Sesión 03

Persistencia y consistencia de datos – Parte I

Instructor:

ERICK ARÓSTEGUI

earostegui@galaxy.edu.pe

GALAXY
TRAINING

ÍNDICE

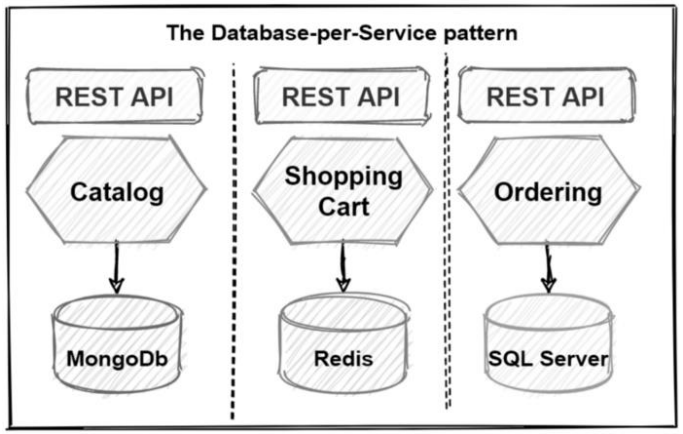
GALAXY
TRAINING

01

Patrones de gobierno de datos

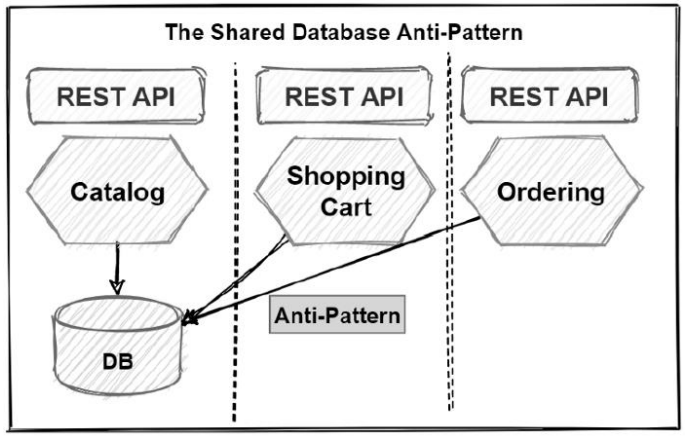
GALAXY
TRAINING

Patrones de gobierno de datos



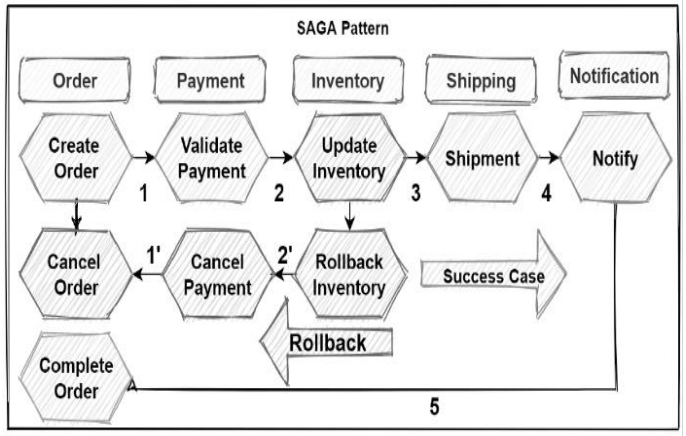
Base de datos por Servicio

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

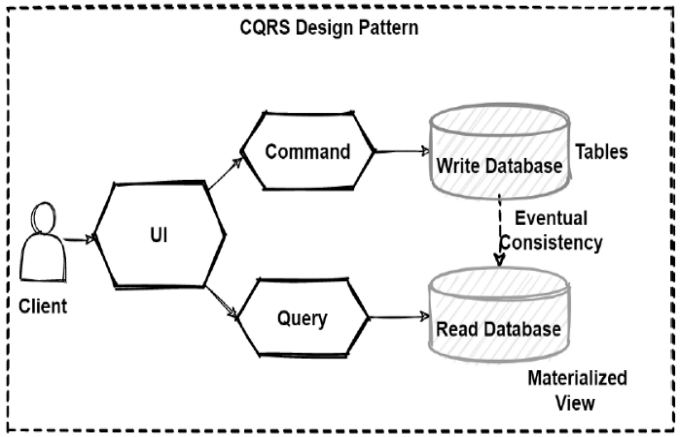
Permite que varios servicios accedan y almacenen datos en la misma base de datos. Si usa la base de datos compartida para varios microservicios, entonces caerá en un **antipatrón** y debe evitar estos enfoques.



Patrón Saga

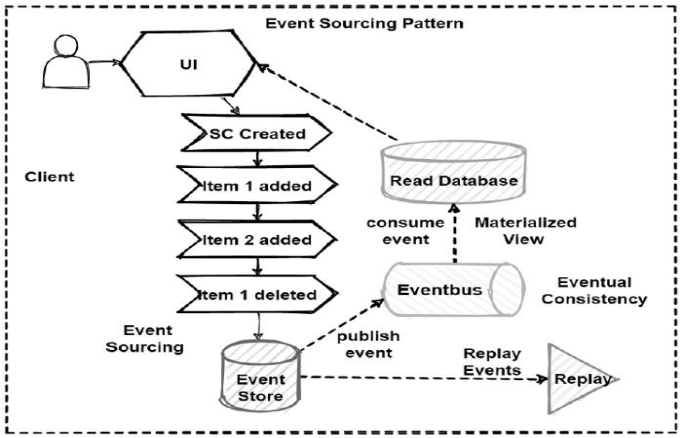
Saga es una secuencia de transacciones locales en la que el resultado de cada transacción depende de un evento anterior. Si alguna transacción falla, saga realiza una serie de transacciones de compensación.

Patrones de gobierno de datos



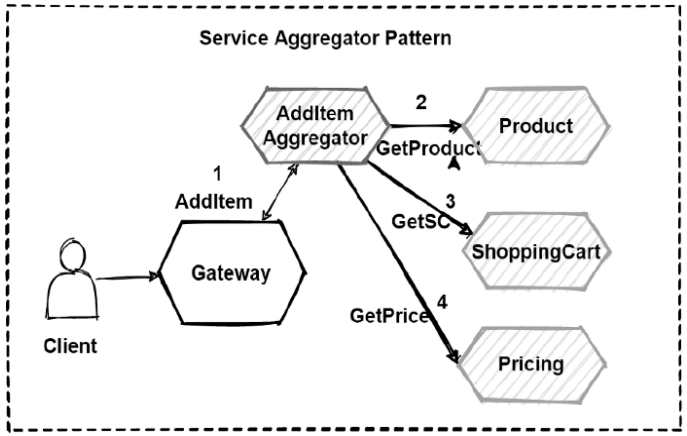
Patrón CQRS

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.**



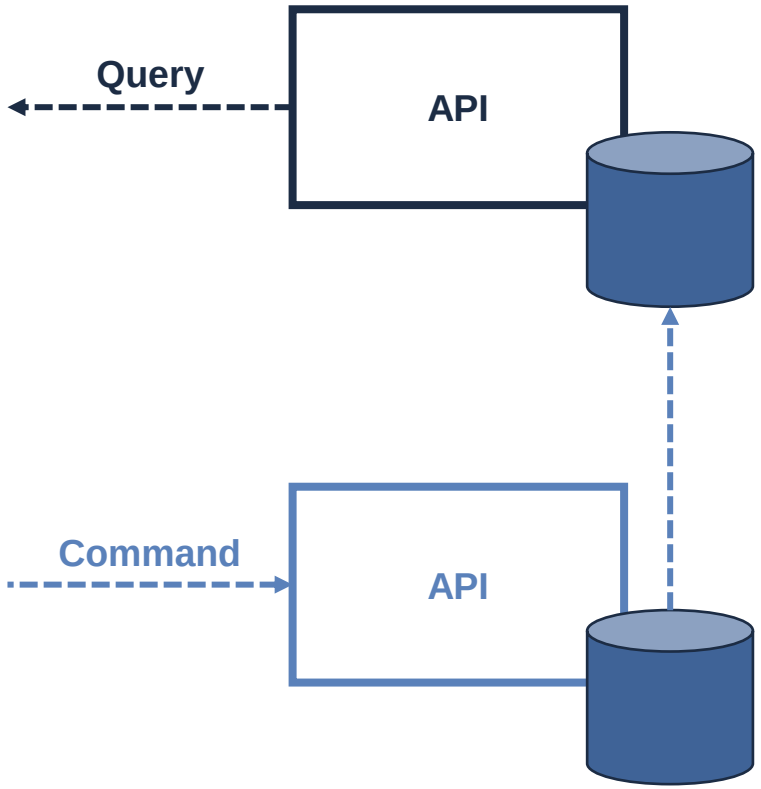
Composición de APIs

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

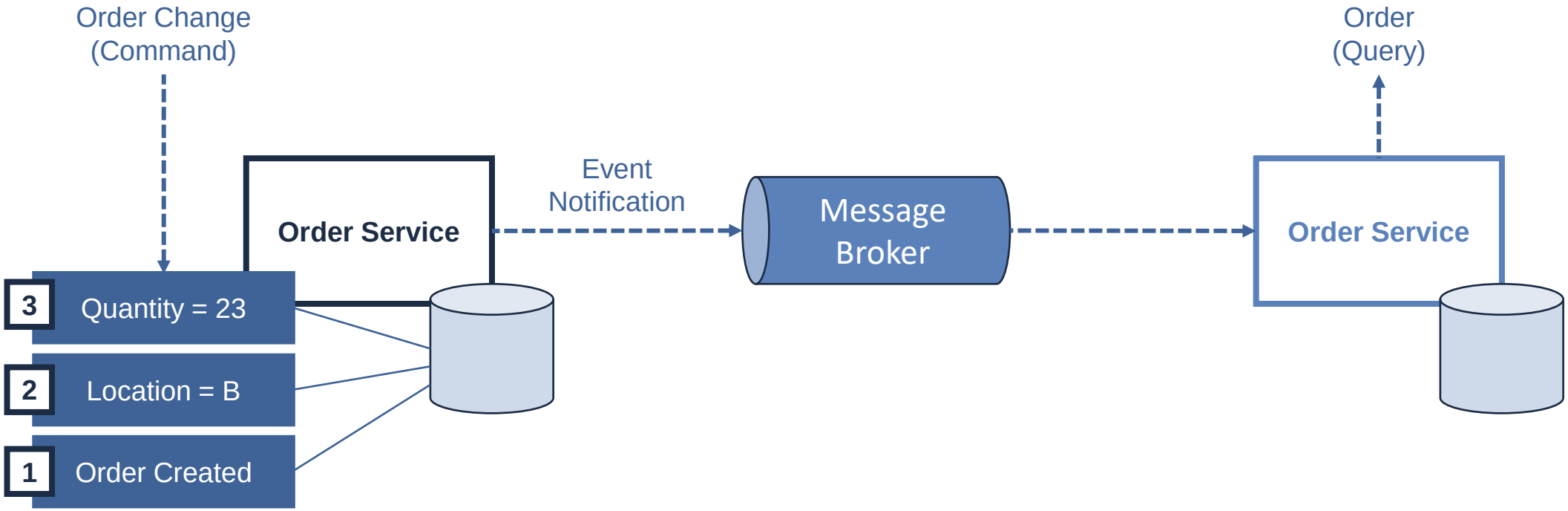
Patrón CQRS a un microservicio

Command Query Responsibility Segregation



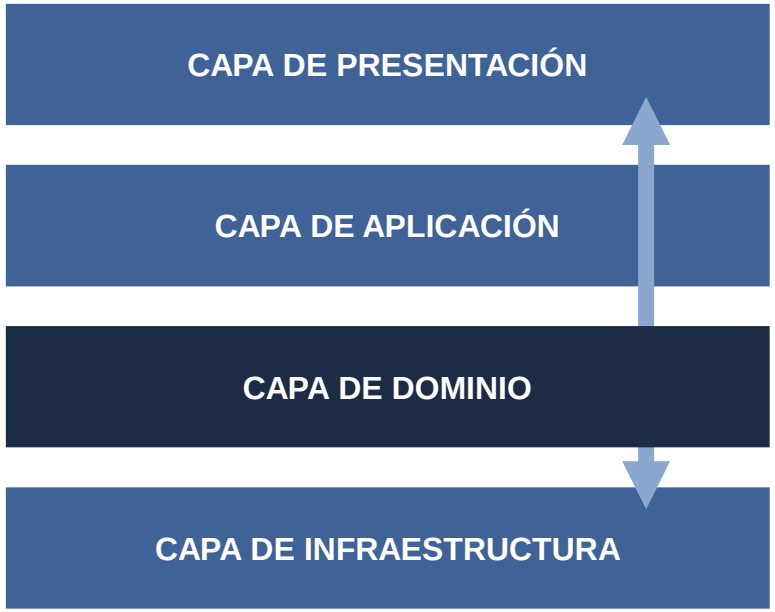
- CQRS
  - Modelos de comando (**Command**) y/o servicios
  - Modelos de consulta (**Query**) y/o servicios
- ¿Por qué?
  - Separación de responsabilidades.
    - Notificaciones de eventos manejadas por comando (**Escritura**)
    - Informes/funciones manejadas por consulta (**Lectura**)
  - Separación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

Command Query Responsibility Segregation

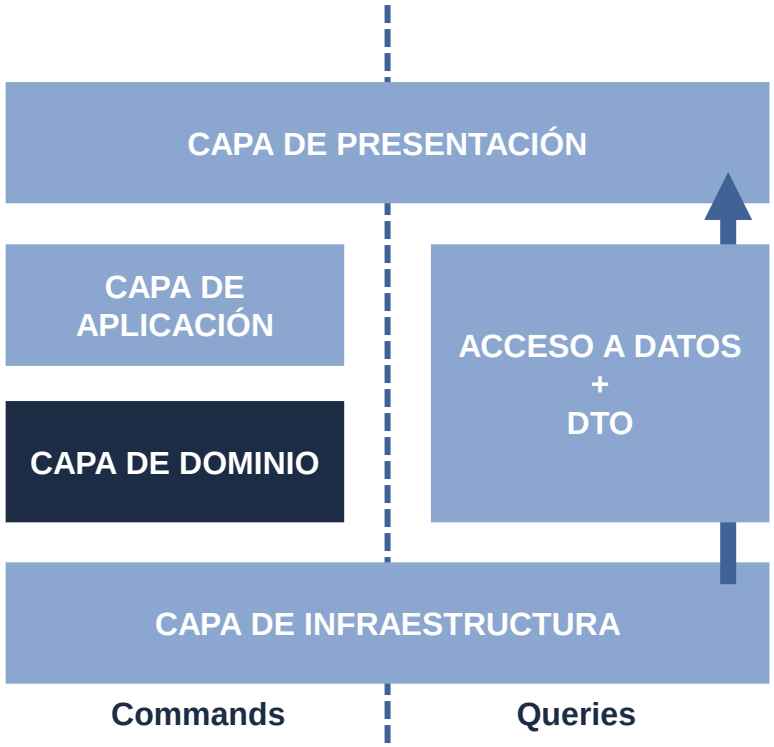


Command Query Responsibility Segregation

CANONICAL
Arquitectura en capas

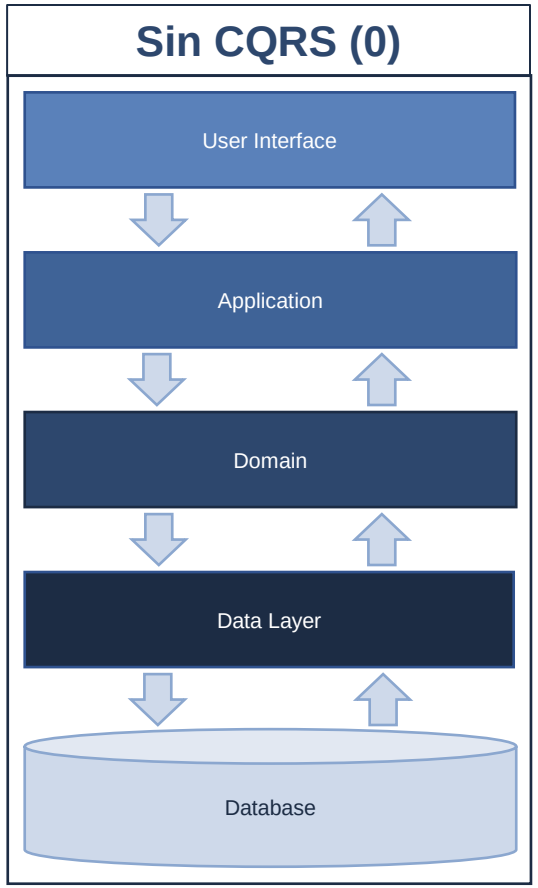


CQRS



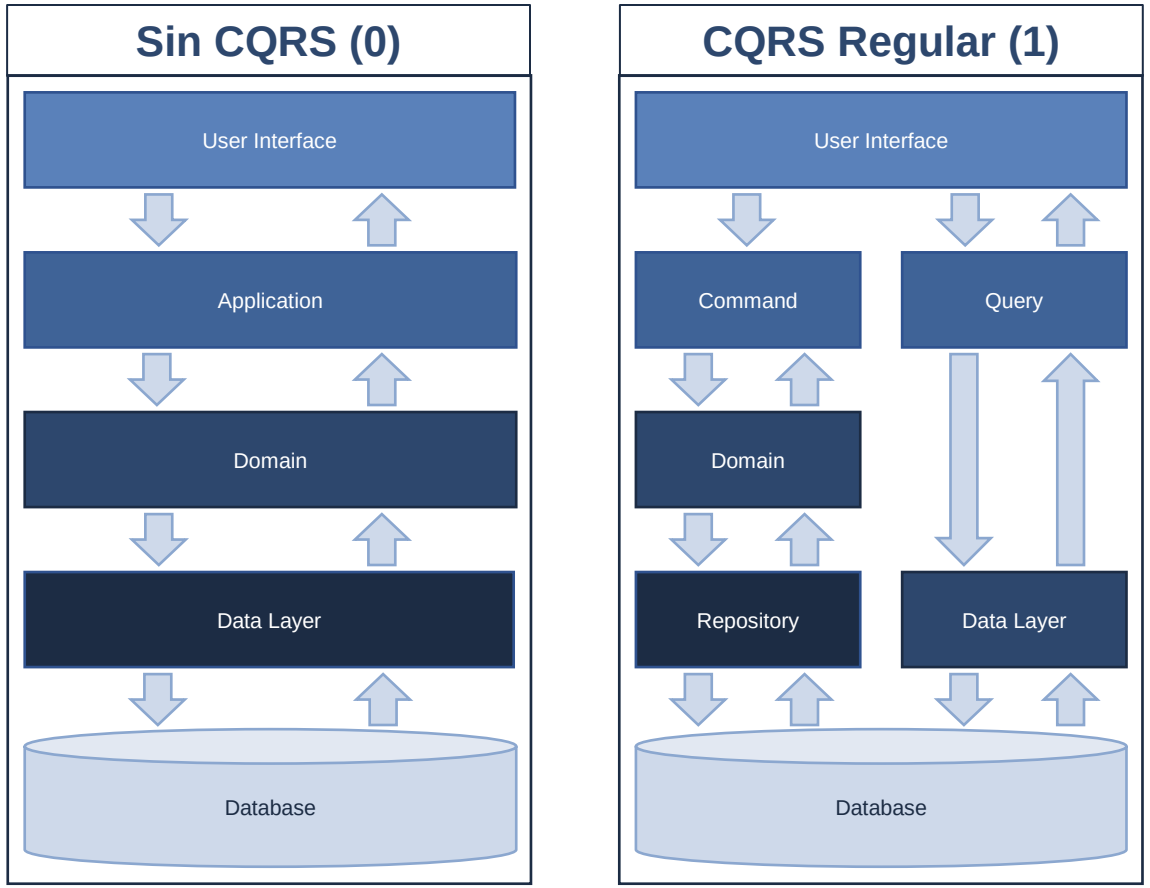
GALAXY TRAINING

Tipos de CQRS



GALAXY TRAINING

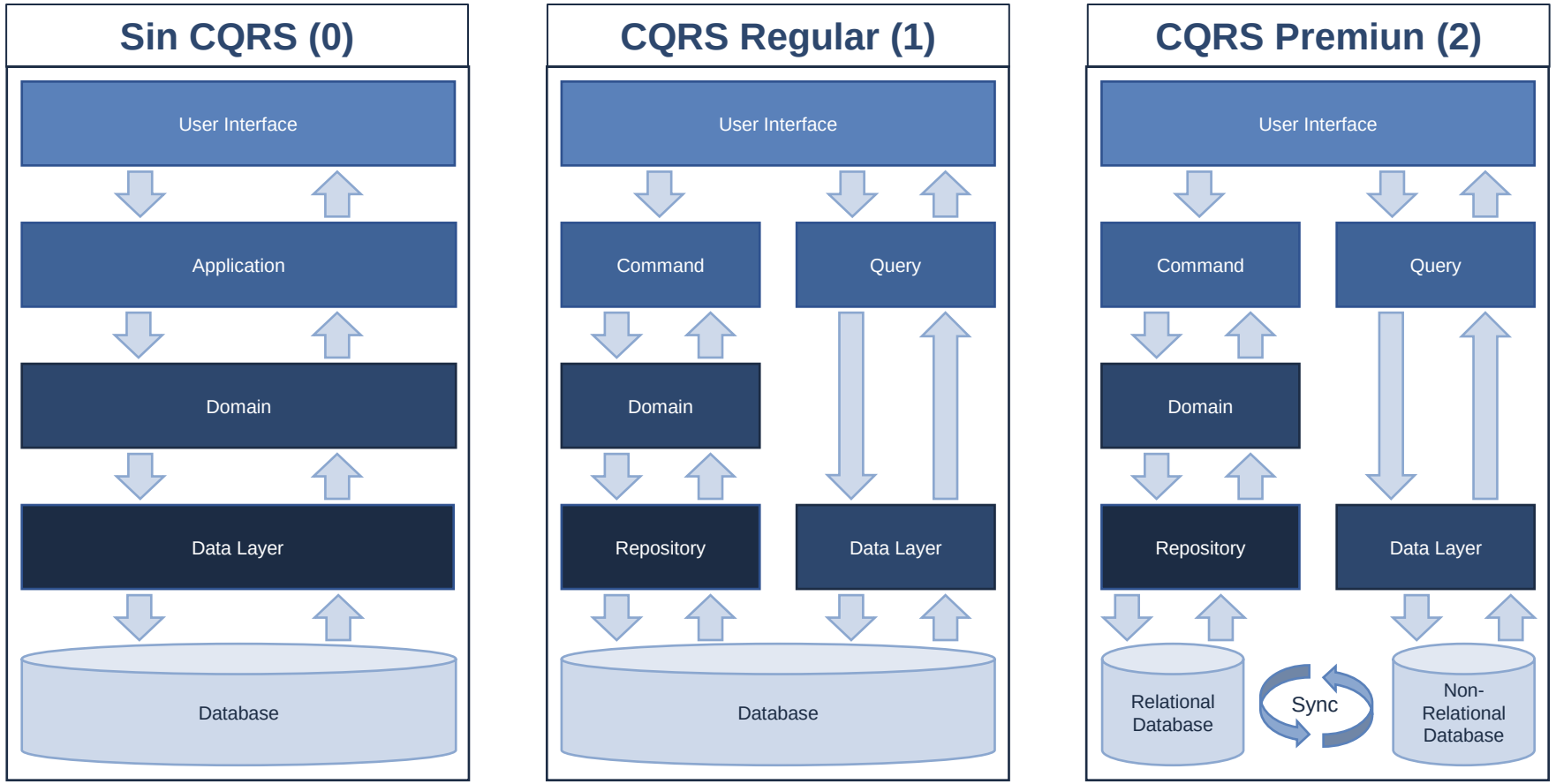
Tipos de CQRS



Patrón CQRS a un microservicio

GALAXY TRAINING

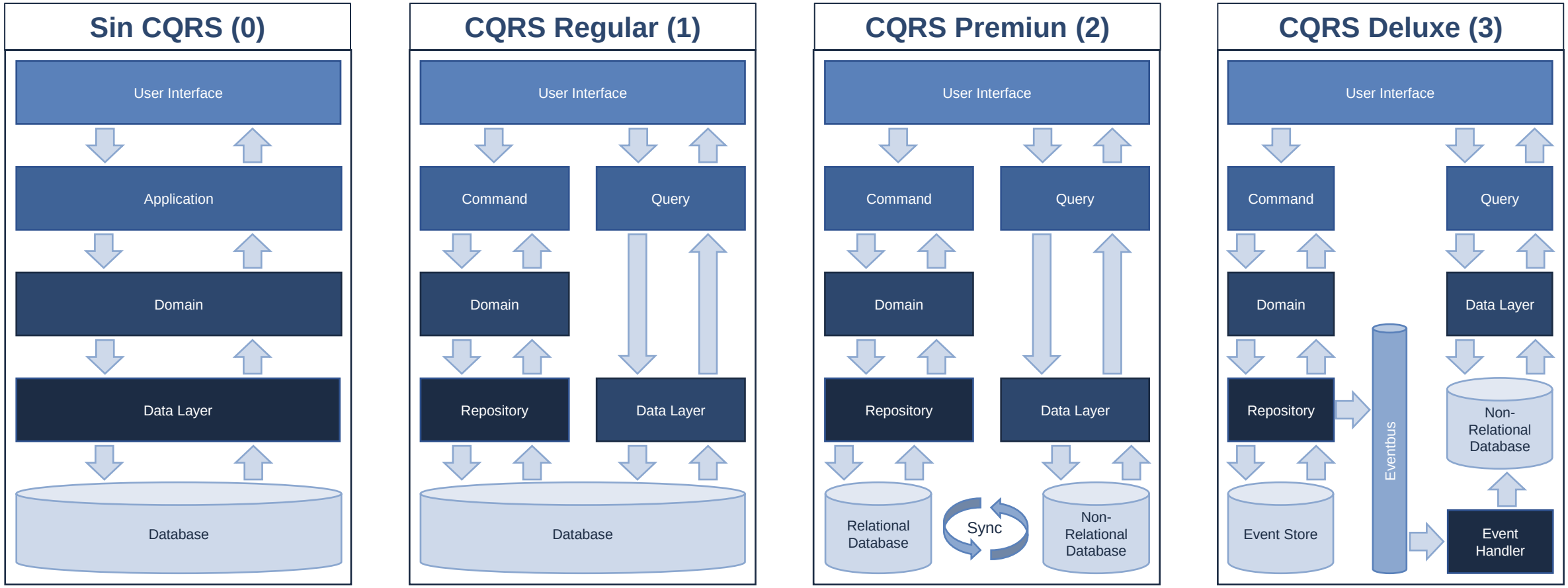
Tipos de CQRS



Patrón CQRS a un microservicio

GALAXY TRAINING

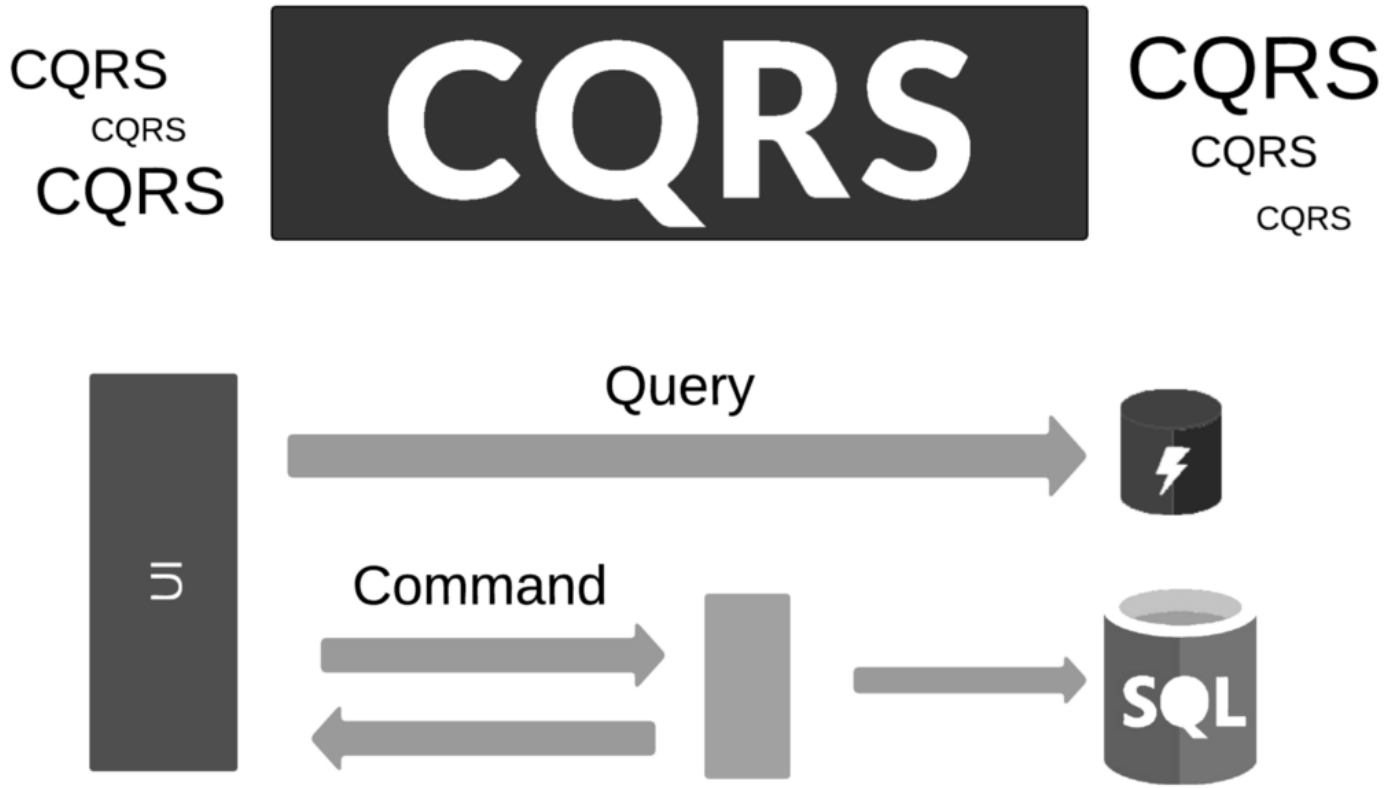
Tipos de CQRS



DEMO

GALAXY TRAINING

Patrón CQRS a un microservicio



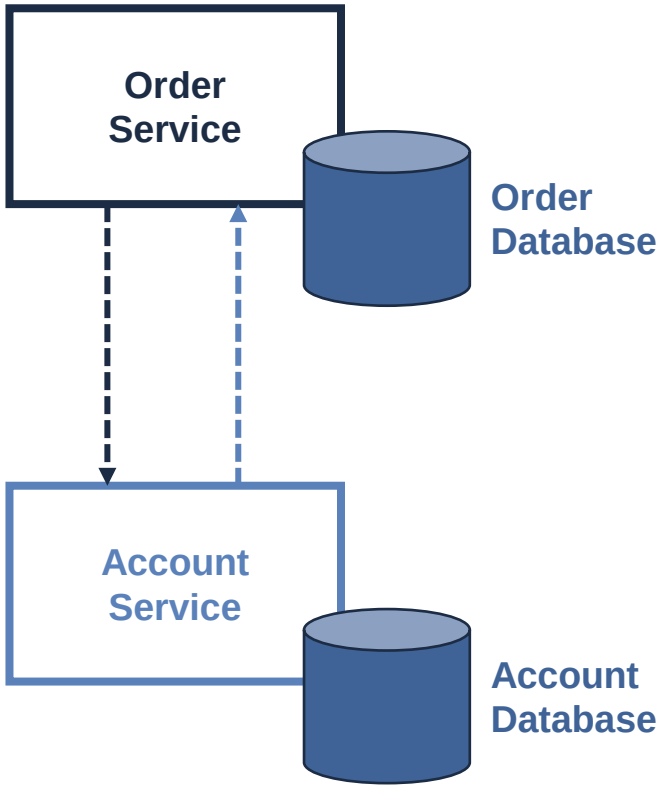
CQRS (Command Query Responsibility Segregation)

03

GALAXY
TRAINING

Consistencia eventual en microservicios

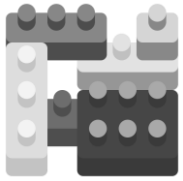
Cómo diseñar microservicios basados en API



- 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

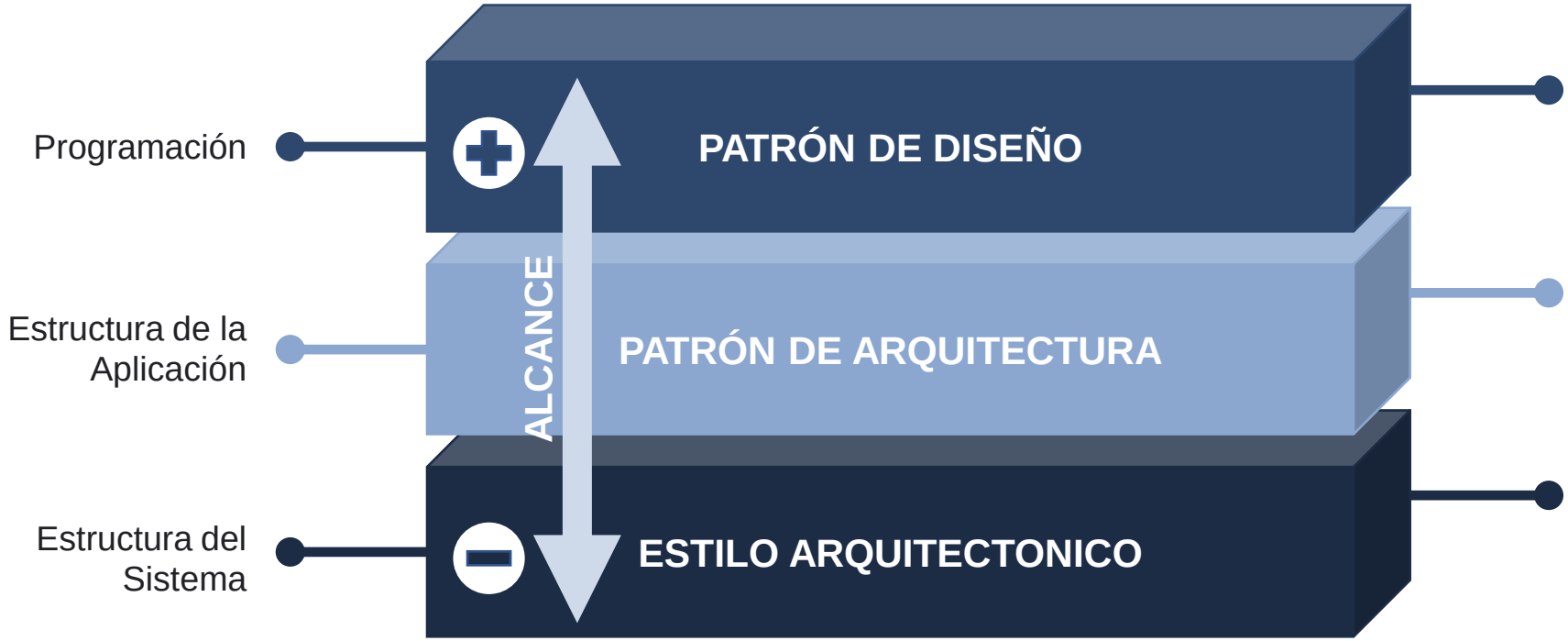
Desplegable independientemente

Contratos de apoyo y honor

API agnóstica a la tecnología

Diseño para fallos

Cómo diseñar microservicios basados en API

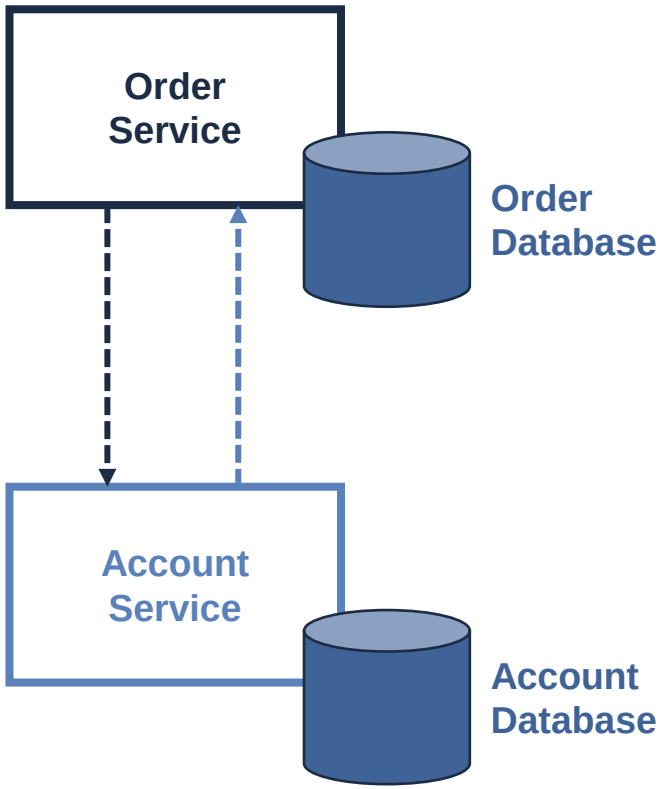


Un patrón de diseño es una forma de resolver un problema localizado (por ejemplo, patrón de observador)

Un patrón de arquitectura es una forma de implementar un estilo arquitectónico (por ejemplo, patrón de modelo-vista-controlador)

Un estilo arquitectónico es el diseño de la aplicación en el nivel más alto de abstracción (por ejemplo, arquitectura en capas)

Opciones de arquitectura



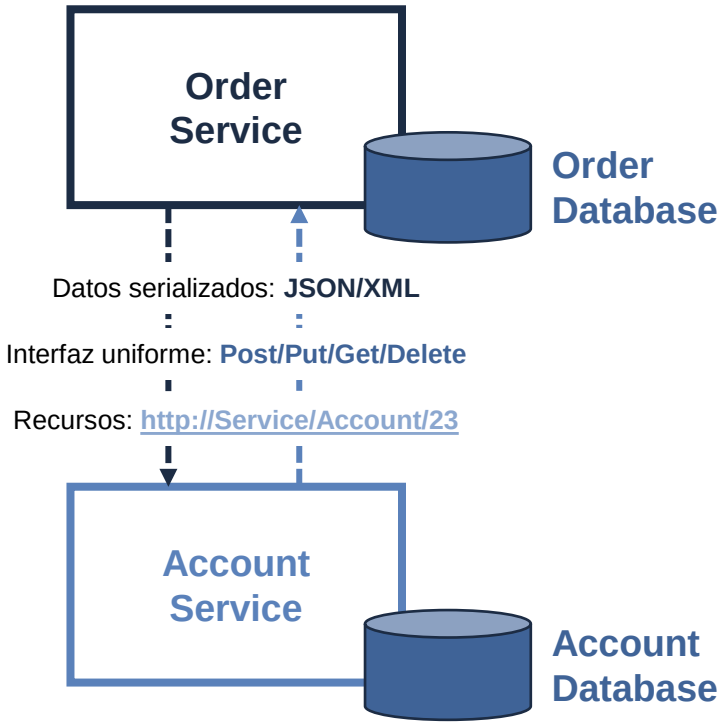
Patrones de arquitectura API

- Facade pattern
- Proxy pattern
- Stateless service pattern

Estilo de arquitectura API

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

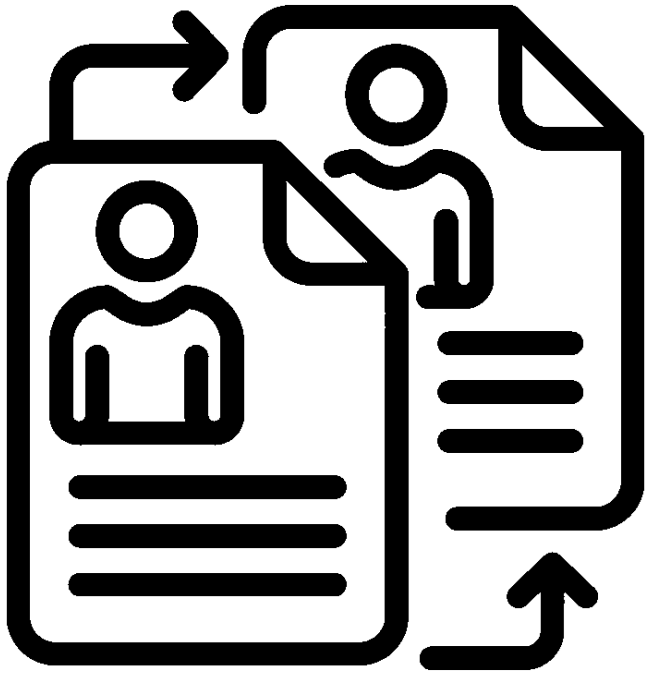
Estilo de arquitectura REST



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

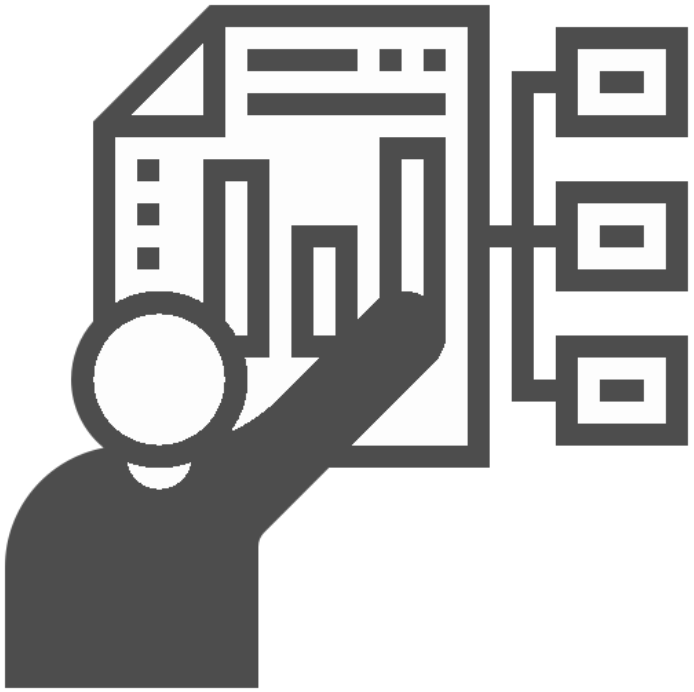
Consistencia Eventual



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.

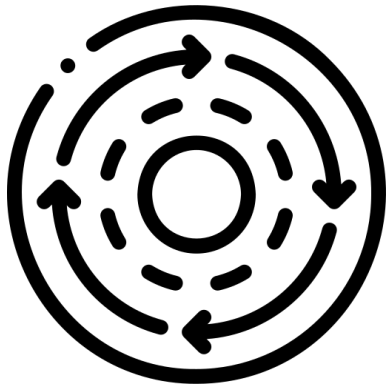
Consistencia Eventual

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.

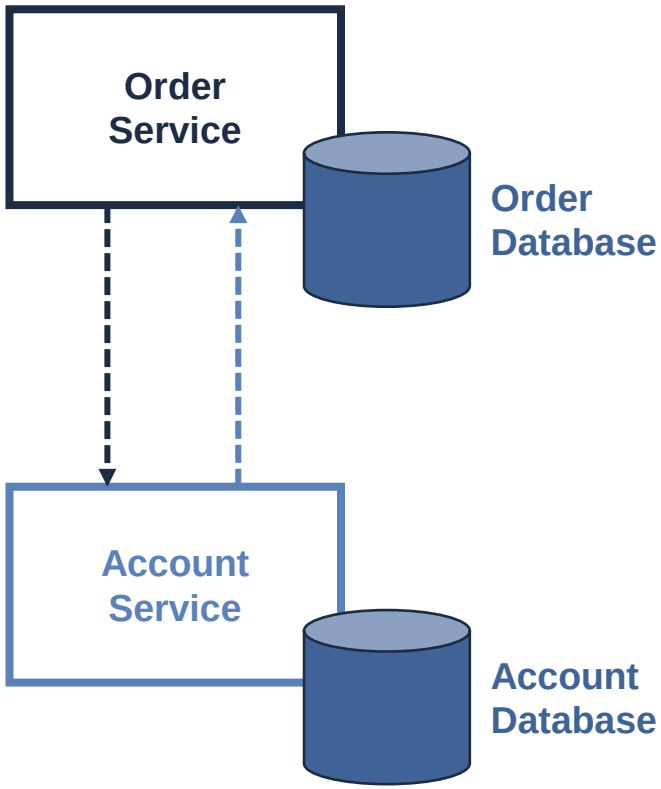
La disponibilidad (Availability)

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 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

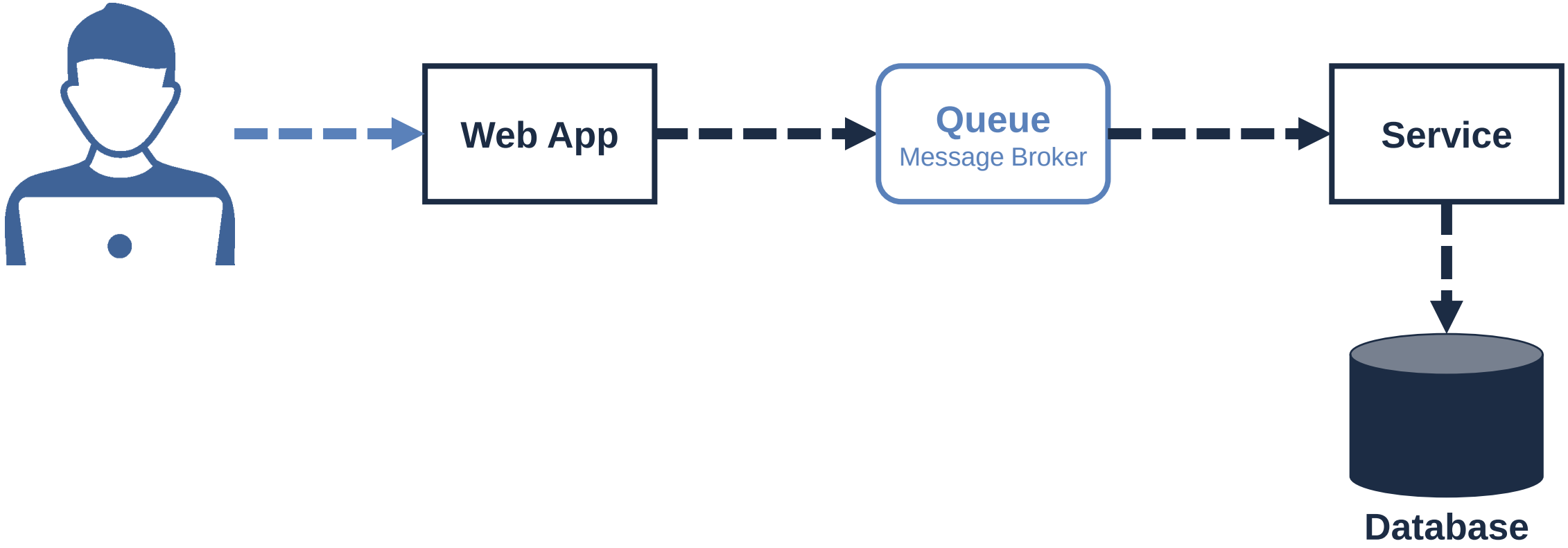
- **Evitar el bloqueo de recursos**
- Ideal para tareas de larga duración.
- **Preparado para inconsistencias**
- Condiciones de carrera

Replicación de datos

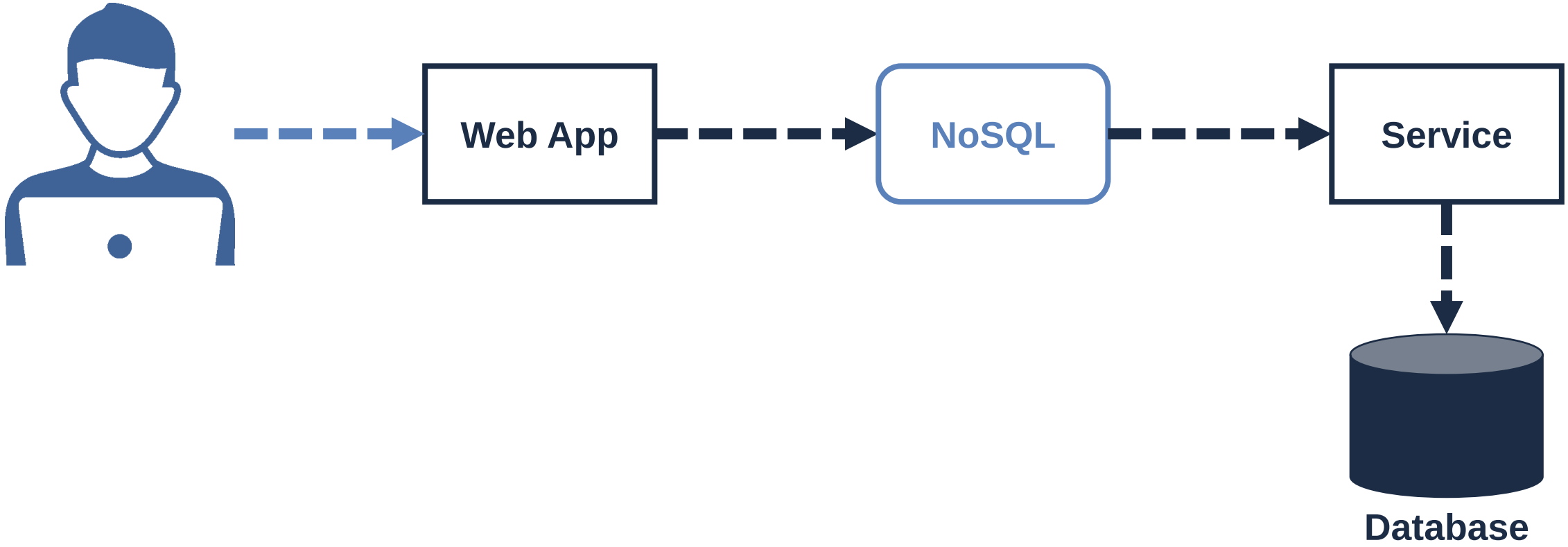
Basado en eventos

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

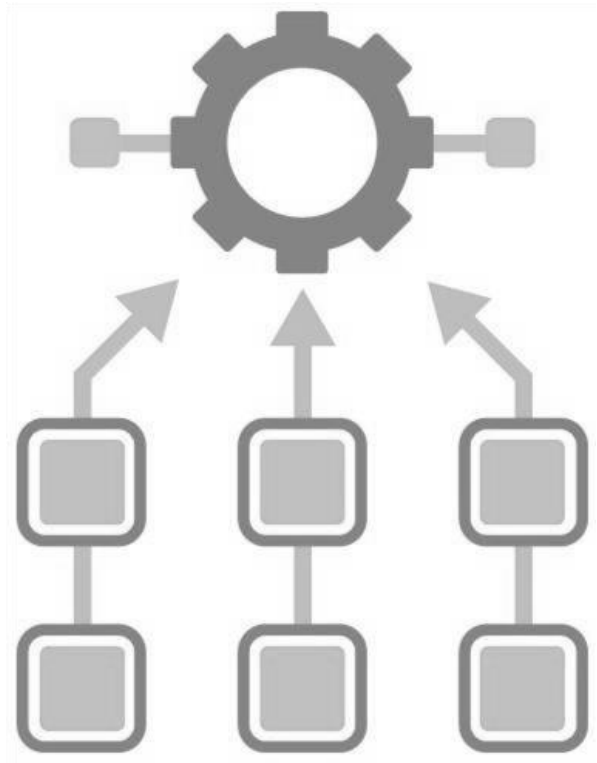
Consistencia Eventual



Consistencia Eventual



Arquitectura Orientada a Eventos (EDA)

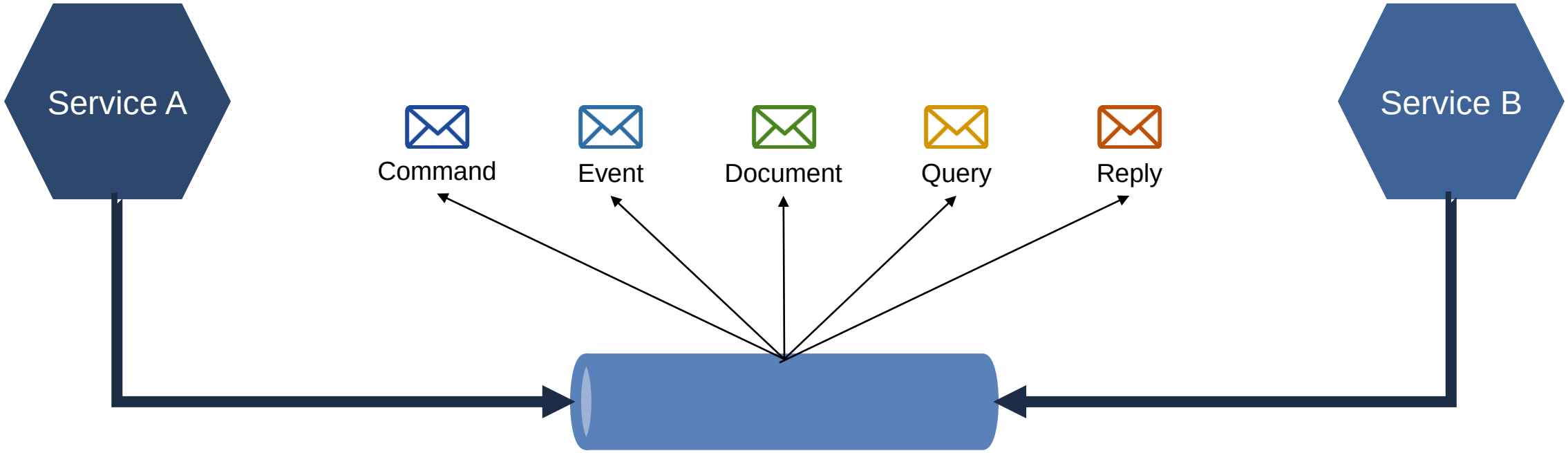


La arquitectura basada en eventos (EDA, por sus siglas en inglés**, Event-driven Architecture**) es un patrón en el que los microservicios se comunican principalmente a través de la **producción y el consumo de eventos**.

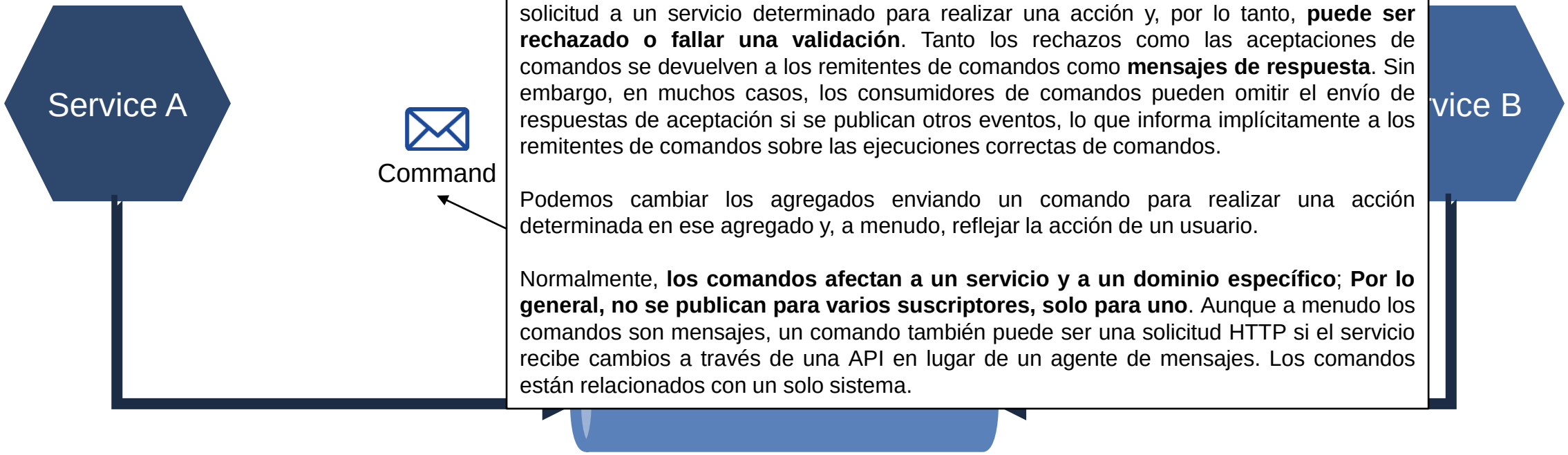
Un evento señala un cambio de estado o la finalización de una acción. **Los servicios pueden reaccionar a estos eventos** sin necesidad de llamar directamente a otros servicios.

Es un enfoque arquitectónico en el cual los **sistemas se diseñan para responder a eventos** o cambios de estado **de manera asíncrona**

Tipos de mensajes



Tipos de mensajes - Comandos

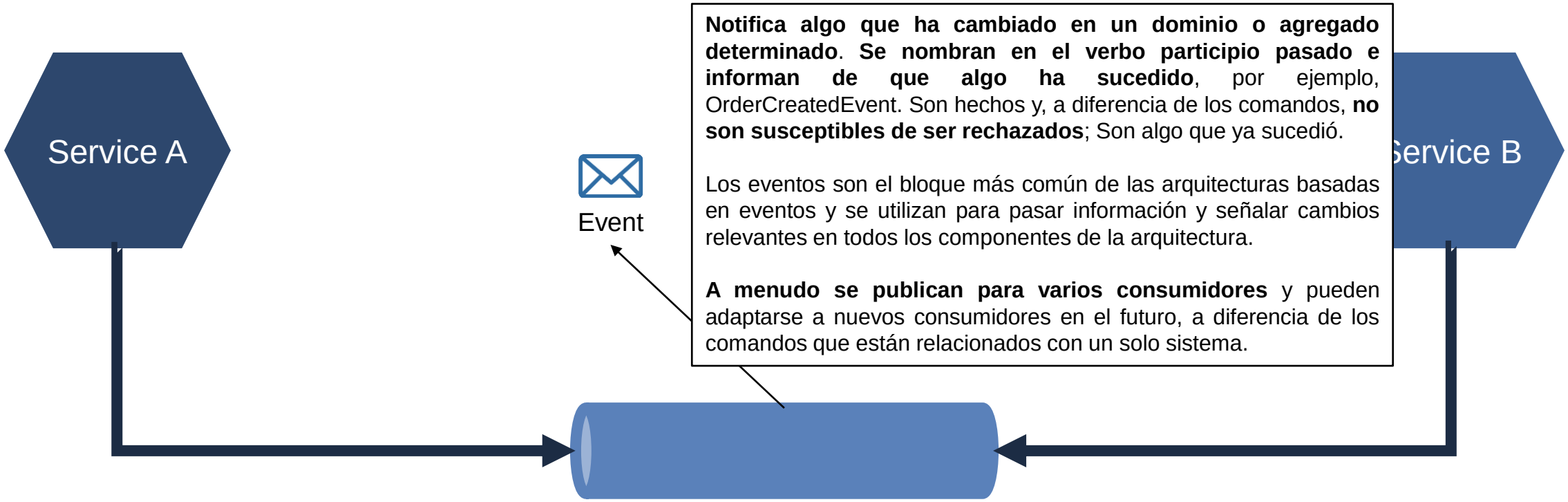


Son **órdenes para realizar una acción** determinada. Debemos **nombrarlos con un verbo** en forma imperativa, por ejemplo, CreateOrderCommand. Un comando es una solicitud a un servicio determinado para realizar una acción y, por lo tanto, **puede ser rechazado o fallar una validación**. Tanto los rechazos como las aceptaciones de comandos se devuelven a los remitentes de comandos como **mensajes de respuesta**. Sin embargo, en muchos casos, los consumidores de comandos pueden omitir el envío de respuestas de aceptación si se publican otros eventos, lo que informa implícitamente a los remitentes de comandos sobre las ejecuciones correctas de comandos.

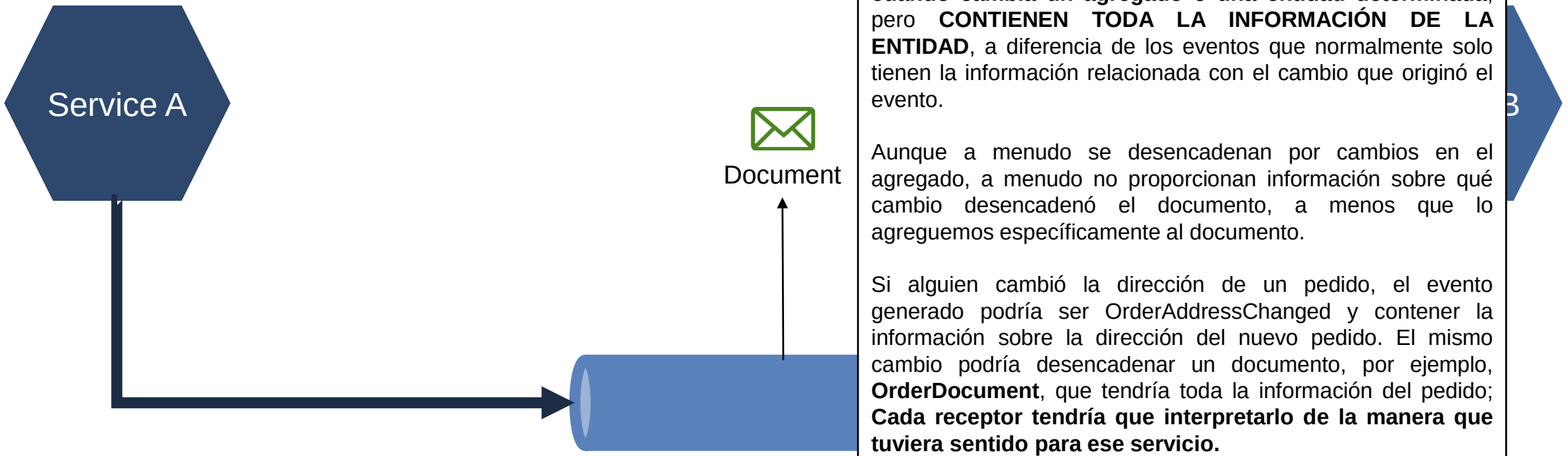
Podemos cambiar los agregados enviando un comando para realizar una acción determinada en ese agregado y, a menudo, reflejar la acción de un usuario.

Normalmente, **los comandos afectan a un servicio y a un dominio específico**; **Por lo general, no se publican para varios suscriptores, solo para uno**. Aunque a menudo los comandos son mensajes, un comando también puede ser una solicitud HTTP si el servicio recibe cambios a través de una API en lugar de un agente de mensajes. Los comandos están relacionados con un solo sistema.

Tipos de mensajes - Eventos



Tipos de mensajes - Documentos

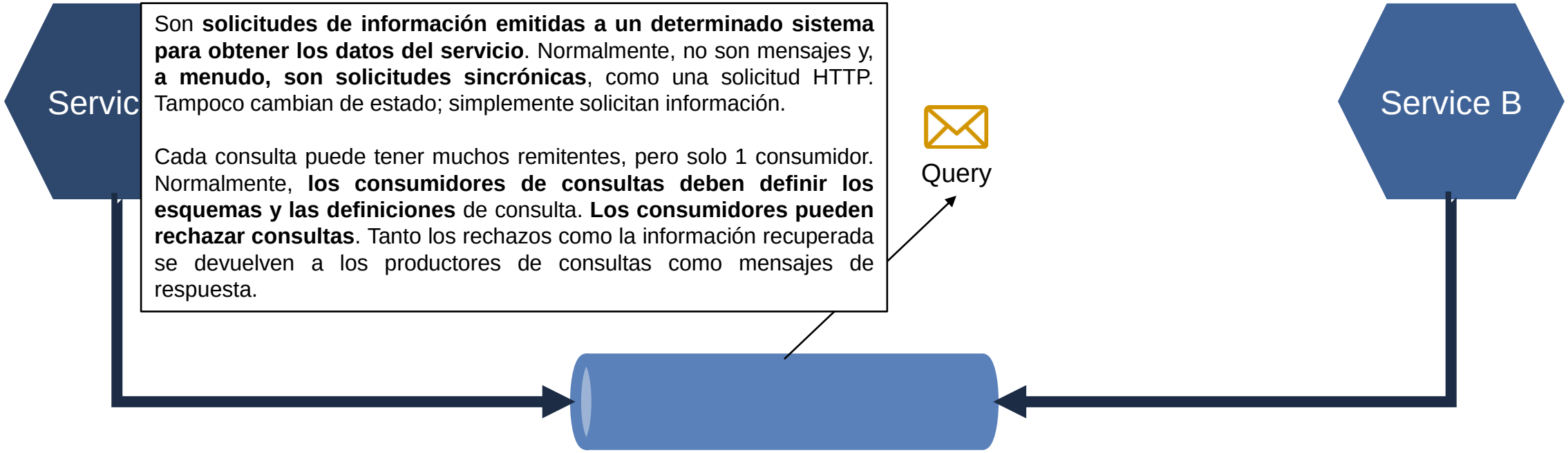


Son muy **parecidos a los eventos; El servicio los publica cuando cambia un agregado o una entidad determinada**, pero **CONTIENEN TODA LA INFORMACIÓN DE LA ENTIDAD**, a diferencia de los eventos que normalmente solo tienen la información relacionada con el cambio que originó el evento.

Aunque a menudo se desencadenan por cambios en el agregado, a menudo no proporcionan información sobre qué cambio desencadenó el documento, a menos que lo agreguemos específicamente al documento.

Si alguien cambió la dirección de un pedido, el evento generado podría ser OrderAddressChanged y contener la información sobre la dirección del nuevo pedido. El mismo cambio podría desencadenar un documento, por ejemplo, **OrderDocument**, que tendría toda la información del pedido; **Cada receptor tendría que interpretarlo de la manera que tuviera sentido para ese servicio.**

Tipos de mensajes - Consulta



Son **solicitudes de información emitidas a un determinado sistema para obtener los datos del servicio**. Normalmente, no son mensajes y, **a menudo, son solicitudes sincrónicas**, como una solicitud HTTP. Tampoco cambian de estado; simplemente solicitan información.

Cada consulta puede tener muchos remitentes, pero solo 1 consumidor. Normalmente, **los consumidores de consultas deben definir los esquemas y las definiciones** de consulta. **Los consumidores pueden rechazar consultas**. Tanto los rechazos como la información recuperada se devuelven a los productores de consultas como mensajes de respuesta.

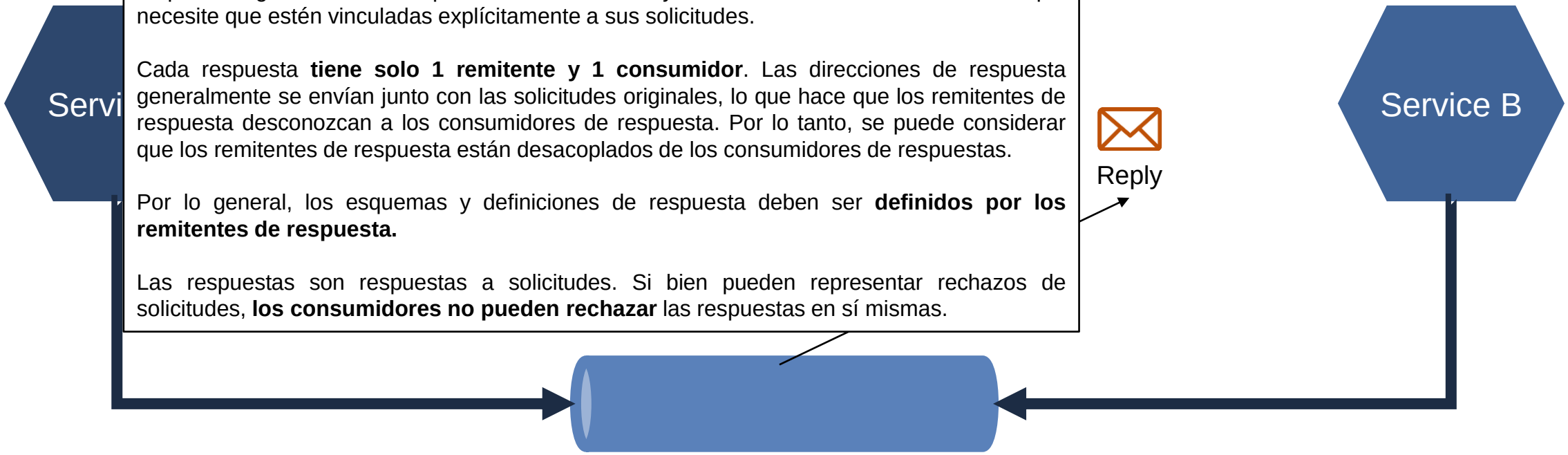
Tipos de mensajes - Respuesta

Las respuestas responden a consultas o comandos anteriores. Las respuestas correctas a las consultas **deben asignarse nombres de acuerdo con las consultas.** Está bien tener respuestas genéricas de aceptación de comandos y rechazo de solicitudes, a menos que necesite que estén vinculadas explícitamente a sus solicitudes.

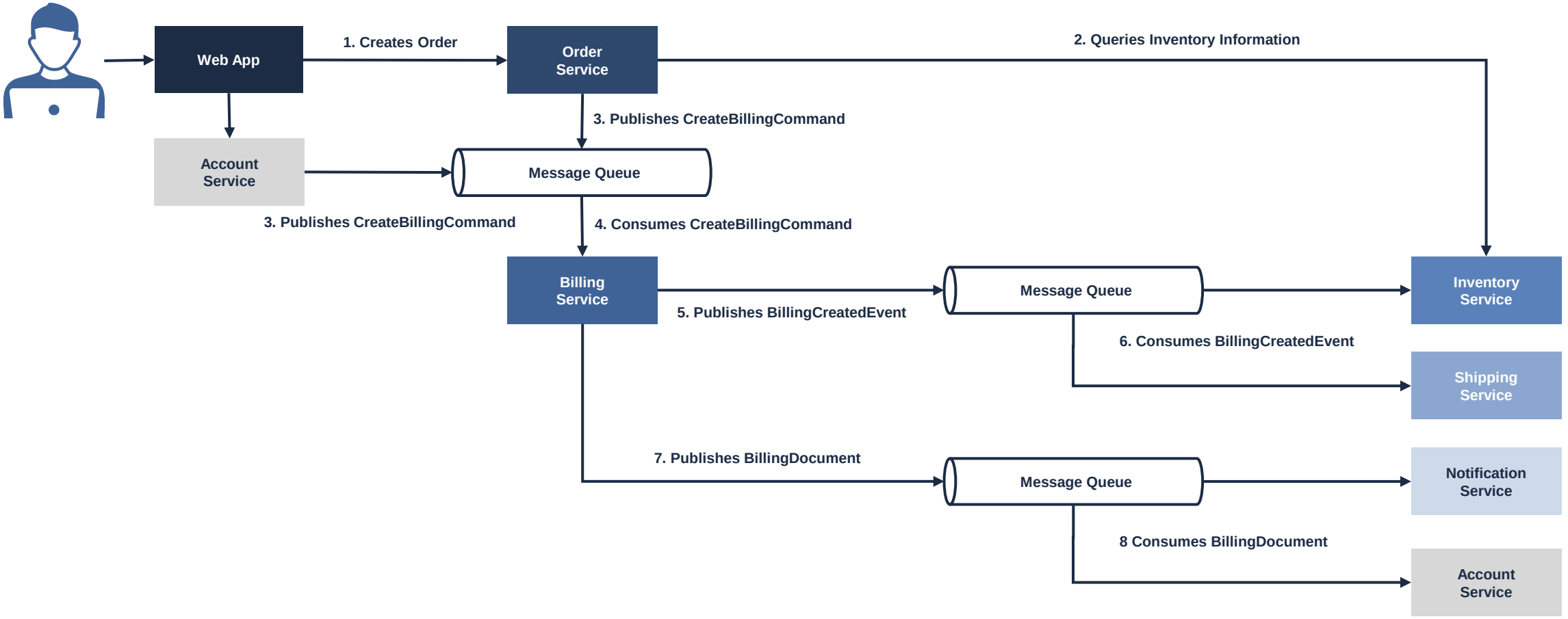
Cada respuesta **tiene solo 1 remitente y 1 consumidor**. Las direcciones de respuesta generalmente se envían junto con las solicitudes originales, lo que hace que los remitentes de respuesta desconozcan a los consumidores de respuesta. Por lo tanto, se puede considerar que los remitentes de respuesta están desacoplados de los consumidores de respuestas.

Por lo general, los esquemas y definiciones de respuesta deben ser **definidos por los remitentes de respuesta.**

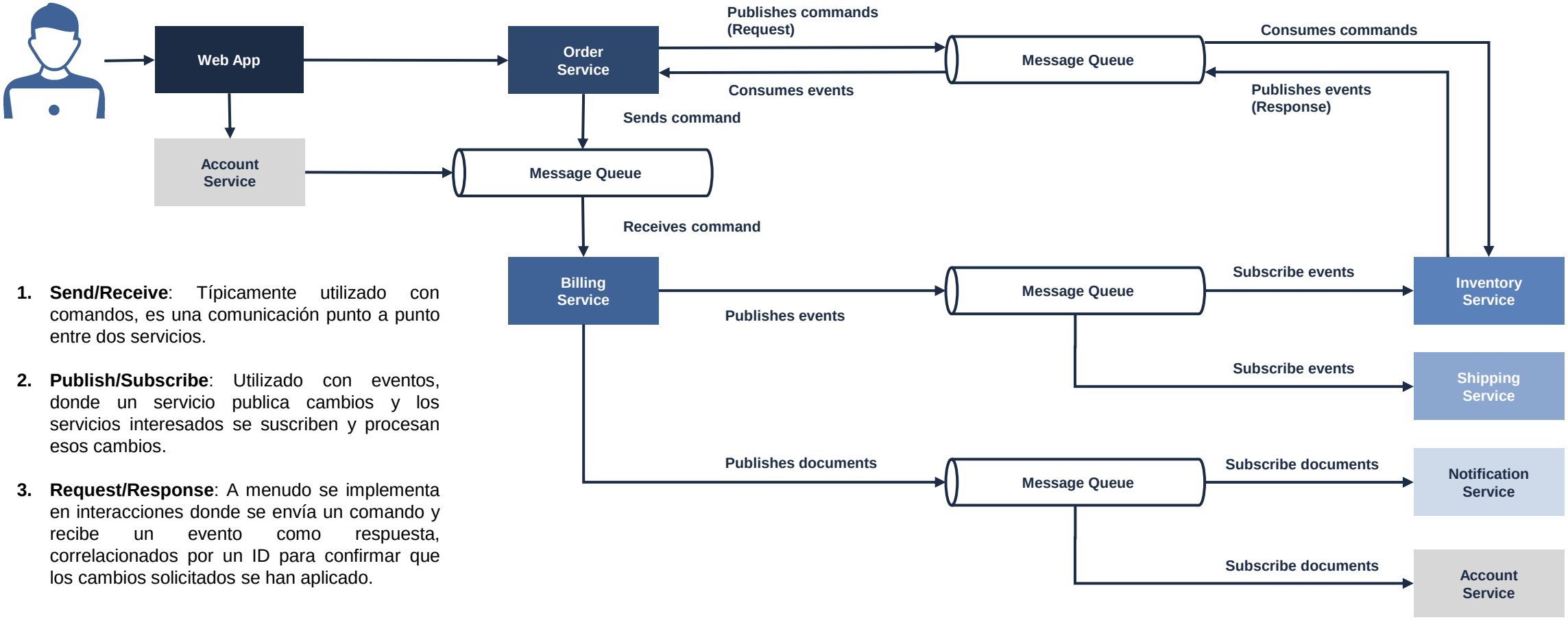
Las respuestas son respuestas a solicitudes. Si bien pueden representar rechazos de solicitudes, **los consumidores no pueden rechazar** las respuestas en sí mismas.



Tipos de mensajes



Patrones de mensajería



1. **Send/Receive**: Típicamente utilizado con comandos, es una comunicación punto a punto entre dos servicios.
2. **Publish/Subscribe**: Utilizado con eventos, donde un servicio publica cambios y los servicios interesados se suscriben y procesan esos cambios.
3. **Request/Response**: A menudo se implementa en interacciones donde se envía un comando y recibe un evento como respuesta, correlacionados por un ID para confirmar que los cambios solicitados se han aplicado.

DEMO

Introducción a eventos

GALAXY
TRAINING

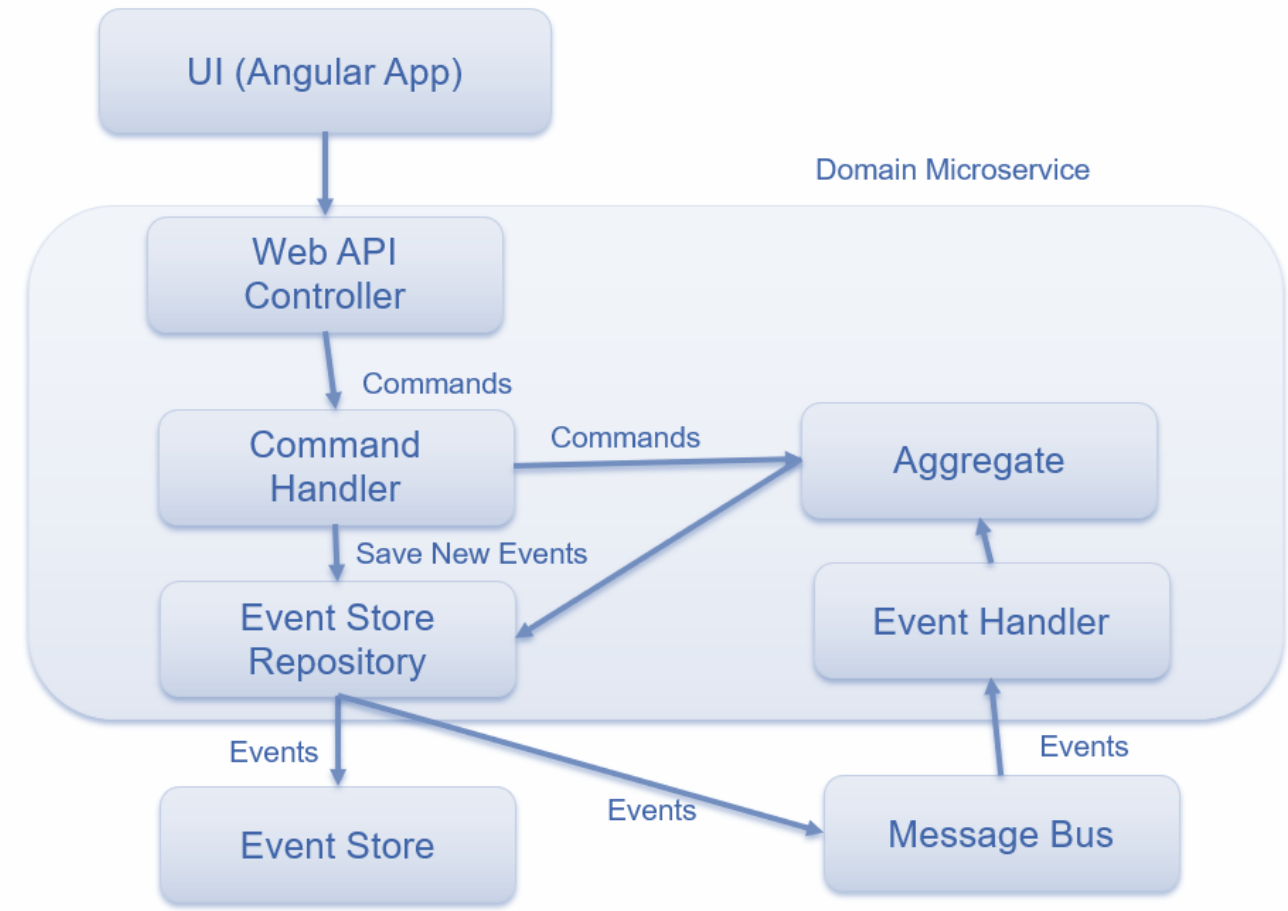
04

Implementando CQRS (SQL y NoSQL)

DEMO

GALAXY TRAINING

Implementando CQRS (SQL y NoSQL)



05

GALAXY
TRAINING

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

GALAXY
TRAINING