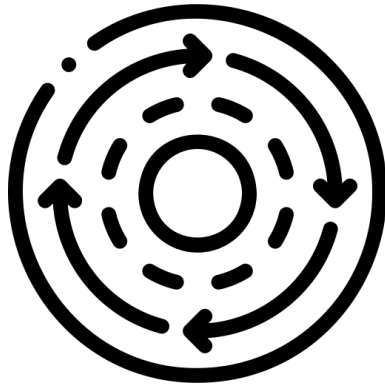


La mayoría de las aplicaciones necesitan bloqueos sin conexión para **evitar transacciones de base de datos de larga duración**.

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, porque las empresas a menudo valoran más la disponibilidad

Consistencia Eventual - Teorema de CAP



La consistencia (Consistency)

Cualquier lectura recibe como respuesta la escritura más reciente o un error.



La disponibilidad (Availability)

Cualquier petición recibe una respuesta no errónea, pero sin la garantía de que contenga la escritura más reciente.

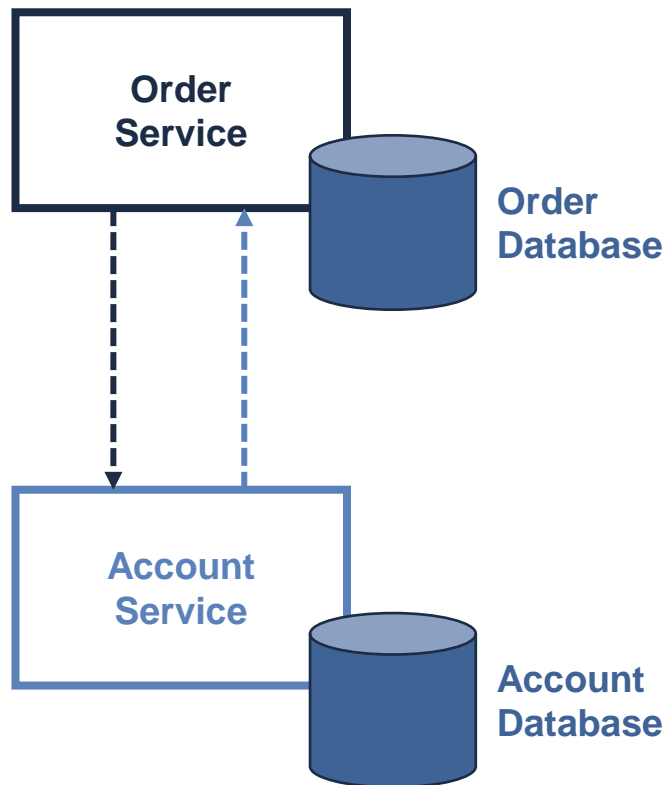


Tolerancia al Particionado (Partition Tolerance)

el sistema sigue funcionando incluso si un número arbitrario de mensajes son descartados (o retrasados) entre nodos de la red

→ Consistencia eventual en microservicios

Consistencia Eventual



Los datos eventualmente serán consistentes

- **BASE**
- BASE vs ACID

Disponibilidad sobre consistencia

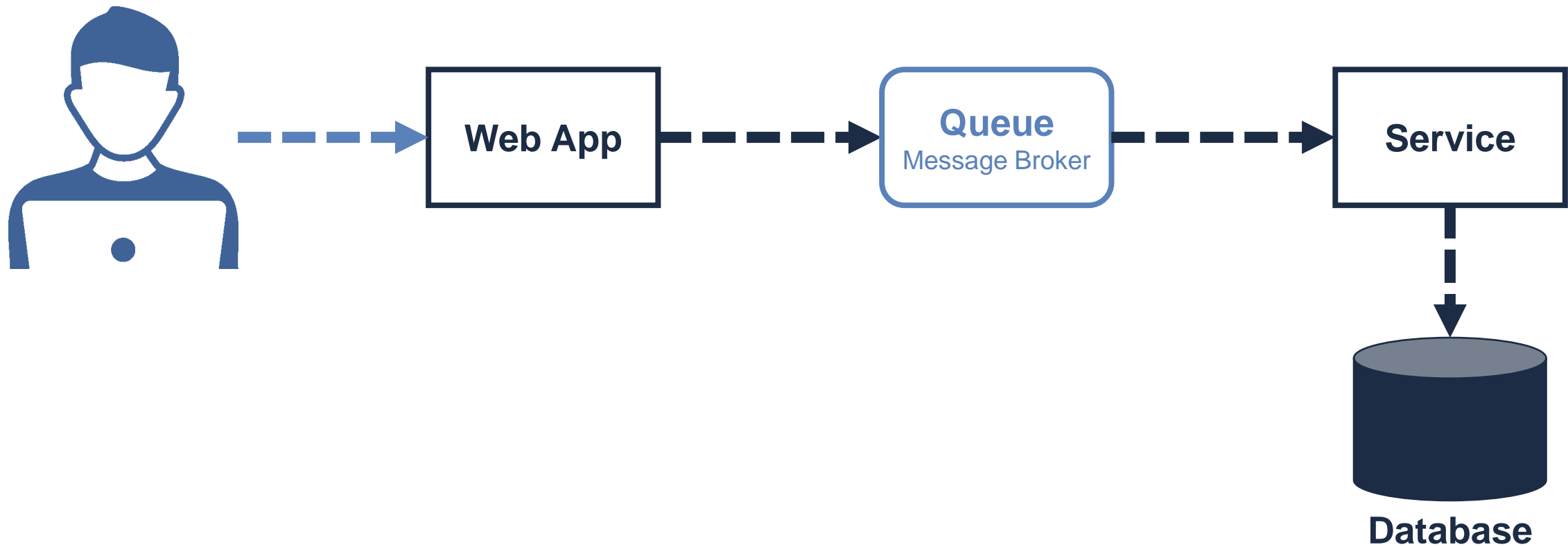
- **Evitar el bloqueo de recursos**
- Ideal para tareas de larga duración.
- **Preparado para inconsistencias**
- Condiciones de carrera

Replicación de datos

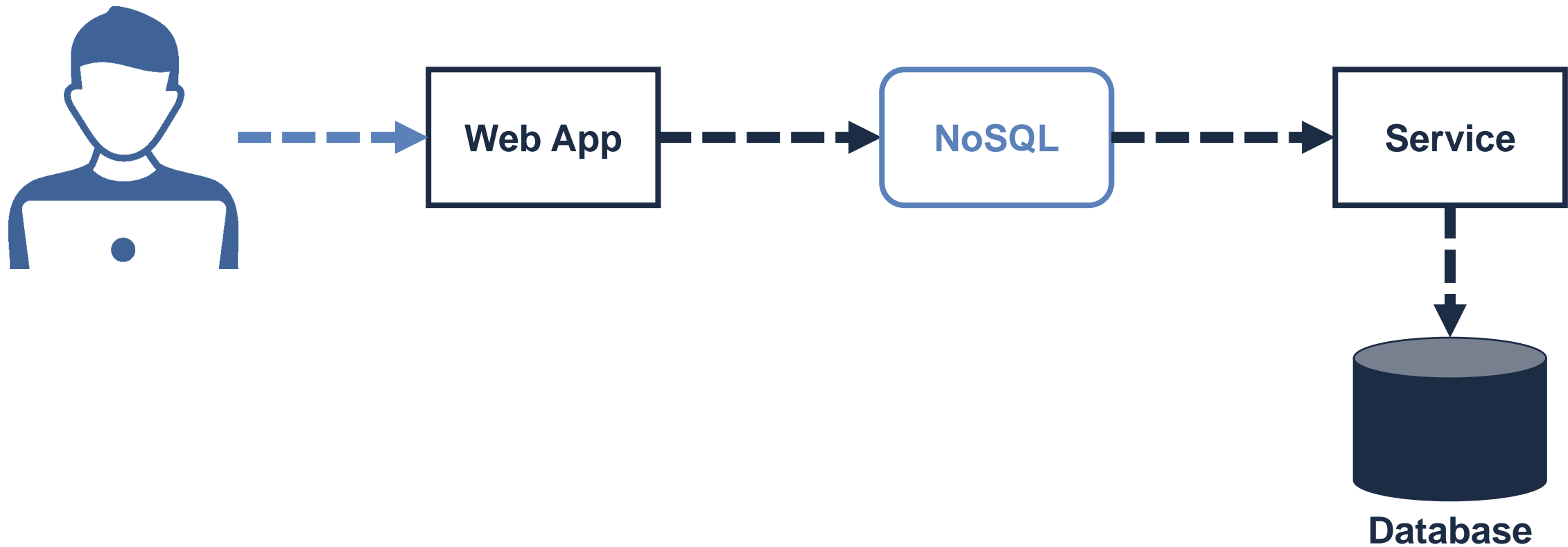
Basado en eventos

- **Transacción/acciones generadas como eventos**
- **Mensajes usando message brokers**

Consistencia Eventual



Consistencia Eventual



Introducción a eventos

DEMO



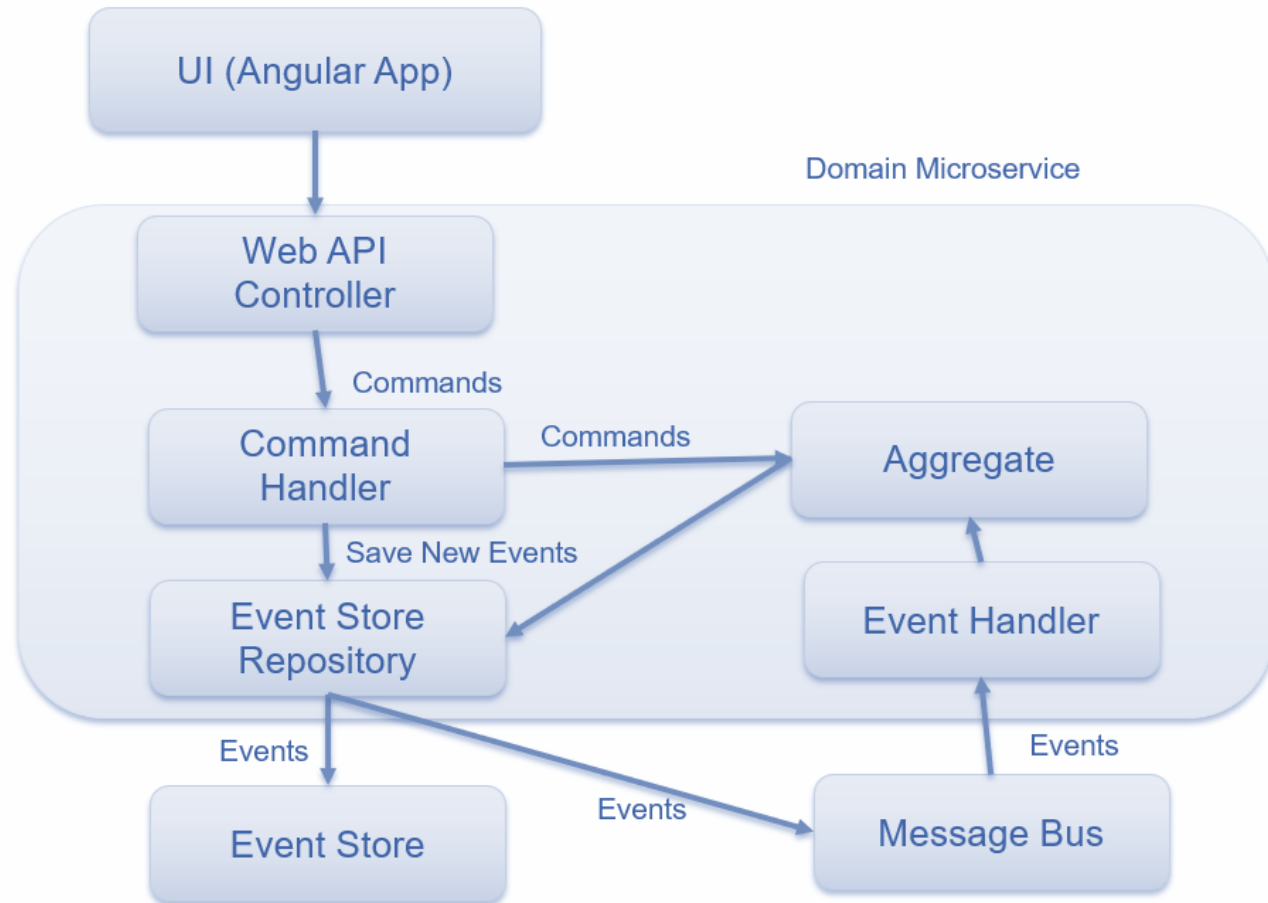
04



Implementando CQRS (SQL y NoSQL)

Implementando CQRS (SQL y NoSQL)

DEMO



05



Recomendaciones para su implementación

Conclusiones

Hay diferentes tipos de CQRS que puede aprovecharse en el diseño de software; No hay nada de malo en apegarse al Tipo **Regular** y no avanzar más allá de los Tipos **Premium** o **Deluxe**, siempre y cuando el Tipo **Regular** cumpla con los requisitos de su aplicación.

CQRS no es una elección binaria. Hay algunas variaciones diferentes entre no separar las lecturas y escrituras en absoluto (tipo regular) y separarlas completamente (tipo deluxe).

Debe haber un **equilibrio entre el grado de segregación y la sobrecarga de complejidad** que introduce. El equilibrio en sí debe encontrarse en cada aplicación de software concreta aparte, a menudo después de varias iteraciones. El CQRS en sí no debe implementarse "**solo porque podemos**"; Solo debe ponerse sobre la mesa para cumplir con requisitos concretos, a saber, **para escalar las operaciones de lectura de la aplicación.**



GRACIAS
POR SU PREFERENCIA

