Università di Catania

Uni ct MATEMATICA E INFORMATICA

Design Patterns for Microservices

Alessandro Midolo, PhD

Dipartimento di Matematica e Informatica

Università di Catania

alessandro.midolo@unict.it

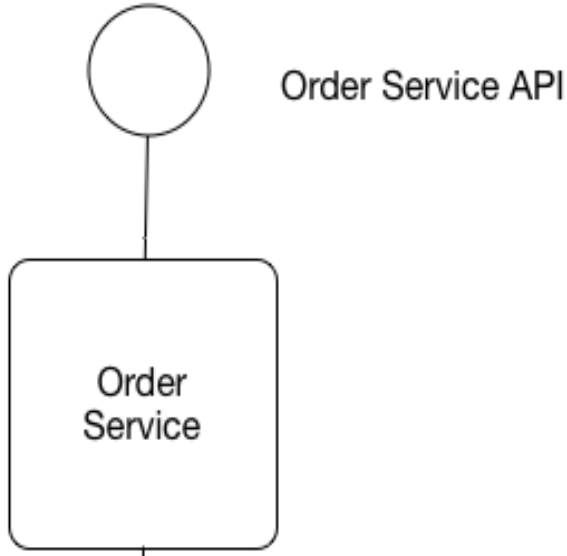
https://www.dmi.unict.it/amidolo/

A.A. 2024/2025

DATABASE PER SERVICE

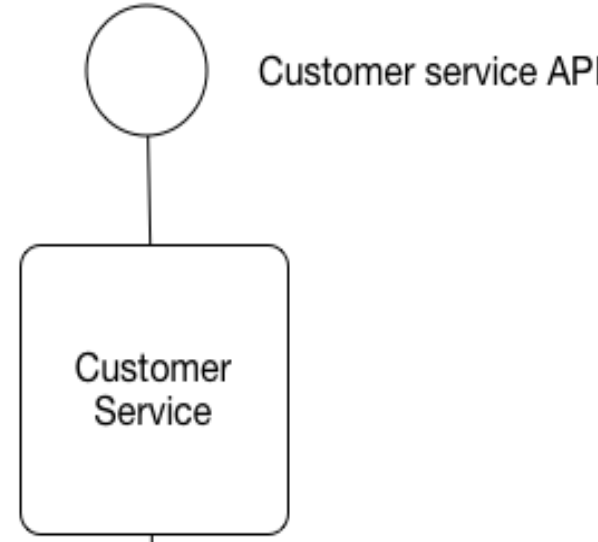
Database per Service

- Immaginiamo di star sviluppando un'applicazione per un negozio online utilizzando un'architettura a **microservizi**
- La maggior parte dei servizi ha bisogno di rendere persistenti i dati in un qualche tipo di database
- Ad esempio, **Order Service** memorizza informazioni sugli ordini e **Customer Service** memorizza informazioni sui clienti



ORDER table

ID	CUSTOMER_ID	STATUS	TOTAL	...
4567	234	ACCEPTED	84044.30	...



CUSTOMER table

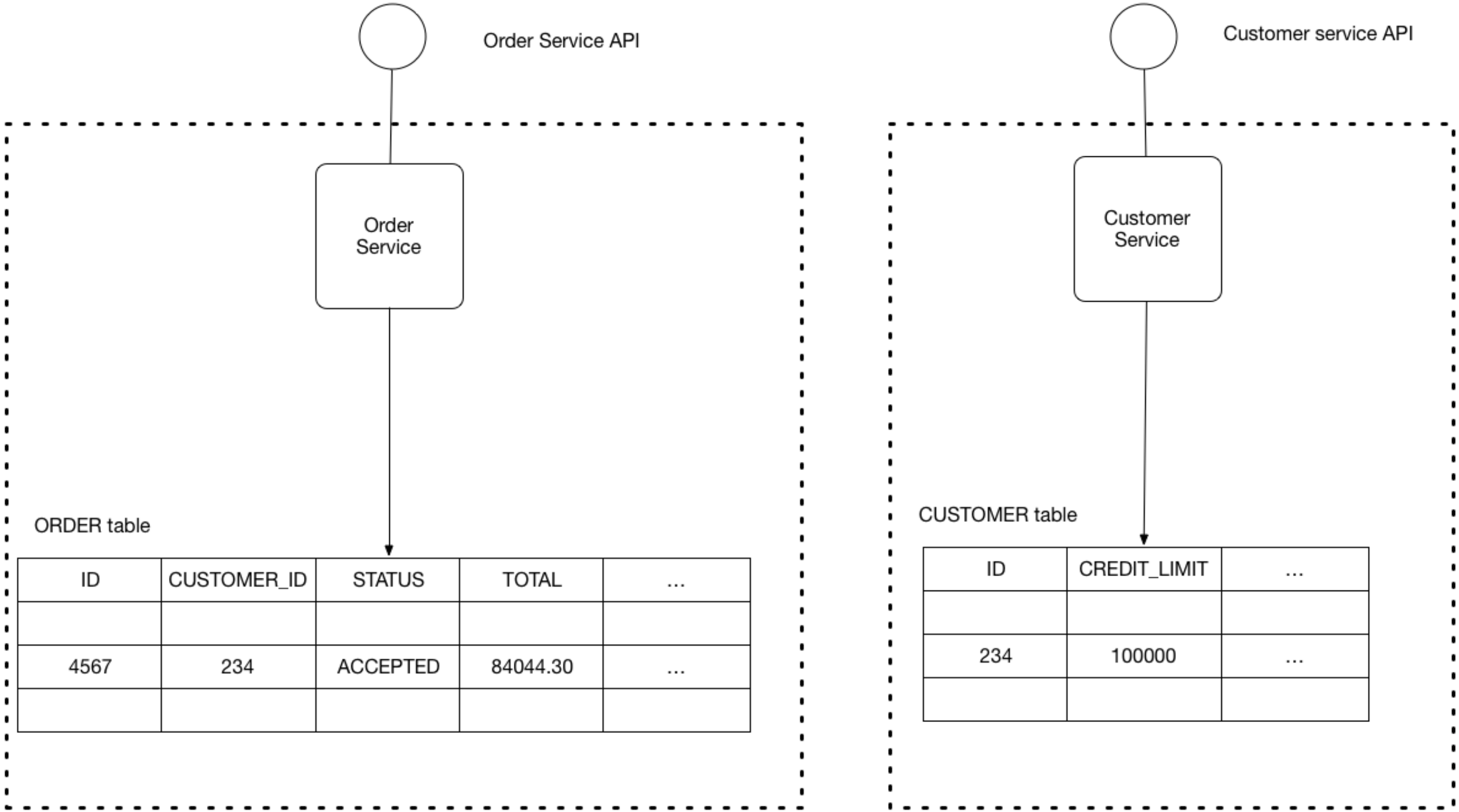
ID	CREDIT_LIMIT	...
234	100000	...

Problems

- I servizi devono essere **accoppiati** in modo **lasco** in modo che possano essere sviluppati, distribuiti e ridimensionati in modo indipendente
- Alcune transazioni aziendali devono imporre **invarianti** che si estendono su più servizi. Ad esempio, il caso d'uso Place Order deve verificare che un nuovo Order non superi il limite di credito del cliente. Altre transazioni aziendali devono aggiornare i **dati** di **proprietà** di più servizi
- Alcune transazioni aziendali devono **interrogare i dati** di proprietà di più servizi. Ad esempio, il metodo View Available Credit deve interrogare il Customer per trovare il creditLimit e gli Orders per calcolare l'importo totale degli ordini aperti
- Alcune query devono **unire i dati** di proprietà di più servizi. Ad esempio, trovare i clienti in una determinata regione e i loro ordini recenti richiede un'unione tra clienti e ordini
- A volte i database devono essere **replicati** e suddivisi in shard per essere ridimensionati
- Servizi diversi hanno requisiti di archiviazione dei dati diversi. Per alcuni servizi, un **database relazionale** è la scelta migliore. Altri servizi potrebbero aver bisogno di un database NoSQL come **MongoDB**, ottimo per archiviare dati complessi e non strutturati, o Neo4J, progettato per archiviare e interrogare in modo efficiente i dati dei grafici

Solution

- Mantieni i **dati persistenti** di ogni microservizio **privati** per quel servizio e accessibili solo tramite la sua API. Le transazioni di un servizio coinvolgono solo il suo database
- Il database del servizio è effettivamente **parte dell'implementazione** di quel servizio. Non è possibile accedervi direttamente da altri servizi
- Esistono diversi modi per mantenere privati i dati persistenti di un servizio. Non è necessario predisporre un database per ogni servizio. Ad esempio, se si utilizza un database relazionale, le opzioni sono:
  - **Private-tables-per-service**: ogni servizio possiede un set di tabelle a cui deve accedere solo quel servizio
  - **Schema-per-service**: ogni servizio ha uno schema di database privato per quel servizio
  - **Database-server-per-service**: ogni servizio ha il suo server di database.
- Private-tables-per-service e schema-per-service hanno il **sovraccarico** più basso. Utilizzare uno schema per servizio è interessante perché rende più chiara la proprietà. Alcuni servizi ad alta produttività potrebbero aver bisogno del proprio server di database
- È una buona idea creare barriere che impongano questa modularità. Ad esempio, è possibile assegnare un ID utente di database diverso a ogni servizio e utilizzare un meccanismo di **controllo dell'accesso** al database come le concessioni


Order Service API
Order
Service
ORDER table
ID
CUSTOMER_ID
STATUS
TOTAL
...
4567
234
ACCEPTED
84044.30
...
Customer service API
Customer
Service
CUSTOMER table
ID
CREDIT_LIMIT
...
234
100000
...

Pros

- Aiuta a garantire che i servizi siano debolmente accoppiati. Le **modifiche** al **database** di un servizio non hanno impatto sugli altri servizi
- Ogni servizio può utilizzare il tipo di database più adatto alle **proprie esigenze**. Ad esempio, un servizio che esegue ricerche di testo potrebbe utilizzare ElasticSearch. Un servizio che manipola un grafico sociale potrebbe utilizzare Neo4j

Cons

- L'implementazione di **transazioni aziendali** che si estendono su più servizi non è semplice. È meglio evitare le transazioni distribuite. Inoltre, molti database moderni (NoSQL) non le supportano
- L'implementazione di query che **uniscono dati** che ora si trovano in più database è impegnativa
- **Complessità** della gestione di più database SQL e NoSQL

EVENT SOURCING

Event Sourcing

- Nelle architetture a microservizi, il coordinamento tra diversi componenti indipendenti è essenziale per mantenere la coerenza dello stato. Ad esempio, in un sistema di e-commerce, i microservizi responsabili di gestire ordini, pagamenti e inventario devono collaborare per garantire che gli ordini siano processati correttamente.
- Quando un cliente effettua un ordine, devono succedere diverse cose:
  - Il microservizio "Ordini" crea un nuovo ordine
  - Il microservizio "Pagamenti" verifica ed elabora il pagamento
  - Il microservizio "Inventario" riserva i prodotti acquistati.
- Se questi passaggi vengono eseguiti usando un approccio transazionale centralizzato, il sistema diventa fragile e difficilmente scalabile
- Inoltre, una soluzione tradizionale potrebbe salvare solo lo stato corrente (ad esempio, "Ordine completato") senza conservare il dettaglio delle azioni che hanno portato a quello stato.

Problems

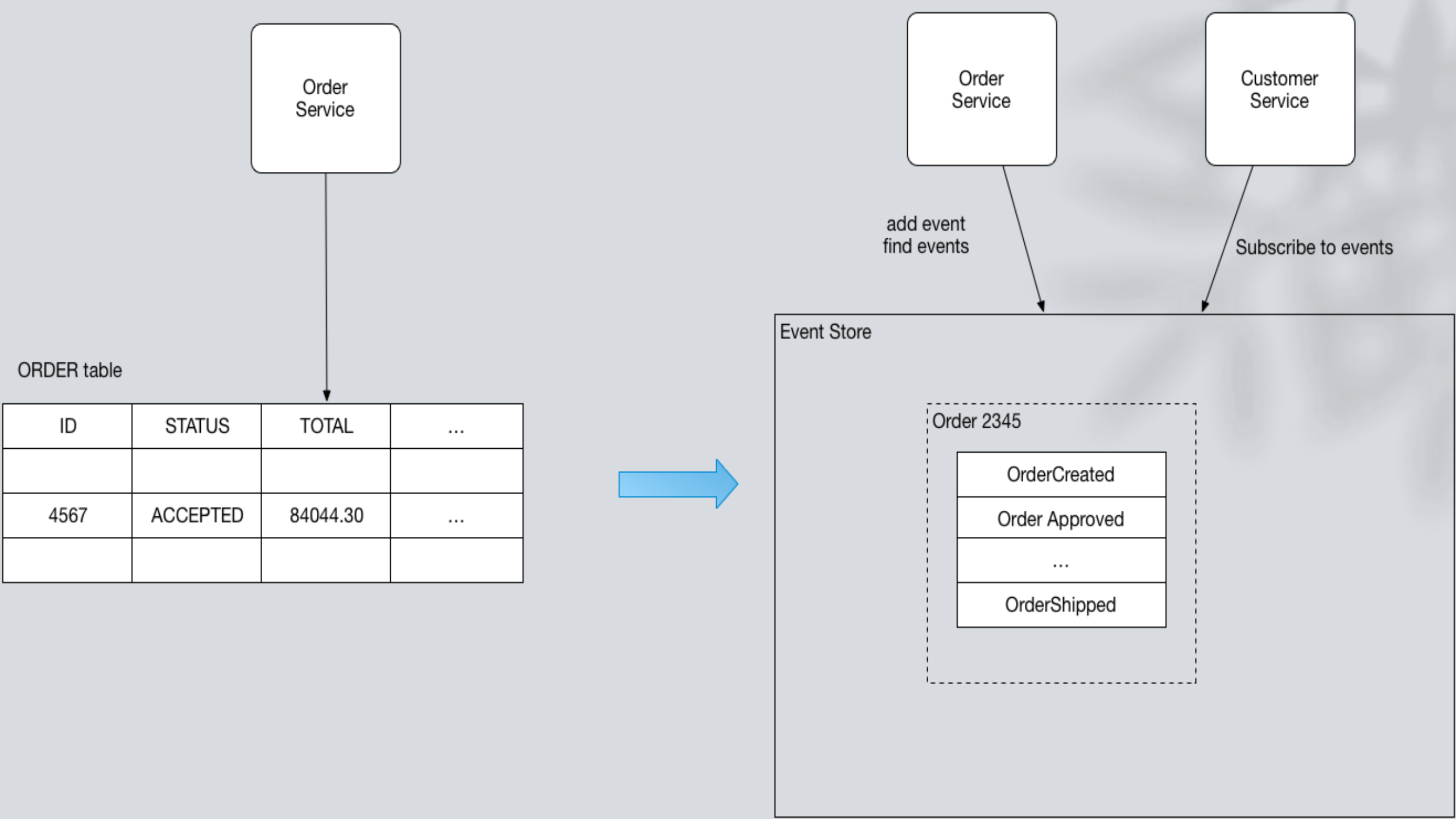
- Come mantenere la **coerenza** tra i microservizi quando ciascuno è indipendente e distribuito?
- Come evitare di **perdere il contesto** delle azioni passate che hanno portato allo stato attuale (audit e tracciabilità)?
- Come garantire che i microservizi reagiscano in modo **asincrono** a eventi rilevanti senza creare **dipendenze rigide** o **blocchi**?
- Come supportare il **debugging** e l'analisi retrospettiva in un sistema che può evolvere nel tempo?

Forces

- **Auditabilità**: In settori come il bancario o l'e-commerce, è essenziale poter dimostrare cosa è successo durante l'elaborazione di un ordine o di una transazione
- **Asincronia**: I microservizi devono comunicare senza dipendenze forti, per garantire resilienza e disponibilità
- **Evoluzione dei requisiti**: Il sistema deve poter cambiare senza compromettere i dati esistenti. Ad esempio, aggiungere nuovi campi o logiche agli eventi non dovrebbe invalidare quelli precedenti
- **Performance e scalabilità**: I sistemi moderni devono gestire un elevato numero di richieste, il che richiede una soluzione performante per il logging e la persistenza.

Solution

- Ogni azione significativa (es. "Ordine creato", "Pagamento effettuato", "Prodotti riservati") viene salvata come **evento immutabile** in un log
- Gli eventi vengono salvati in un database **append-only** (ad esempio, EventStore o Kafka)
- Lo stato corrente di un'entità, come un ordine, viene derivato da tutti gli eventi relativi. Per esempio:
  - **Ordine creato ➔** stato: "In attesa di pagamento"
  - **Pagamento effettuato ➔** stato: "In attesa di spedizione"
  - **Prodotti riservati ➔** stato: "Completato"
- Gli eventi vengono pubblicati su **un sistema di messaggistica** (es. Kafka, RabbitMQ), permettendo ai microservizi interessati di reagire in modo asincrono
- **Snapshot** dello stato corrente possono essere creati periodicamente per evitare di rigiocare tutti gli eventi ogni volta


Order Service
ORDER table
ID
STATUS
TOTAL
...
4567
ACCEPTED
84044.30
...
Order Service
Customer Service
add event
find events
Subscribe to events
Event Store
Order 2345
OrderCreated
Order Approved
...
OrderShipped

Pros

- Risolve uno dei problemi chiave nell'implementazione di un'architettura basata sugli eventi e consente di pubblicare in modo **affidabile** gli eventi ogni volta che cambia lo stato
- Poiché rende **persistenti** gli eventi anziché gli oggetti di dominio, evita principalmente il problema di mancata corrispondenza dell'impedenza tra oggetti e relazioni
- Fornisce un **registro di controllo** affidabile al 100% delle modifiche apportate a un'entità aziendale
- Consente di implementare **query temporali** che determinano lo stato di un'entità in qualsiasi momento

Cons

- Si tratta di uno stile di programmazione diverso e non familiare, quindi richiede una **curva di apprendimento**
- L'archivio eventi è difficile da interrogare poiché richiede **query specifiche** per ricostruire lo stato delle entità aziendali. Di conseguenza, l'applicazione deve utilizzare Command Query Responsibility Segregation (CQRS) per implementare le query

SERVICE REGISTRY

Service Registry

- Nelle architetture a microservizi, ogni servizio è un'entità **indipendente** che può essere distribuita su diverse macchine o container
- Poiché i microservizi possono **scalare dinamicamente** (ad esempio, aggiungendo o rimuovendo istanze), i loro **endpoint** (indirizzi e porte) possono **cambiare frequentemente**
- Per permettere ai microservizi di comunicare tra loro, è necessario un meccanismo che consenta di **individuare** in modo affidabile gli **endpoint** dei servizi disponibili
- Ad esempio, in un sistema di e-commerce, il microservizio "Ordini" deve invocare il microservizio "Pagamenti". Tuttavia, poiché le istanze di "Pagamenti" possono cambiare **dinamicamente**, "Ordini" ha bisogno di un metodo per trovarle senza dipendere da configurazioni statiche

Problems

- Come garantire che un microservizio trovi **dinamicamente** gli endpoint di altri servizi in un sistema distribuito?
- Come gestire i **cambiamenti frequenti** degli endpoint causati da scalabilità, guasti o spostamenti tra macchine?
- Come evitare **configurazioni statiche rigide** che richiedono interventi manuali ogni volta che un endpoint cambia?

Forces

- **Scalabilità dinamica**: I microservizi possono essere replicati o rimossi dinamicamente in base al carico, rendendo gli endpoint imprevedibili
- **Disponibilità**: Gli endpoint devono essere sempre aggiornati e disponibili per garantire una comunicazione affidabile tra i microservizi
- **Decoupling**: I microservizi devono essere il più indipendenti possibile, evitando di dipendere da configurazioni statiche o centralizzate
- **Fault Tolerance**: Il sistema deve continuare a funzionare anche in presenza di guasti parziali, come la disconnessione temporanea di un microservizio.

Solution

- **Registrazione dinamica**: Ogni microservizio registra automaticamente il proprio endpoint (IP, porta e metadati) nel registro quando si avvia.
  - Questo può essere fatto direttamente dal servizio o tramite un proxy (ad esempio, un sidecar).
- **Discovery dinamica**: I microservizi client interrogano il registro per trovare l'endpoint più recente di un altro servizio. Questo può essere eseguito in due modalità:
  - **Client-side discovery**: Il client si connette direttamente al registro per ottenere l'endpoint.
  - **Server-side discovery**: Un load balancer (es. API Gateway) si occupa della discovery per conto del client.
- **Monitoraggio e aggiornamento**: Il Service Registry monitora costantemente i servizi registrati, rimuovendo quelli non più disponibili (ad esempio, tramite un meccanismo di **health check**).

Resulting Context

- **Risoluzione dinamica degli endpoint**: I microservizi possono individuare facilmente altri servizi senza configurazioni statiche. Ad esempio, "Ordini" può ottenere l'endpoint aggiornato di "Pagamenti" interrogando il registro
- **Scalabilità migliorata**: Quando nuove istanze di un servizio vengono aggiunte o rimosse, il Service Registry aggiorna automaticamente il proprio stato
- **Fault Tolerance**: Il sistema rimane resiliente anche in presenza di guasti temporanei, poiché i client possono ottenere sempre endpoint validi
- **Decoupling tra microservizi**: I microservizi non hanno bisogno di conoscere direttamente gli endpoint degli altri servizi, riducendo le dipendenze
- **Efficienza operativa**: La gestione degli endpoint è centralizzata, semplificando la manutenzione e riducendo la necessità di intervento umano
- A meno che il registro dei servizi non sia integrato nell'infrastruttura, è un altro componente dell'infrastruttura che deve essere **impostato, configurato e gestito**. Sebbene i client debbano memorizzare nella cache i dati forniti dal registro dei servizi, se il registro dei servizi fallisce, quei dati alla fine diventeranno obsoleti. Di conseguenza, il registro dei servizi deve essere **altamente disponibile.**

HEALTH CHECK API

Health Check API

- In un sistema distribuito basato su microservizi, è essenziale sapere se ogni servizio è **operativo** per garantire la continuità del **funzionamento** complessivo
- I microservizi possono andare **offline** per molte ragioni, come guasti hardware, errori software o problemi di rete
- Per evitare che richieste vengano indirizzate a servizi non disponibili, è necessario un meccanismo per **monitorare lo stato di salute** dei microservizi
- Ad esempio, un microservizio "Ordini" deve comunicare con il microservizio "Pagamenti". Se "Pagamenti" non è disponibile, sarebbe **inefficiente** (e dannoso) inoltrare richieste che non possono essere elaborate

Problems

- Come determinare automaticamente se un microservizio è operativo o se è in uno stato **degradato**?
- Come garantire che solo i servizi **sani** ricevano richieste da altri microservizi o dal bilanciatore di carico (load balancer)?
- Come gestire **dinamicamente** la rimozione o il reintegro di servizi non disponibili?

Forces

- **Affidabilità del sistema**: È fondamentale evitare che richieste critiche vengano inviate a servizi non funzionanti
- **Automazione**: Il monitoraggio manuale dello stato dei microservizi non è pratico in ambienti dinamici
- **Minimizzazione dei falsi positivi/negativi**: Un servizio non disponibile deve essere rilevato rapidamente, ma senza segnalarlo erroneamente come guasto
- **Performance**: Gli health check devono essere leggeri per non aggiungere overhead significativo ai microservizi

Solution

- **Endpoint dedicato**: Ogni microservizio espone un endpoint, come /health o /status, che restituisce informazioni sul proprio stato. Lo stato può includere:
  - **Alive**: Il servizio è attivo e risponde.
  - **Ready**: Il servizio è pronto per elaborare richieste (es. ha completato la connessione ai database o ad altre dipendenze).
- **Monitoraggio continuo**: I sistemi esterni, come il Service Registry, il load balancer o strumenti di monitoring (es. Prometheus, Consul), interrogano periodicamente l'endpoint di health check.
- **Verifiche interne**: L'endpoint può eseguire controlli interni, come:
  - Connettività ai database.
  - Stato dei collegamenti con altri servizi critici.
  - Utilizzo delle risorse (es. memoria, CPU).
- **Risposte standardizzate**: Il risultato del check dovrebbe essere leggibile e standard
- **Rimozione automatica**: Un servizio ritenuto non disponibile viene temporaneamente escluso dal Service Registry o dal bilanciatore di carico, fino a che non torna in uno stato operativo

Resulting Context

- **Miglior resilienza**: Le richieste vengono indirizzate solo ai microservizi sani, riducendo downtime e malfunzionamenti
- **Automazione del monitoraggio**: I servizi non disponibili vengono rilevati automaticamente, senza intervento umano
- **Diagnosi veloce**: L'API di health check fornisce informazioni utili per identificare rapidamente le cause dei problemi (es. un database non raggiungibile)
- **Minore impatto sui client**: I client non vengono più esposti a errori derivanti da microservizi offline, migliorando l'esperienza utente complessiva
- **Scalabilità operativa**: In ambienti con centinaia di microservizi, l'health check automatizzato riduce il carico amministrativo

Università di Catania

Uni ct MATEMATICA E INFORMATICA

The End 

Alessandro Midolo, PhD

Dipartimento di Matematica e Informatica

Università di Catania

alessandro.midolo@unict.it

https://www.dmi.unict.it/amidolo/

A.A. 2024/2025