

# Information Systems Design and Big Data

## Introduzione

### Sviluppo Sequenziale - Cascata

- Ogni **fase** del progetto deve essere completata per andare alla successiva (sequenziale)
- Introdotto a partire dagli anni '70, molto utilizzato prima del 2000
- Composto da 5 fasi molto **distinte**, si comincia dalla stesura dei **requisiti** e si termina con la continua manutenzione dopo il rilascio
- Le fasi sono ben **documentate**, la gestione del progetto è molto **semplice**
- Gestione **lineare**, difficoltà nell'apportare modifiche una volta completata una fase, non possono esserci continue integrazioni con stakeholders/utenti/committenti

### Sviluppo Iterativo - Agile

- Le fasi sono raggruppate in qualche giorno (**sprint**), ognuna è **dinamica**
- Formalizzata nel 2001, utilizzata ancora oggi
- Modello che prevede un **ciclo iterativo** per ogni sprint, si comincia con la **pianificazione** e si finisce con il **rilascio**
- La gestione è estremamente **flessibile**, è possibile avere un **feedback continuo** dai clienti e migliorare conseguentemente
- La dinamicità dei progetti potrebbe complicare notevolmente la loro gestione

### Extreme Programming

- Concetto analogo di sprint
- Introdotto negli anni '90 e utilizzato ancora oggi
- Utilizza il concetto di **Test Driven Development**, una delle prime fasi è la scrittura di test automatizzati che permettano il continuo **refactoring** del codice (per comprenderne la leggibilità)
- L'integrazione continua è fondamentale, appena si integrano nuove funzionalità si effettuano i test, questo serve a rilevare subito eventuali problemi
- Largo utilizzo del **Pair Programming**, due sviluppatori lavorano sullo stesso pc, questo serve a creare codice migliore e condividere le conoscenze
- Le **release** sono **continue**, questo è coerente con tutto l'idea dell'XP, consente feedback continui

### Scrum

- *Basato* su agile, introdotto a metà anni '90
- 3 ruoli principali:

- **Owner**: definisce funzionalità e priorità
- **Master**: si assicura che si seguano le pratiche scrum
- **Dev team**: gruppo autogestito che completa gli obiettivi degli sprint
- Gli sprint sono una parte fondamentale, durano dalle 2 alle 4 settimane
- Si utilizzano i Backlog:
  - di **prodotto**: lista prioritaria di funzionalità e requisiti
  - di **Sprint**: lista di attività da completare nello sprint

## Cloud Native

La necessità di utilizzare il cloud computing si può associare a una serie di vantaggi assolutamente non irrilevanti:

- **DevOps e CI/CD**: è possibile **integrare e distribuire** continuamente, gli aggiornamenti sono rapidi, frequenti e non interrompono i servizi
- **Alta scalabilità**: si ottimizzano globalmente le risorse in base ai carichi, permettendo un utilizzo **efficiente** delle infrastrutture
- **Resilienza**: la progettazione deve prevedere **fault-tolerance** e auto-riparazione per garantire **affidabilità**
- **Hardware-agnosticismo**: infrastruttura non importante, si crea un **ambiente** apposito per evitare problemi di configurazione (dipendenze, pacchetti e simili)
- **Servizi gestiti**: si **delega** la gestione dei servizi a degli enti esterni (Azure, AWS, ...) per ridurre i costi operativi, le responsabilità, la gestione e tutto ciò che c'è "sotto" il software

Questo è possibile farlo grazie a due concetti fondamentali, ovvero:

- **Microservizi**: suddivisione in piccoli servizi indipendenti che vengono sviluppati e distribuiti separatamente
- **Containerizzazione**: utilizzo di ambienti appositi (container), sono isolati, contengono tutte le dipendenze di cui ha bisogno il software e permettono la portabilità

È possibile individuare 3 tipologie di architetture, da quelle tradizionali a quelle attuali:

- **Struttura monolitica**: architetture molto grandi, altamente accoppiate e difficili da scalare
- **Migrata su cloud**: architetture monolitiche portate su cloud con ambienti virtualizzati
- **Cloud Native**: architetture sviluppate appositamente per il cloud, divise in microservizi con l'utilizzo delle pratiche DevOps

## 12-factor

### 1) Base di codice

Avere una base di **codice tracciabile** che segue un **versioning** (similmente alla questione del kernel Linux, con major e minor) che serve per comprendere lo

sviluppo di un software. Questo serve per distribuire diverse versioni a partire dalla stessa base di codice (ad esempio per sviluppo, staging e produzione).

## 2) Dipendenze

La necessità di dichiarare e isolare in modo esplicito tutte le dipendenze. Di solito in un ambiente python è possibile installare tutte le librerie necessarie con:

```
pip install -r requirements.txt
```

Dove requirements.txt è un file contenente una lista di librerie e versioni utilizzate per far girare il software.

## 3) Configurazione

I file relativi alla configurazione devono essere separati dalla base di codice. È il caso di Token utilizzati da API o di informazioni sensibili che ovviamente non è possibile caricare su repository remote, di solito si risolve utilizzando variabili d'ambiente (su file .env) e si carico prima dell'esecuzione del codice.

## 4) Servizi di supporto

Sono servizi integrati nel software che

## 5) Build, Release, Run

Le fasi sono rigorosamente separate in costruzione e rilascio. Nel caso di docker:

- Build: docker build -t myapp:build .
- Release: docker tag myapp:build myapp:release
- Run: docker run -d --name myapp -p 80:80 myapp:release

## 6) Processi

L'applicazione viene eseguita come uno o più processi senza stato. Generalmente è il caso dello storing delle sessioni in una web app in un database, questo ci permette di scalare senza avere grossi problemi.

## 7) Port Binding

L'esposizione di un servizio tramite una porta specifica che può essere definita come variabile d'ambiente.

## 8) Concorrenza

## 9) Disposability

10) Dev/Prod Parity

11) Logs

12) Admin Processes

## Microservizi

Avere una serie di servizi in base alle funzionalità permette una serie di vantaggi, possono essere riassunti con:

- **Indipendenza** in sviluppo e deployment
- **Scalabilità selettiva**
- **Recupero dai guasti veloce**

## Serverless

Si riferisce all'idea di delegare la gestione della parte fisica a terzi (AWS, Azure), permettendo agli sviluppatori di occuparsi solo della parte software, con vantaggi quali:

- **Diminuzione delle responsabilità**
- **Scalabilità**
- **Efficienza dei costi**

Utilizzando questa filosofia i costi si riducono per diverse motivazioni:

- **Eliminazione di costi fissi:** si eliminano i costi di acquisto e gestione di un server fisico
- **Pay-as-you-go:** i provider fanno pagare in base all'utilizzo
- **Uso efficiente di risorse:** le risorse vengono allocate in base al carico, si cerca di minimizzare l'utilizzo complessivo

Unendo questi concetti con DevOps e CI/CD si può arrivare a:

- **Rilascio veloce:** il deployment è più veloce e con meno rischi
- **Processi automatizzati:** è possibile effettuare test, integrazione e deployment in modo automatico, consentendo di avere feedback in modo più rapido
- **Sperimentazione:** è possibile sperimentare senza aver paura di downtime o fallimenti di sistema

## Scalabilità

- **Orizzontale:** più istanze del servizio
- **Verticale:** più risorse per il servizio specifico
- **Flessibilità:** modifiche o aggiornamenti dell'app **senza downtime**

## Sistemi Legacy

Uno dei problemi più importanti è relativo al fatto che molti dei sistemi fossero pensati come **monoliti**, questo fa sì che negli step per portare in cloud sistemi legacy è necessario effettuare una filosofia simile:

- **Monoliti → Microservizi**
- Processi **manuali** → Processi **automatici**
- Infrastrutture **Rigide** → Infrastrutture **Flessibili**

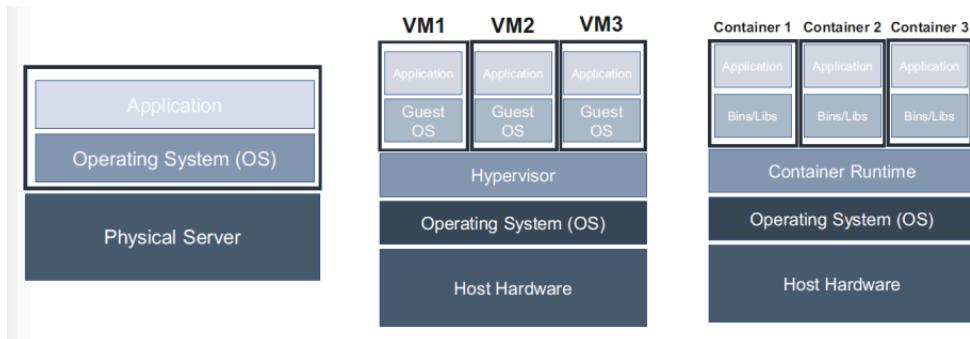
Al fine di effettuare questo passaggio è necessario introdurre la figura del **Software architect**, questo si occuperà di:

- Creare **sistemi resilienti** a fallimenti di singole componenti
- Creare sistemi che **permettano lo scaling**
- Costruire **sistemi sicuri** rispettando le normative correnti riguardo la **protezione dei dati**
- **Collaborare** con il team **DevOps** per migliorare l'integrazione e il deployment in tempi brevi, ottimizzando la pipeline

## Difficoltà di implementazione

- **Sicurezza:** gli ambienti devono essere **sicuri** e devono garantire la **privacy**
- **Vendor lock-in:** **difficoltà** nella **migrazione** di ambienti tra provider diversi
- **Skill gaps:** **difficoltà** nel **reperimento** di figure professionali **esperte** in tecnologie **cloud**

## Tipologie di Server



## Bare-metal

- Server **fisici** costituiti da una sola macchina che serve un carico di lavoro e opera in modo **indipendente**
- **Sistema operativo** installato su **macchina fisica** che ha bisogno di tutti i software
- **Alti costi di gestione**
- **Basso utilizzo**
- **Complessità di gestione**

## Server virtuali

Per risolvere parzialmente i problemi legati alla bassa efficienza dei server fisici si è cominciato a parlare di **virtualizzazione**, ovvero:

- **Astrazione** di hardware fisico in **risorse virtuali**
- Su una singola macchina girano **diversi OS** con diverse app
- Poche macchine fisiche
- **Possibilità** di effettuare **testing** senza dover acquistare macchine aggiuntive
- **Disaster recovery** grazie a degli **snapshot**

## Hypervisor

Layer che si occupa della virtualizzazione, si divide in:

- **Tipo 1:**
  - Installati direttamente sull'hardware fisico
  - Accesso **diretto** alle risorse hardware
  - Maggiore efficienza e prestazioni, manca un OS che intervenga a monte
  - Utilizzato soprattutto dove sono necessarie le prestazioni (data center)
  - Più sicuro, minori superfici di attacco, è necessario agire sull'hardware
  - Necessita di una figura professionale per il mantenimento della piattaforma fisica

- **Tipo 2**
  - Installato su un OS di una macchina
  - Le risorse che gli devono essere assegnate vengono gestite in primo luogo dall'OS principale
  - Innegabile facilità di utilizzo, impiegato principalmente per sviluppo o testing, consente di avere molteplici OS (generalmente con GUI)
  - Meno sicuro, ha più superfici di attacco, se viene compromesso l'OS host si compromette tutto
  - Gestione molto più facile

## Benefici

Tra i benefici riscontrabili con l'utilizzo delle VM abbiamo:

- **Efficienza** delle risorse in base alla richiesta istantanea
- **Isolamento** di ogni VM
- Scalabilità, quindi anche la possibilità di poter aggiungere o rimuovere istanze
- **Continuità operativa** (di conseguenza anche il disaster recovery e la resilienza)
- Possibilità di poter fare **testing** senza impattare sull'ambiente nel completo
- **Diminuzione dei costi** dovuti alla gestione fisica delle infrastrutture
- **Compatibilità**, dovuta alla convivenza di diversi OS nella stessa macchina

## Tipi di virtualizzazione

In base al funzionamento di base la virtualizzazione può essere divisa in:

- **Completa**: L'hypervisor gestisce le risorse hardware e le ripartisce tra le varie VM
- **Paravirtualizzazione**: collaborazione tra l'OS host e le VM, serve a migliorarne l'efficienza
- **A livello di Os**: le VM vengono eseguite sopra l'OS host, maggiore overhead

È possibile fare separazione anche relativamente al modo in cui si gestisce la rete:

- **Con rete interna**: suddivisione della rete fisica in più reti virtuali
- **Con rete esterna**: connessione di diverse reti fisiche attraverso una virtuale

## Virtualizzazione su cloud

## Container

Tecnologia di **virtualizzazione** che permette l'esecuzione di applicazioni in **ambienti isolati** e in modo più **efficiente**. Si è cominciato a parlare di container dall'introduzione di **docker**, che ne ha permesso la diffusione in modo molto ampio in quanto ha semplificato le operazioni di creazione, distribuzione e gestione di container

## Principi

- **Isolamento**: gli ambienti sono separati per ogni applicazione, virtualizzazione delle risorse a livello di OS. Non ci sono conflitti di dipendenze
- **Efficienza**: necessità di meno risorse rispetto a un OS (ne condivide il kernel) e minori tempi di avvio (e caricamento)

- **Scalabilità**: facilità di aprire o chiudere istanze di container in base alla domanda
- **Portabilità**: agnosticismo rispetto all'OS host, basta che si abbia un ambiente che supporti i container
- **Riduzione dei costi**: meno risorse impiegate vuol dire anche minori costi fissi di gestione
- **Agilità**: l'utilizzo dei container permette di migliorare il flusso di lavoro, c'è fault tolerance e semplifica il deployment

## Deployment

Tra le possibilità di effettuare deployment abbiamo:

Caratteristiche	Rolling	Blue/Green
<b>Definizione</b>	<b>Aggiornamento graduale</b> dell'app su un numero di server o container per volta	Creazione di due <b>ambienti identici</b> , la <b>blu</b> con la versione attuale e la <b>verde</b> con quella nuova
<b>Downtime</b>	<b>Minimale</b> , ci sono istanze disponibili durante l'aggiornamento	<b>Nulli</b> , l'ambiente viene testato prima della sostituzione complessiva
<b>Rischi</b>	<b>Rischio moderato</b> , possono esserci versioni diverse in esecuzione	<b>Rischi bassi</b> , l'ambiente viene testato prima del completo passaggio alla versione nuova
<b>Complessità</b>	<b>Semplice</b> da implementare (soprattutto con pochi nodi)	<b>Complesso</b> da implementare, bisogna controllare e gestire le versioni
<b>Scalabilità</b>	Adatto ad ambienti in cui ci sono <b>aggiornamenti frequenti</b> e <b>scalabilità continua</b>	Adatto ad ambienti in cui è importante la <b>stabilità</b> e la <b>disponibilità</b> , gli aggiornamenti non sono così tanto frequenti
<b>Undo</b>	<b>Difficile</b> , l'aggiornamento viene eseguito gradualmente	<b>Facile</b> , c'è sempre la versione precedente (blue)

## Orchestrazione

Per **orchestrazione** si intende **l'automizzazione** del ciclo di vita dei container nel totale, fino all'eliminazione. È particolarmente utile quando si parla di ambienti con centinaia/migliaia di container. Un esempio di orchestratore è Kubernetes.

Tra i vantaggi abbiamo:

- **Miglioramento della scalabilità**
- **Diminuzione del lavoro manuale** (automazione)
- **Miglioramento dell'affidabilità** in quanto si occupa della distribuzione e della gestione dei container
- **Riduzione dei costi** in seguito a una gestione efficiente delle risorse

## Monitoring e Logging

Il **monitoring** serve per poter avere sotto controllo le **risorse** e lo **stato di “salute”** dei container. Tra gli aspetti chiave abbiamo:

- **Metriche**
  - **Usage**: CPU, mem, disco in percentuale, per evitare bottleneck e ottimizzare l'allocazione
  - **Uptime**: controlli sullo stato di attività dei container, diminuiscono i tempi di inattività
  - Response-time: misurazioni sul tempo impiegato dal servizio per rispondere
- **Alerting**
  - **Threshold**: alert triggerati dal superamento di certe soglie come l'utilizzo della cpu per troppo tempo
  - **Incident Response**: risposte automatiche in caso di problemi per diminuire i danni sull'utente finale
- **Visualizzazione**
  - **Dashboard**: interfacce per mostrare le metriche in tempo reale
  - **Analytics**: interfacce per poter identificare tendenze ed effettuare previsioni

Il **logging** invece prevede la **raccolta** di **dati** sui processi in esecuzione con il fine di poter effettuare in modo semplice debug e altri tipi di **analisi**

## Security e best practices

Esistono delle best practices per assicurarsi che l'ambiente sia sicuro, tra queste abbiamo:

- **No-root**: evitare l'utilizzo di container con privilegi
- **Segmentazione di rete**: separazione la comunicazione tra i container
- **Aggiornamenti frequenti**: i container devono essere aggiornati e possibilmente con le ultime patch di sicurezza
- **Immagini affidabili**: si scaricano le immagini solo da fonti verificabili e affidabili
- **Restrizioni su risorse**: limiti su CPU, mem o throughput per evitare qualsiasi tipo di attacco
- **Sicurezza a livello host**: utilizzo di firewall, antivirus e simili
- **Controlli di accesso**: Implementazione di controlli rigorosi per far accedere solo agli utenti autorizzati

## Modelli di servizi su cloud

### IaaS (Infrastructure as a Service)

Con IaaS si intende un **servizio** che fornisce le **risorse di calcolo** necessarie, ad esempio in forma di server virtuali, in modo **dinamico**. Questo permette di poter utilizzare solo lo stretto necessario.

Tra le caratteristiche abbiamo:

- Possibilità di **scalare** le risorse in base alla domanda
- **Eliminazione** dei costi dell'**hardware fisico**
- **Flessibilità** in base alla domanda
- **Accessibilità** da qualsiasi luogo con banda internet
- **Complessità di gestione e configurazione**
- **Costi di manutenzione nel tempo**

- **Sicurezza** che dipende **dall'utente** (e in parte anche dal **fornitore**)

## PaaS (Platform as a Service)

Con PaaS si fa riferimento a una piattaforma di sviluppo e hosting per applicazione. Questa include anche tutto il necessario, ovvero strumenti di sviluppo ad hoc, database e server. Nelle caratteristiche troviamo:

- **Strumenti e ambienti forniti**
- **Bassi costi di gestione e manutenzione**
- Possibilità di **scalare** in base alla domanda
- **Accessibilità** da qualsiasi luogo con banda internet
- **Dipendenza dal fornitore**
- **Difficoltà di integrare** l'app con altri sistemi
- **Sicurezza** che dipende dal **fornitore**

## SaaS (Software as a Service)

Con PaaS si offre un software su cloud, accessibile tramite internet e utilizzabile con un abbonamento.

Nelle caratteristiche troviamo:

- **Accessibilità** da qualsiasi luogo con banda internet
- Si delegano le **installazioni** e le **manutenzioni** dei software
- **Elimina i costi di manutenzione e aggiornamento**
- Possibilità di scalare in base alla domanda
- L'utente **dipende dal fornitore** sia per la qualità che per la stabilità
- **Poca personalizzazione**
- La **sicurezza** dipende soprattutto dal **fornitore**

## Confronto

Caratteristiche	IaaS	PaaS	SaaS
Scalabilità	Alta	Alta	Alta
Costi Iniziali	Bassi	Bassi	Bassi
Gestione	Alta	Media	Bassa
Personalizzazione	Alta	Media	Bassa
Sicurezza	Dipende dall'utente	Dipende dal fornitore	Dipende dal fornitore

## SaaS Multi-Tenant

Sempre a proposito dell'architettura SaaS è possibile utilizzare un approccio **Multi-Tenant**, ovvero un modello per cui **molteplici clienti** usufruiscono del servizio **condividendo** la stessa infrastruttura ma in **modo isolato**. La figura del **cliente** viene chiamata **"tenant"**. Tra le sue caratteristiche abbiamo:

- **Condivisione delle risorse** intrinseca all'idea del modello, permette di **condividere i costi operativi**
- **Facilità di scalare orizzontalmente**
- **Isolamento dei dati dei tenant**
- **Personalizzazione** di ogni istanza senza influenzare gli altri tenant, ovviamente più limitata rispetto a una soluzione dedicata
- Aggiornamenti **centralizzati**
- **Manutenzione ridotta**
- Aggiornamenti **senza interruzioni** del servizio
- **Limitazione delle prestazioni** rispetto a una soluzione dedicata
- **Problemi di vulnerabilità** sono comuni a tutti i tenant

## Confronto

Caratteristiche	Multi-tenant	Single-tenant
<b>Costo</b>	Più basso grazie alla condivisione delle risorse	Più alto, richiede infrastruttura dedicata
<b>Scalabilità</b>	Alta, facile da scalare	Moderata, richiede più risorse dedicate
<b>Isolamento dei Dati</b>	Logico, non fisico	Fisico, maggiore sicurezza e isolamento
<b>Personalizzazione</b>	Limitata	Elevata, permette configurazioni specifiche
<b>Aggiornamenti</b>	Centralizzati e uniformi	Personalizzati, possono richiedere più tempo

## Modelli

Più nello specifico è possibile classificare ulteriormente i modelli in base alla **separazione** (logica e fisica) delle **risorse**

### Pooled

Chiamato anche “**condiviso**”, è un modello in cui le **risorse** (fisiche e virtuali) vengono **condivise** tra tutti i tenant. Questo consente di utilizzare in modo **efficiente** l'infrastruttura.

Tra le caratteristiche abbiamo:

- Facile **scalabilità**
- Facile **gestione operativa**
- **Efficienza nei costi**
- **Isolamento logico** ma non fisico
- **Prestazioni che variano** in base agli altri tenant

Proprio a proposito dell'ultimo punto è utile parlare di un problema di cui potrebbe soffrire questo modello, cioè il “noisy neighbor”. Questo si riferisce a una situazione in cui uno o più tenant utilizza le risorse in modo intensivo, peggiorando l'esperienza degli altri tenant

## Siloed

Questo modello utilizza una filosofia opposta rispetto a quella del modello precedente. Ogni tenant utilizza delle risorse separate sia fisicamente che logicamente. Tra le caratteristiche abbiamo:

- Maggiore **sicurezza e privacy**
- **Prestazioni non influenzabili** da altri tenant
- **Costi maggiori e sistemi meno efficienti**
- **Scalabilità più complicata** (le risorse andrebbero valutate prima)

## Ibrido

Questo approccio combina elementi del modello pooled e altri del modello siloed. Permette ad esempio di rendere alcuni servizi più facilmente scalabili e altri meno soggetti ai carichi degli altri utenti.

Tra le caratteristiche abbiamo:

- **Alta flessibilità**, permette il giusto bilanciamento a livello di **costi, di sicurezza e prestazioni**
- **Utilizzo efficiente** delle risorse senza precludere l'isolamento
- Notevole **complessità di gestione**

## Comparazioni

Caratteristiche	Pooled	Siloed	Ibrido
<b>Efficienza dei Costi</b>	Alta	Bassa	Media
<b>Isolamento dei Dati</b>	Basso	Alto	Medio
<b>Prestazioni</b>	Variabili	Stabili	Equilibrate
<b>Scalabilità</b>	Alta	Limitata	Flessibile
<b>Complessità di Gestione</b>	Semplice	Complessa	Complessa

## Modelli di deployment

### Public Cloud

Modello di cloud computing in cui le **risorse** vengono **offerte** da un **fornitore** tramite internet e vengono condivise da più utenti. Questi sono:

- **Accessibili** da qualunque posto con una connessione internet
- **Costi bassi, divisi** tra più utenti
- Facile **scalabilità**
- **Manutenzione** a carico del **fornitore**
- La **sicurezza** dipende sia dal **fornitore** che dalla **configurazione dell'utente**
- Bassa **personalizzazione**

## Private Cloud

Ambiente di cloud computing **dedicato** a una sola organizzazione, con risorse in base alle necessità. Tra le caratteristiche abbiamo:

- Maggiore **sicurezza** (data dall'**isolamento fisico**)
- Maggiore **personalizzazione**
- **Prestazioni** non influenzabili da altri utenti
- **Costi** elevati
- I **costi di gestione** sono elevati, necessarie delle **figure competenti** rispetto a quelle tecnologie

## Hybrid Cloud

**Combina** elementi di public cloud e private cloud, permette di utilizzare risorse locali e in cloud

- Bilancia **sicurezza, costi e prestazioni** in modo eccellente
- Facile **scalabilità** (soprattutto per le risorse online)
- Maggiore **resilienza e continuità operativa** garantita
- Complessità di **gestione** e di **integrazione** con servizi già esistenti
- **Costi** variabili (sia per la parte fisica che in cloud)

## Community Cloud

Modello di cloud computing in cui c'è una **condivisione delle risorse** basata su **requisiti operativi e normativi**. È il caso di **organizzazioni** come quelle statali, di finanza e altre che hanno bisogno di ambienti simili (soprattutto per quanto riguarda la **sicurezza**).

- **Costi** condivisi tra le organizzazioni
- **Sicurezza** maggiore (in quanto bisogna che si rispettino determinate **normative**)
- **Personalizzazione** più limitata
- Necessità di **gestire l'ambiente** per tutte le organizzazioni

## Confronto

Caratteristiche	Public Cloud	Private Cloud	Hybrid Cloud	Community Cloud
<b>Costo</b>	Basso	Alto	Variabile	Medio
<b>Sicurezza</b>	Media	Alta	Alta	Alta
<b>Scalabilità</b>	Alta	Media	Alta	Alta
<b>Personalizzazione</b>	Bassa	Alta	Alta	Media
<b>Gestione</b>	Gestita	Autonoma	Complessa	Condivisa
<b>Accessibilità</b>	Alta	Limitata	Alta	Alta

## Cloud Deployment e sicurezza

# Principi di architettura

I principi di architettura sostanzialmente servono per gestire la complessità dei sistemi, gestendo i requisiti business e quelli tecnici. Definiscono la struttura, le interazioni e le relazioni tra le componenti. Attualmente è noto il principio API-First:

- API esposte per usufruire di un servizio
- Accessibili tramite interfacce REST HTTP
- Definizione delle proprietà a priori
- Servono come "contratto"
- Permettono la modularità e il riciclo di componenti già ingegnerizzate

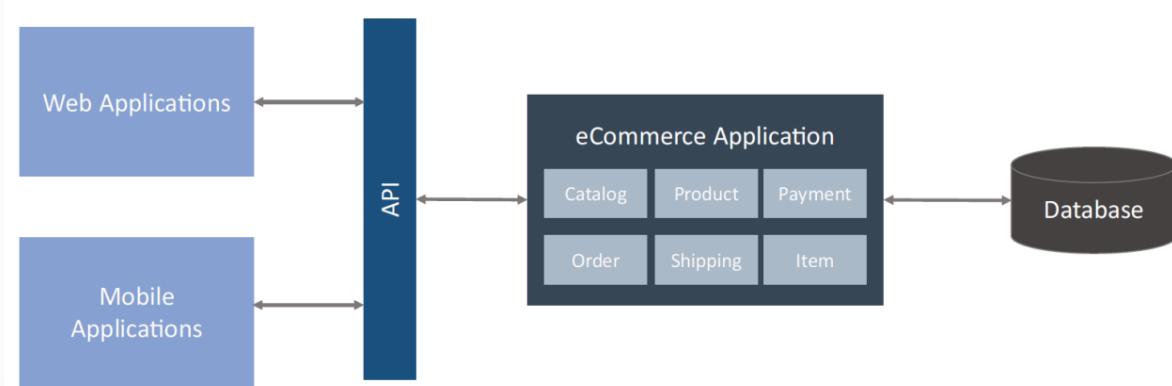
## Architetture monolitiche

L'approccio classico era basato su architetture di tipo monolitico. Questo consisteva in:

- Componenti altamente accoppiate
- Condivisione di risorse fisiche (anche la memoria)
- Esposizione di API direttamente al client

Se da un lato è un approccio tutto sommato accettabile per progetti piccoli, da un altro mostra tutti i suoi problemi quando si vuole progettare un'applicazione più grande, questo perché:

- Difficoltà a scalare, se ho bisogno di più risorse anche solo per una parte del servizio dovrò migliorare tutto
- Complessità, per effettuare cambiamenti in una parte dovrò avere una conoscenza molto profonda del funzionamento



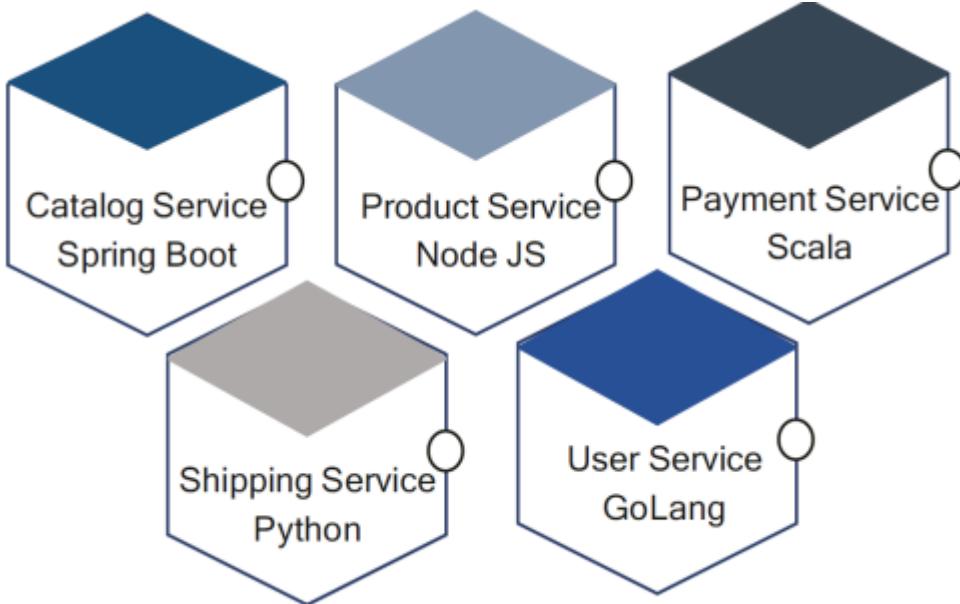
## Architetture Polilitiche

Il primo passo in avanti è stato compiuto attraverso un utilizzo diffuso di architetture polilitiche. Queste hanno:

- Componenti indipendenti, si possono costruire, deployare e scalare indipendentemente
- Favoriscono un approccio modulare

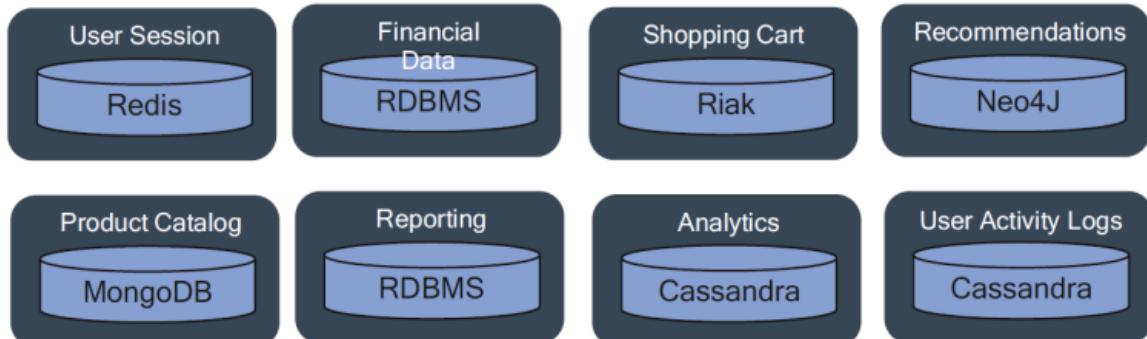
Risolvono anche i due problemi affrontati con le architetture monolitiche, ovvero:

- Facile scalabilità, questo perché ogni singolo componente può essere scalato indipendentemente, in base alla richiesta
- Autonomia, ogni team si può occupare di un singolo componente, senza conoscere approfonditamente tutta la struttura



## Principio della persistenza poliglotta

Durante la progettazione bisogna valutare anche il tipo di database in base all'applicazione nello specifico.



In questo modo miglioriamo sia l'efficienza che la modularità. Tra i consigli ci sono:

- DB relazionali per sistemi basati su transazioni
- DB NoSQL per sistemi con alto throughput, favorisce la scalabilità
- DB a grafi per gestire relazioni e query complesse

## Domain Driven Design

Alla base del domain driven design c'è l'idea per cui l'app si sviluppa a partire dalla conoscenza approfondita del dominio, unendo dunque la logica implementativa a quella business. Questo sicuramente aiuta perché:

- Si disaccoppiano i sistemi, serve solo conoscenza specifica riguardo il proprio task

- Si possono utilizzare le stesse soluzioni in ambienti simili

## Principio del Consumatore al primo posto (Consumer first)

Il consumatore diventa il centro delle decisioni aziendali, si comincia la progettazione dalla formalizzazione dei servizi tramite API con il fine di comprendere le necessità dell'utente e rendere di conseguenza il prodotto facilmente utilizzabile. L'idea alla base è quella per cui bisogna fidelizzare il proprio cliente. Bisogna però rispettare:

- Standard e coerenza, ovvero tutte le scelte già affermate e che gli altri utenti percepiscono come standard
- Documentazione per semplificare il processo di apprendimento

## Principio della decentralizzazione (Decentralize Everything)

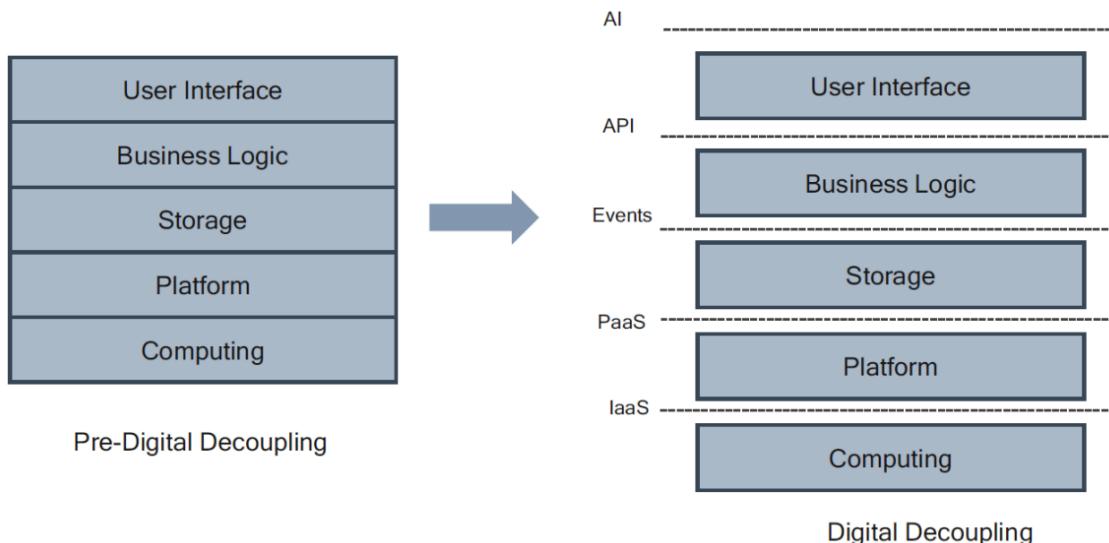
Una practice importante è quella della decentralizzazione, questa ha l'obiettivo di distribuire il controllo tra più sistemi. Questo ha una serie di vantaggi, come:

- Possibilità di effettuare testing
- Utilizzo di Domain-Driven Design, possibilità di gestire contesti ontologicamente differenti
- Deployment indipendente
- DevOps decentralizzato, ogni gruppo può lavorare a dei pezzetti di un sistema complesso
- Decentralizzazione della gestione dei dati (ovvero il principio della persistenza poliglotta)

## Principio del disaccoppiamento digitale (DPP)

Secondo questo principio tutte le componenti digitali devono essere separate, in modo tale da permettere lo sviluppo, la gestione e la modifica in modo indipendente. L'obiettivo di questo approccio è ridurre i rischi legati alla transizione di ambienti legacy. Esistono delle strategie a proposito:

- Utilizzare lo sviluppo cloud native per sviluppare rapidamente microservizi
- Utilizzo di API first e Consumer first
- Implementazione di interazioni real-time basato sui comportamenti dell'utilizzatore
- Isolamento delle infrastrutture



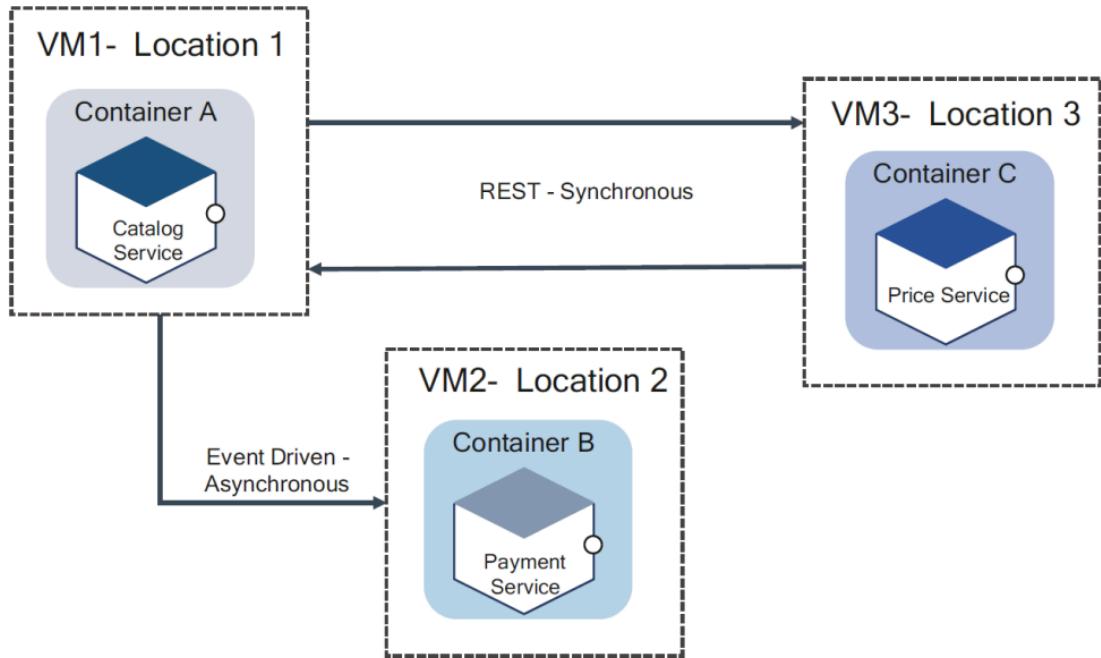
## Principio dell'isolamento isolato (IFP)

Principio che suggerisce l'isolamento delle componenti di un sistema per evitare che un guasto abbia effetto sull'intero sistema. Questo da sicuramente una serie di vantaggi notevoli:

- Sistema resiliente
- Protezione dal Single point of failure, il sistema non si interrompe quando c'è un guasto
- Continuità operativa nonostante alcune componenti hanno problemi

## Principio dell'indipendenza di locazione (LIP)

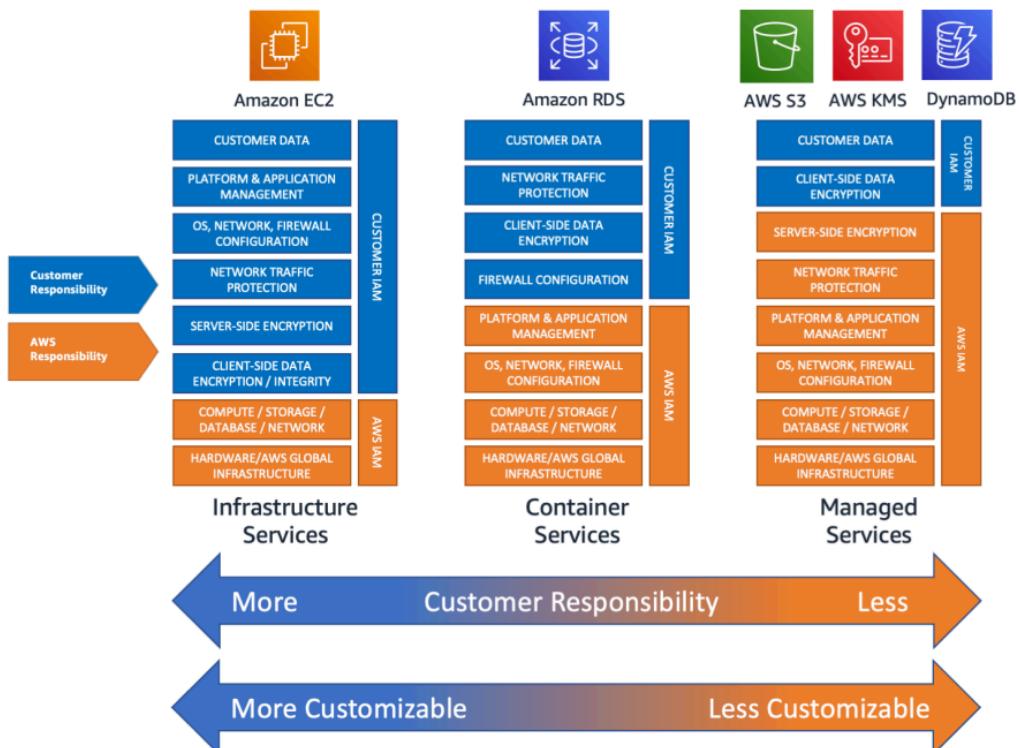
Principio che sostiene l'accessibilità alle risorse e ai servizi senza vincoli di accesso relativi alla posizione fisica del deployment. Chiaramente bisogna affrontare i limiti tecnologici e normativi, quindi bisogna gestire più ambienti distribuiti geograficamente (per avere sempre una bassa latenza) e rispettare le normative vigenti in fatto di gestione dati



## Principio della Difesa in Profondità

Secondo questo principio devono esistere diversi livelli di sicurezza nei sistemi cloud native per assicurare una buona protezione. Sostanzialmente questo approccio permette di rendere davvero complesso qualsiasi tipo di attacco, per cui se si buca un livello bisogna farlo anche con gli altri. È possibile farlo con:

- Sicurezza di rete: firewall, gruppi di sicurezza e VPN
- Sicurezza a livello applicativo: secure coding, autenticazione e criptaggio dei dati
- Criptaggio dei dati: questo vale sia mentre i dati sono in transito che quando vengono conservati



In questo esempio possiamo vedere diversi approcci in base alle responsabilità. A partire da sinistra, con EC2 il sistema è altamente personalizzabile ma lascia tutte le responsabilità all'utente, a destra invece servizi come S3 che danno meno responsabilità e meno personalizzazione.

## Principio della Security by Design

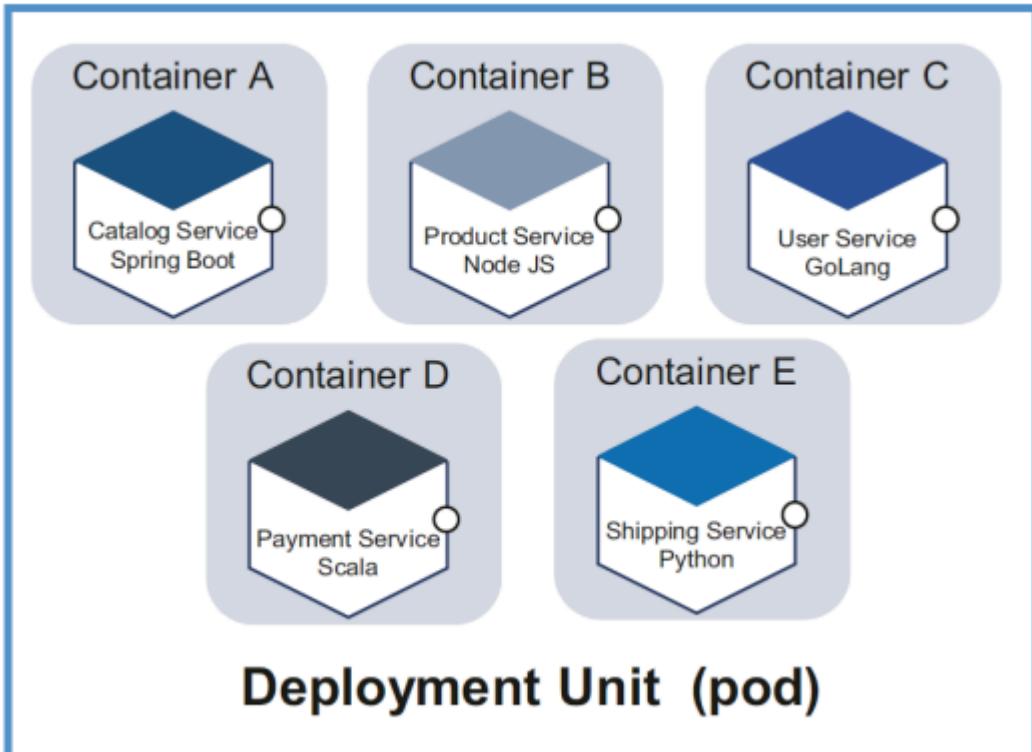
Principio per cui la sicurezza è il fulcro della progettazione e questa viene implementata fin dai primi momenti di vita del progetto. Questo è in netto contrasto con le pratiche “classiche” di programmazione. Tra le best practices troviamo:

- Minimizzare le superfici di attacco
- Regole forti per accesso e registrazione
- Privilegi essenziali
- Sicurezza a livelli
- Fail-Security
- Separazione dei compiti
- Shift-left (anticipare l'integrazione della sicurezza nello sviluppo)

## Principio del singolo interesse (SCP)

Ogni container deve avere una sola responsabilità, permettendo a ogni microservizio di avere un solo ruolo ben specifico. Ovviamente bisogna rispettare:

- Confini ben precisi, questo in base a dei criteri che vedremo dopo (relativamente a coesione e accoppiamento)
- Possibilità di riutilizzo di un container per poter effettuare un deployment consistente
- Il design deve seguire specifici pattern per cercare di rispettare il concetto per cui ogni singolo container ha una sola responsabilità



## Principio dell'alta osservabilità

Ogni sistema deve essere progettato per poter tenere traccia di tutti gli eventi che accadono, sia in tempo reale che successivamente. Questo è possibile mediante il monitoraggio e il logging e permette di capire lo stato di salute di un servizio. Esistono anche delle metriche per effettuare delle valutazioni

## Principio di Conformità del Ciclo di Vita

I container devono essere conformi (ovvero rispettare il motivo per cui sono stati creati) durante tutto il loro ciclo di vita. Bisogna gestire:

- Graceful shutdown: chiusura “tranquilla”, dà il tempo di eseguire tutte le routine prima di interrompere il servizio
- Forceful shutdown: se il processo non si interrompe rispettando le tempistiche, riceve un comando per l'interruzione istantanea
- PreStop e PostStart:
  - PreStop: pulire tutte le risorse prima dell'interruzione
  - PostStart: routine di inizializzazione

## Principio dell'immutabilità dell'immagine (IIP)

Questo principio si basa sul fatto che una volta buildati, le immagini dei container siano di fatto non più modificabili. Se si volesse fare una modifica occorre rebuildare tutto il progetto. Per quanto possa sembrare un conto, questo in realtà dà molteplici benefici:

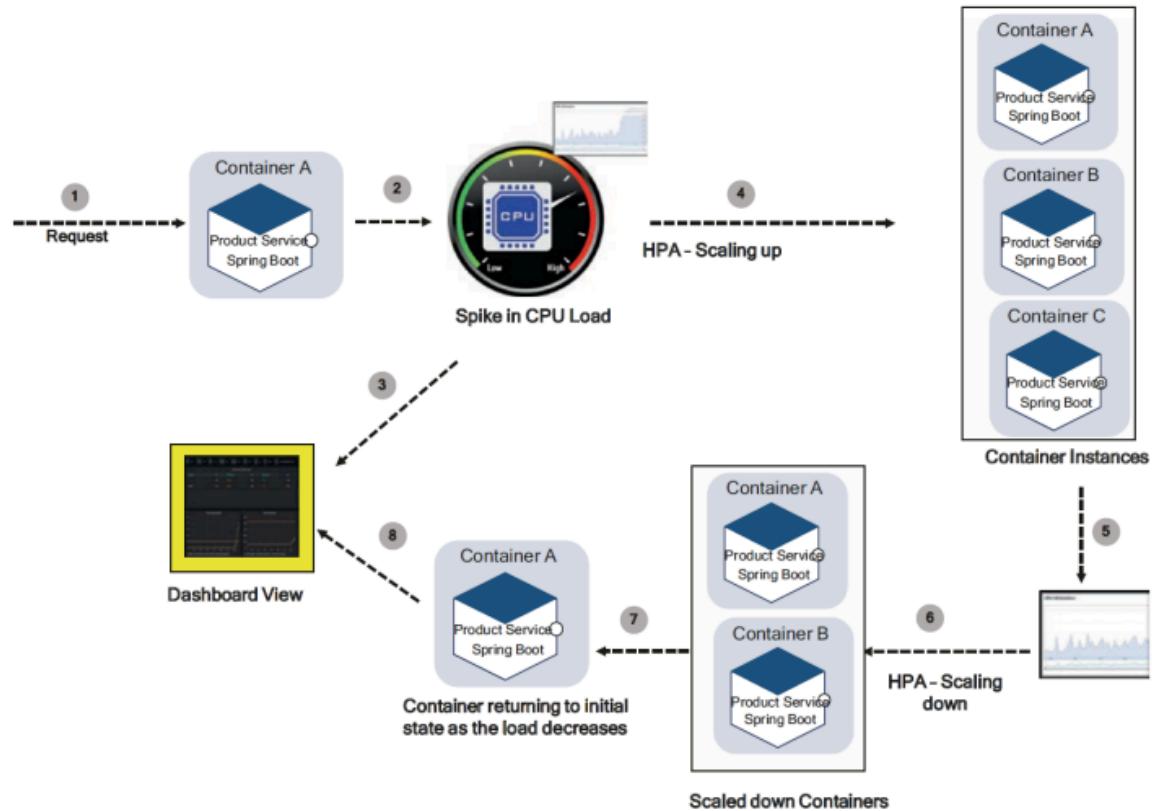
- Applicazioni immutabili: le applicazioni di fatto sono fatte per essere immutabili

- Possibilità di gestire le configurazioni: le variabili e le configurazioni vengono gestite esternamente al container
- Sicurezza nel deployment: il deployment deve essere di fatto ripetibile e il rollback si riduce al deploy dell'immagine vecchia
- Consistenza tra ambienti: la stessa immagine può essere deployata in ambienti diversi senza problemi

## Principio della Sacrificabilità (disposability) dei Processi (PDP)

Alla base di questo principio c'è il concetto di **container effimero**, ovvero la facilità di rimpiazzo di qualunque istanza di un container. Quindi:

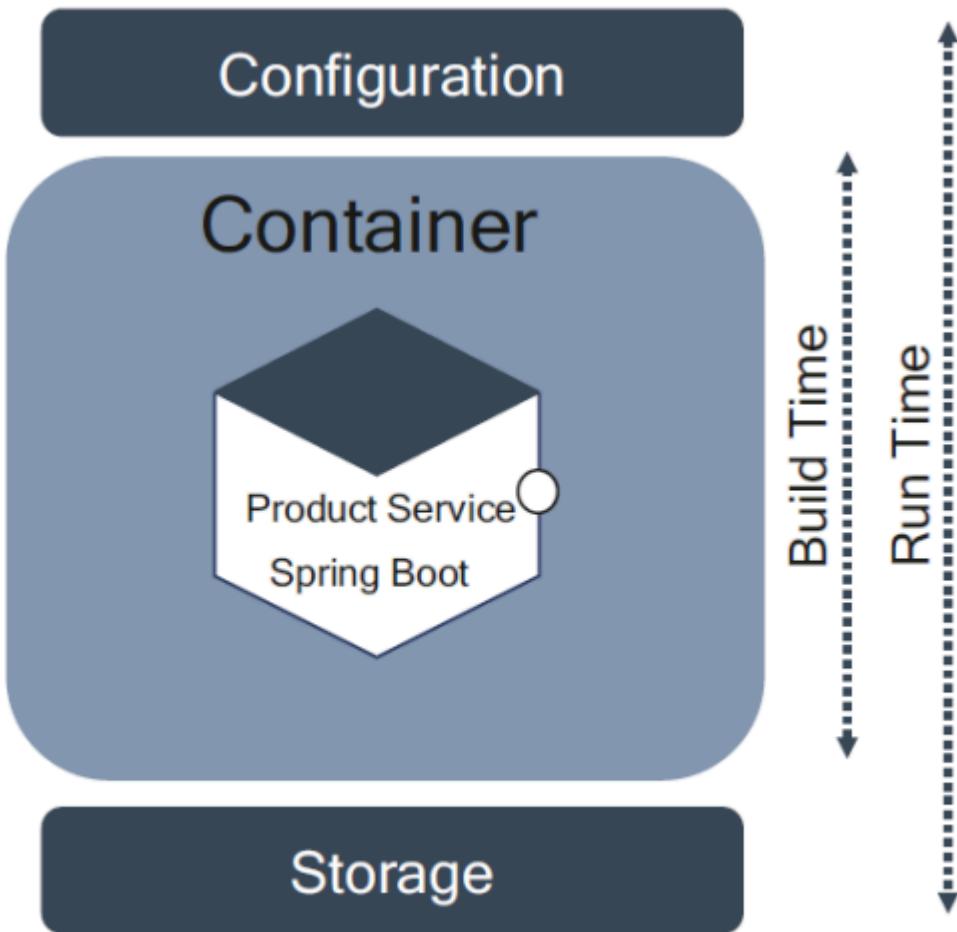
- Ogni servizio deve essere sacrificabile e progettato per un avvio (e un'interruzione) veloce
- È un approccio particolarmente utile relativamente alla scalabilità e alla resilienza, il riscontro è rapido
- Supporta i pattern cloud native tipo l'auto scaling



## Principio dell'Auto-contenimento

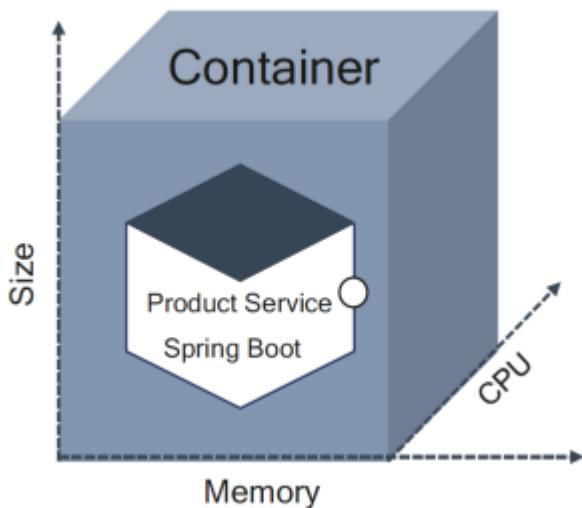
Ogni componente dovrebbe avere all'interno di sé tutto il necessario per funzionare in modo indipendente. Questo permette di ridurre la dipendenza rispetto ad altri componenti.

Ovviamente ogni container ha la propria configurazione, le proprie dipendenze e il proprio linguaggio



## Principio di contenimento in runtime

Ogni container dichiara i propri requisiti in termini di risorse e li comunica alla piattaforma. Questo vale relativamente all'utilizzo della CPU, la memoria e le dimensioni dello storage. Chiaramente questo è necessario per coordinare la gestione complessiva delle risorse permettendo lo scheduling e l'autoscaling. Sarebbe consigliato rispettare i requisiti altrimenti si rischia la terminazione o la migrazione.



## Qualità del software

### Don't Repeat Yourself (DRY)

Ogni componente deve essere pensata per essere riutilizzata, evitando di ripetere del codice.

- Riduce la ridondanza del codice
- Semplifica sia la manutenzione che gli aggiornamenti
- I moduli (librerie) devono avere poco codice
- Un riutilizzo eccessivo può portare a problemi di dipendenze nascoste
- Alcune duplicazioni sono inevitabili (e devono essere gestite attentamente)
- Difficile da implementare in prodotti legacy
- Richiede una conoscenza approfondita del sistema

Tra le best practices abbiamo:

- Refactoring del codice per risolvere le duplicazioni
- Largo utilizzo di librerie
- Documentazione
- Versioning semantico

## Isolamento nelle architetture Cloud Native

I cambiamenti a un modulo non impattano gli altri moduli. Ognuno di questi è responsabile per il proprio stato ed è accessibile solo tramite API. E' in un container e in caso di fallimento non danneggia tutto il sistema.

- Miglioramenti alla resilienza
- Aumenta la sicurezza
- Permette la gestione indipendente
- Aumenta la scalabilità
- Aumenta la complessità
- Necessita di una gestione efficace delle risorse

Tra le best practices abbiamo:

- Containerizzazione di tutte le componenti del sistema

- Adozione di un'architettura a microservizi
- Utilizzo di strumenti di monitoring e logging

## Separazione degli interessi (SoC)

Ovvero il disaccoppiamento di tutte le componenti del sistema in moduli isolati che si occupano solo delle loro responsabilità.

- Modularità
- Facilita la manutenzione del codice
- Necessita di un'architettura ben pensata
- Aumenta il tempo di sviluppo

Tra le best practices abbiamo:

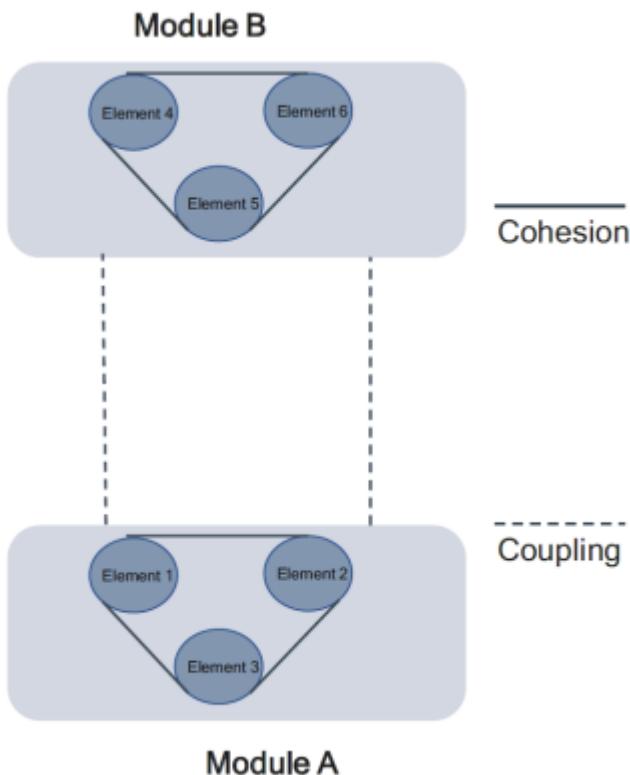
- Necessità di strutturare il codice a moduli ben definiti
- Definizione di interfacce chiare tra i moduli

## Livelli nell'architettura Cloud Native

L'organizzazione di un'applicazione viene effettuata in layer verticali e orizzontali:

- Presentazione → Presentazione: Interfacce web e mobile
- Business → Integrazione: Architettura Event-Driven per lavori in batch
- Persistence → Legacy: Integrazioni con app enterprise
- Database → SaaS terze parti: Funzionamento con servizi esterni

## Principi di design ortogonale



## Coesione

- Si assicura che ogni **modulo** lavora a dei **compiti ben definiti**.
- Misura quanto **fortemente** sono **collegati** gli elementi all'interno dei moduli
- L'obiettivo è avere una coesione il più alta possibile, rappresentabile come:
  - Ogni **modulo** è **specializzato** su una **funzionalità**
  - Funzionalità **facili** da comprendere e mantenere
  - **Migliore riusabilità** tra i moduli

Ci sono diversi livelli di coesione, partendo da quello preferibile abbiamo:

- **Funzionale**
  - Ogni elemento nel modulo contribuisce a un compito specifico
- **di Sequenza**
  - L'output di un elemento diventa l'input di un altro modulo (e così via)
- **Comunicazionale**
  - Gli elementi lavorano su dati comuni o referenziano le stesse cose
- **Procedurale**
  - Il raggruppamento viene fatto per ordine di esecuzione, ignorando la loro relazione
- **Temporale**
  - Elementi che vengono raggruppati perché vengono eseguiti contemporaneamente
- **Logica**
  - Elementi raggruppati perché effettuano compiti simili (le interfacce diventano difficili da seguire)
- **Coincidentale**
  - Raggruppamento casuale, senza alcuna relazione sensata

## Accoppiamento

- Riduce le interdipendenze tra i moduli, rendendoli flessibili
- Misura il grado di indipendenza tra i moduli
- Si cerca di avere un accoppiamento più basso possibile che serve per:
  - Ogni cambiamento in un modulo non ha impatti sugli altri moduli
  - Aumenta la manutenibilità del codice
  - Migliora la estendibilità, permettendo di estendere i moduli indipendentemente
  - Migliora la testabilità, permettendo di testare singolarmente i moduli

Ci sono diversi livelli di accoppiamento, partendo da quello preferibile abbiamo:

- **Disaccoppiamento**
  - I moduli non comunicano né dipendono tra di loro
- **di Messaggio**
  - I moduli comunicano senza passare dei dati come parametri
- **di Dati**
  - I moduli comunicano scambiandosi dei dati semplici come parametri, senza scambiare dati non necessari
- **di Dati-Strutтуrati**
  - I moduli passano intere strutture dati anche se una piccola parte di queste è necessaria

- **di Controllo**
  - Un modulo controlla il comportamento o la logica passandoli parametri o flag specifici
- **Esterno**
  - I moduli condividono un formato, un protocollo o un'interfaccia condivisa impostata esternamente
- **Comune**
  - I moduli dipendono da dati globali condivisi, creando un alto accoppiamento tra di loro
- **di Contenuto**
  - Un modulo accede o modifica il contenuto interno di un altro modulo

## Principi SOLID

### Princípio di singola responsabilità (SRP)

- Ogni modulo o microservizio dovrebbe avere una sola responsabilità o uno scopo
- Facilita la separazione di interessi, facilitando la testabilità e la manutenibilità
- Facilità la comprensione del sistema
- Le modifiche in un modulo non influenzano gli altri moduli
- Potrebbe richiedere una ristrutturazione di codice già esistente
- Potrebbe aumentare il tempo necessario per l'organizzazione prima di sviluppare

### Aperto-Chiuso (OCP)

- Le classi dovrebbero essere aperte all'estensione ma chiuse alla modifica
- Introduzione di nuove features senza intaccare funzionalità già presenti dando buona estendibilità
- Migliora la manutenibilità
- Necessita di buona progettazione
- Possibile complessità a livello implementativo

### Princípio di sostituzione di Liskov (LSP)

- Gli oggetti di una classe derivante dovrebbero poter sostituire gli oggetti della classe base senza alterare il comportamento del programma
- Serve per assicurarci la flessibilità e l'affidabilità, permettendoci di effettuare miglioramenti alle sottoclassi senza rompere le implementazioni esistenti
- Migliora la manutenibilità
- Richiede buona comprensione delle gerarchie di classe
- Possibile complessità a livello implementativo

### Princípio di segregazione dell'interfaccia (ISP)

- i consumer non dovrebbero essere costrette a dipendere da interfacce che non utilizzano
- Permette di avere un design più efficiente e pulito per le interfacce riducendo le dipendenze non necessarie

- Migliora la manutenibilità
- Potrebbe aumentare il numero di interfacce nel sistema
- Richiede una buona progettazione delle interfacce

## Principio dell'inversione delle dipendenze (DIP)

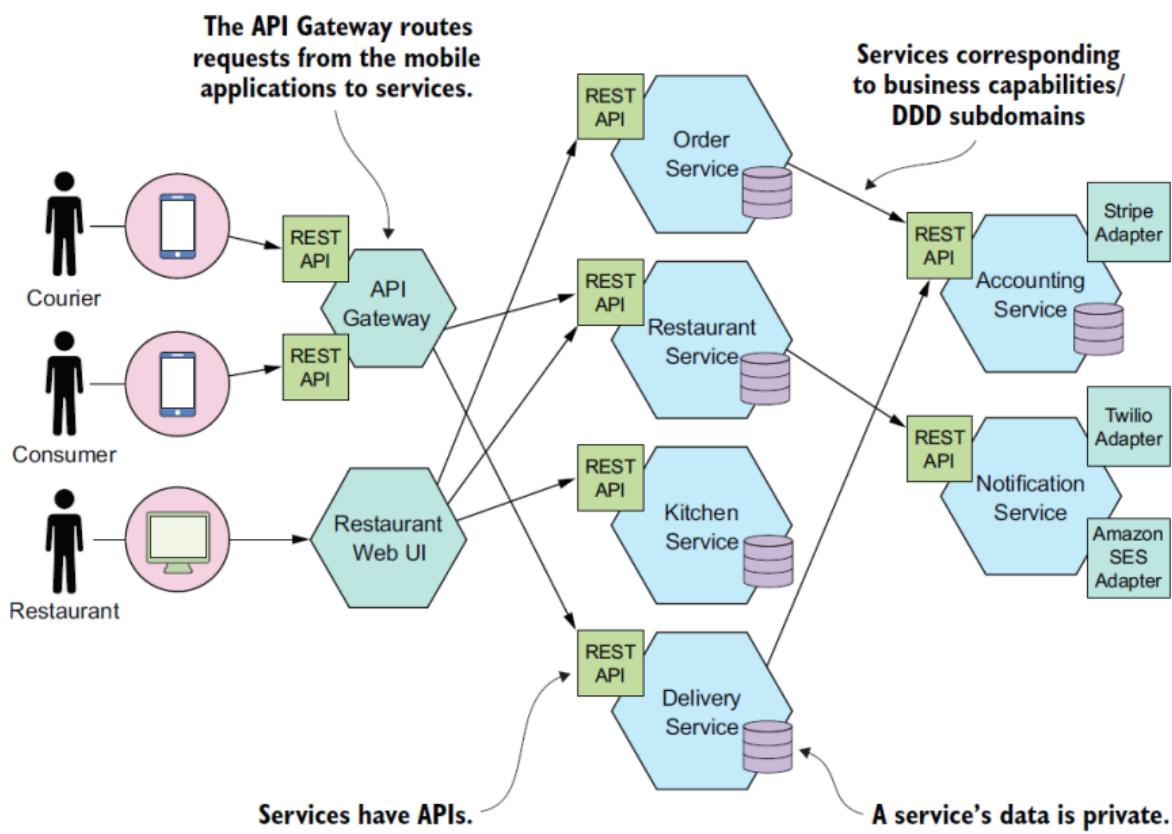
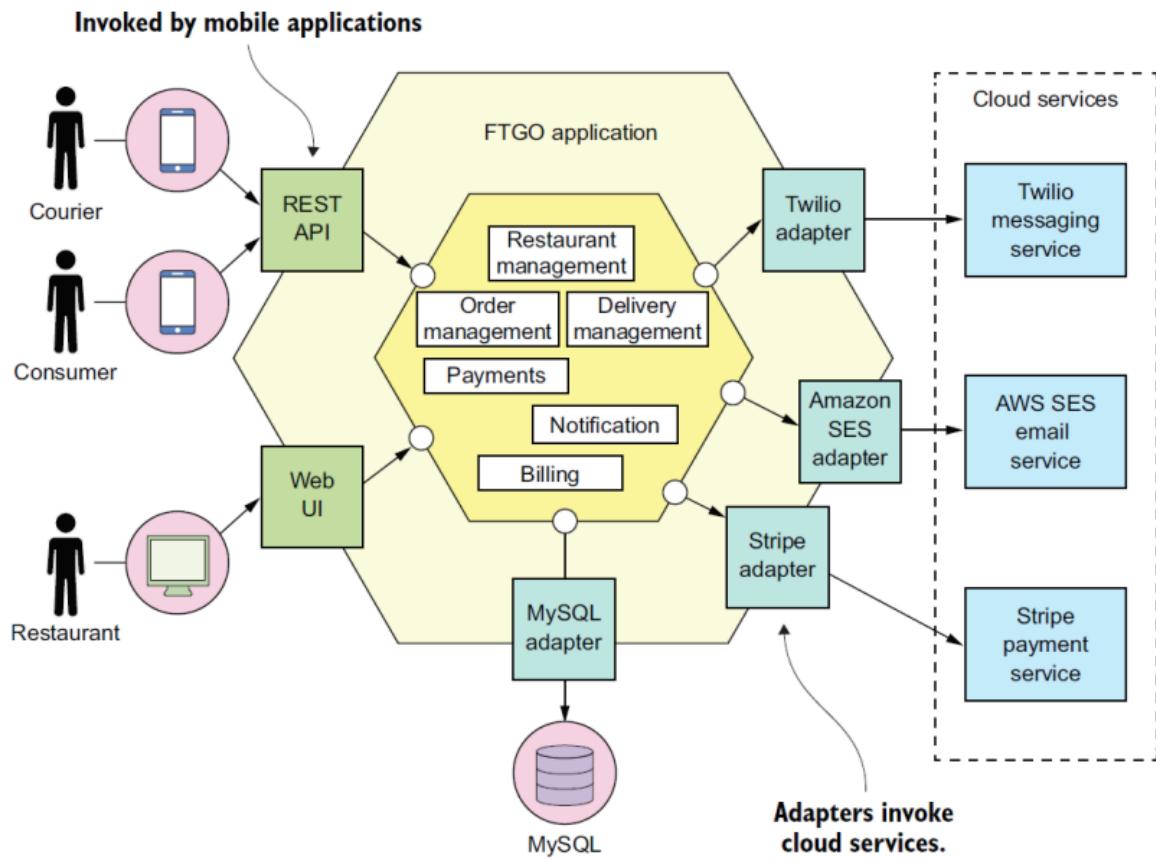
- I moduli di alto livello non dovrebbero dipendere da quelli di basso livello ma entrambi dovrebbero dipendere da astrazioni
- Permette di avere basso accoppiamento tra i moduli
- Migliora la manutenibilità e la flessibilità del sistema
- Richiede una buona comprensione delle astrazioni
- Possibile complessità a livello implementativo

# Architettura a microservizi

## Architettura esagonale

- Conosciuta anche come Ports and Adapters
- Basata sulla separazione tra Core (contenente la logica di business) e le interfacce esterne (tipo DB e servizi esterni)
- Abbiamo i seguenti attori:
  - Core
    - **Logica di business:** Regole e logiche che definiscono il comportamento dell'app
    - **Modelli di dominio:** Oggetti e entità principali del dominio applicativo
  - Porte
    - **Primarie:** Esterno → Core, anche chiamate porte di ingresso
    - **Secondarie:** Core → Esterno, anche chiamate porte di uscita
    -
  - Adattatori
    - **Primari:** implementano le porte primarie, consentono al core di interagire con i servizi esterni
    - **Secondari:** implementano le porte secondarie, consentono al core di utilizzare i servizi esterni

Nelle slide viene fatto l'esempio dell'app FTGO, inizialmente pensata come architettura monolitica e in seguito suddivisa in microservizi.



Per poter effettuare la migrazione è necessario:

- Identificare le operazioni che il sistema deve supportare (ogni operazione corrisponderà a un compito o un processo)
  - Es.: Possibili operazioni possono essere piazzare un ordine, aggiornarlo o processare un pagamento
- Mappare ogni operazione nei microservizi specifici
  - Es.: L'operazione per creare un ordine viene mappata all'interno del servizio per gli ordini
  - Es.: L'operazione per creare un ordine e quella per gestire il pagamento vengono gestite da servizi diversi per motivi legati alla scalabilità e all'isolamento
- Individuare le competenze di business e trasstrarle in microservizi appositi
  - Es.: Cominciare dividendo le responsabilità tra il microservizio che si occupa degli acquirenti e quello che si occupa dei venditori. Quest'ultimo a sua volta gestirà separatamente i ristoranti e i corrieri. Nello specifico si preferisce mantenere questi due in un unico servizio per rappresentare le loro funzionalità interconnesse e aumentare la coerenza di servizi, eventuali aggiornamenti sarebbero disponibili per entrambe le parti

## Delimitare i servizi con i Sotto-domini

### Introduzione

- Metodologia per creare software complessi con un modello di dominio orientato agli oggetti
- Definizioni utili sono quelle di:
  - Sottodomainio  
Area specifica all'interno del dominio che corrisponde a una competenza di business
  - Contesto limitato  
Ogni sottodomainio ha un modello di dominio distinto, riducendo la complessità

### Principi chiave

#### Princípio di singola responsabilità

Ogni servizio all'interno dell'architettura ha una responsabilità molto specifica e limitata per permettere al sistema di ridurre la complessità e aumentare la testabilità

Es.: Nel sistema FTGO dividiamo le responsabilità in:

- Servizio per gli ordini
- Servizio per la cucina
- Servizio per la spedizione

## Principio di chiusura comune

Le componenti che cambiano per le stesse ragioni dovrebbero essere “impacchettate” insieme. Questo serve per evitare di rideployare un gran numero di servizi dopo gli aggiornamenti.

Es.: Nel nostro caso il servizio per ordinare e per spedire vanno messi insieme

## Difficoltà implementative

### Latenza di rete

- Il fatto che ci siano dei “trip” potrebbe aumentare la latenza e peggiorare le performance
- In genere si cerca di minimizzare la comunicazione utilizzando le chiamate api in batch, ovvero facendo più operazioni in una singola richiesta HTTP

### Consistenza dei dati

- È difficile mantenere la consistenza dei dati tra tutti i servizi
- Si utilizza il pattern saga per mantenere la consistenza, cercando di coordinare le transazioni tra i servizi utilizzando dei messaggi asincroni

## Decomposizione di servizi

### Problema delle God-class

Per “god class” si intende una classe molto grande, monolitica, che ha troppe responsabilità, violando di fatto il principio di singola responsabilità (SRP). Per risolvere questa situazione è necessario strutturare le classi in modo da delimitare il loro dominio e pensare ogni servizio come un piccolo pezzo dell’intero sistema

Es.: Nel caso di FTGO la classe Order non sarà comune tra il servizio di spedizione e quello della cucina perché altrimenti questa dovrebbe avere un numero gigantesco di attributi, essendo sia specializzata in un modo che nell’altro. Andremo dunque a creare una classe per il servizio della spedizione (che avrà tutta la logica per le spedizioni, magari se è fragile o molto grande) e un’altra per la cucina (magari con una lista di ingredienti o cose così)

### Alto accoppiamento

Centralizzare la parte critica di un sistema richiede delle particolari attenzioni riguardo l’aggiornamento (relativamente agli accessi e le scritture), pertanto è necessario isolare gli accesi e le funzionalità tra i servizi e assicurare la possibilità per ogni servizio di aggiornarsi indipendentemente

## Best practices

### Decomposizione per Sottodomainio

- I microservizi saranno direttamente correlati con i sottodomini di business

- Es.: FTGO avrà 3 sottodomini, cioè Ordini, Cucina e Spedizione

## Decomposizione per SRP

- I microservizi devono rimanere piccoli e coesi, minimizzando le possibilità di cambiamenti
- Es.: In FTGO il servizio degli Ordini avrà a che fare solo con i servizi a lui affini, stesso per la Cucina e Spedizione

## Versioning semantico

Per poter descrivere la versione di un servizio a cui stiamo lavorando possiamo ricorrere a questa metodologia. Una versione è fatta in questo modo:

MAJOR.MINOR.PATCH

- MAJOR: si aumenta se ci sono cambiamenti incompatibili
- MINOR: si aumenta se ci sono nuove feature backward-compatible
- PATCH: si aumenta se ci sono fix backward-compatible

### Forward compatibility

Le versioni esistenti sono compatibili con quelle future, ignorando gli elementi “sconosciuti”

### Backward compatibility

Le nuove versioni sono compatibili con quelle esistenti

## Versioning delle API

### Tramite URI

La versione della API è direttamente inclusa nell'URL, un esempio è:

`https://api.example.com/v1/resource`

- La versione è chiaramente visibile nell'URL
- Facile sia da comprendere che implementare
- Ogni versione ha il suo URI separato

Tra i pro e i contro abbiamo:

- Facilità da parte degli utenti di capire la versione della API che stanno usando
- Coesistenza di diverse versioni
- Aumento di endpoint da gestire
- Richiesto aggiornamento manuale degli URL quando si cambia versione

### Tramite MIME-Type

La versione richiesta viene specificata tramite l'header HTTP Accept:

application/vnd.example.v1+json

- La versione è specificata nell'header della richiesta
- Gli URI rimangono puliti e non cambiano con le versioni
- Il server può negoziare il formato della risposta in base all'header Accept

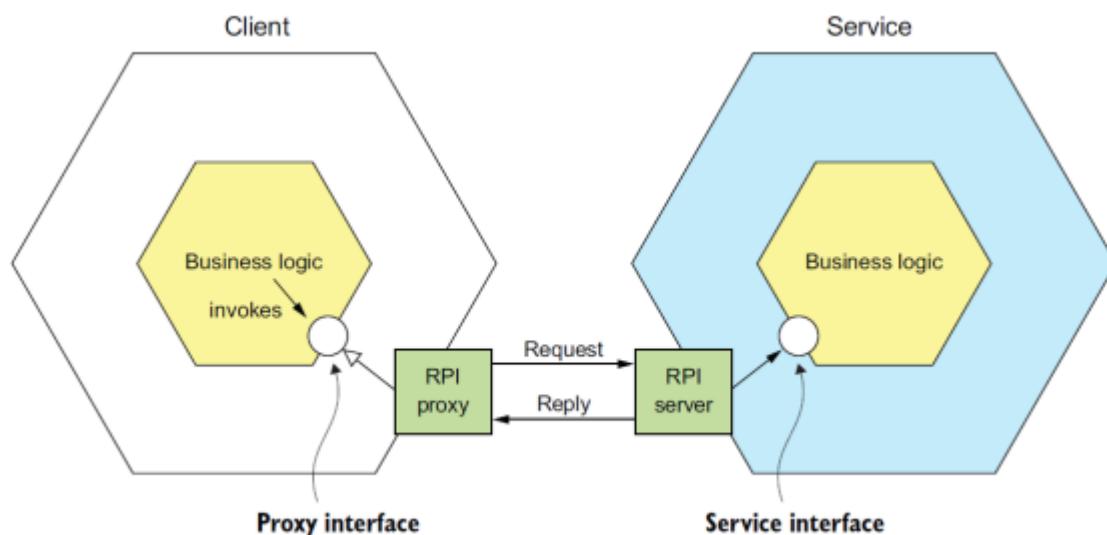
Tra i pro e i contro abbiamo:

- URL puliti e stabili
- Gestione di più versioni senza cambiare gli URL
- Permette di aggiungere nuove versioni senza cambiare l'URI
- Maggiore complessità lato server per interpretare gli header Accept
- Minore visibilità della versione

## Comunicazione tra microservizi

### Remote Procedure Invocation

Il client manda una richiesta a un servizio e ne aspetta la risposta



Concetti fondamentali di questa architettura sono:

- Client: invoca un'interfaccia Proxy implementata da una classe dell'adapter di RPI Proxy
- RPI Proxy: direziona la richiesta al servizio
- RPI Server Adapter: gestisce le richieste invocando la logica di business del servizio
- Flusso di risposta: Il servizio processa la richiesta e manda una risposta all'RPI Proxy che a sua volta manda il risultato al Client

### Comunicazione con REST

- Richieste e risposte sincrone quindi bloccanti
- Risorse identificate come URI e dati rappresentati con JSON o XML
- Si utilizzano i metodi HTTP (GET, POST, ...)
- Solitamente servono più richieste HTTP

- Il versioning deve essere gestito bene, alcuni cambiamenti alle API possono rompere tutto
- Necessarie delle strategie di Service Discovery

## OpenAPI

- Standard per la descrizione di API RESTful
- Si utilizzano JSON o YAML per la descrizione
- Possibile utilizzarlo con qualsiasi linguaggio
- Documentazione dettagliata di endpoint, metodi, parametri e risposte delle API
- Genera automaticamente i client stub e i server skeleton

## Swagger

- Fornisce un gran numero di strumenti per lavorare con le API
- Dà la possibilità di creare una documentazione interattiva

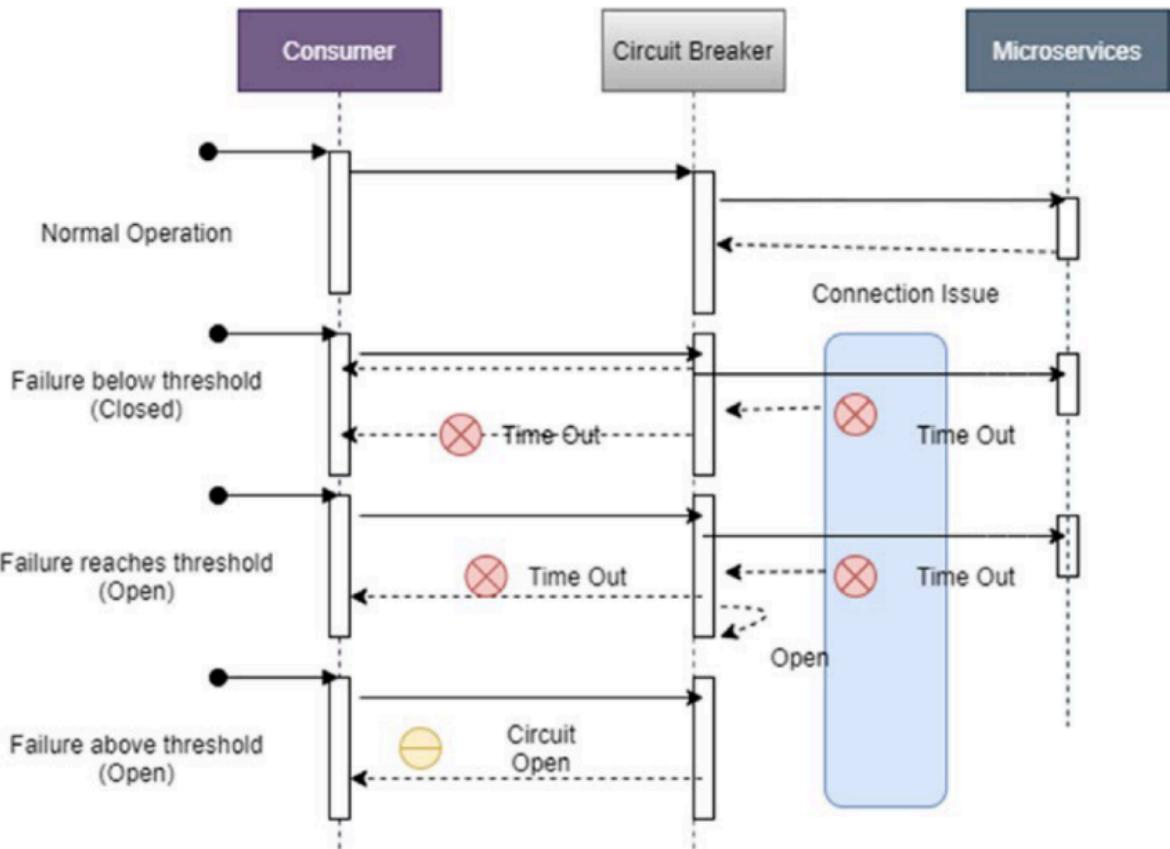
## Comunicazione con gRPC

gRPC è un framework per remote procedure calls che utilizza HTTP/2 per il trasporto e i protocol buffer come formato di serializzazione (mandare dati in binario è più efficiente che farlo con JSON)

- Utilizzo di versioning e tipizzazione forte grazie ai protocol buffer
- Supporto limitato a HTTP/2

## Circuit Breaker

Il circuit breaker è un meccanismo che si occupa di rifiutare le richieste che vengono fatte a un servizio dopo che questo ha superato una certa “soglia di fallimento”. Dopo un certo numero di fallimenti ripetuti il circuito si apre, evitando ulteriori tentativi. Dopo un certo timeout il client può riprovare con la richiesta, se questa va a buon fine il circuito si chiude.



### Sviluppo di RPI Proxy robusti

- Necessità di stabilire timeout di rete (per evitare che delle risorse siano vincolate senza un limite di tempo)
- Limitare il numero massimo di richieste in sospeso per evitare sovraccarichi
- Implementare il Circuit Breaker Pattern per tenere sotto controllo situazioni critiche

### Strategie di ripristino

- Bisogna far ritornare gli errori in caso di errori nella comunicazione
- Restituire delle risposte “di ripiego” nel caso in cui un servizio essenziale non sia disponibile

### Bulkhead Pattern

- Strategia di design per isolare le componenti in compartimenti stagni (come le paratie delle navi, evitavano di farle affondare in caso di allagamento)
- Ogni microservizio ha il proprio pool di risorse ed è totalmente separato dagli altri
- In caso di fallimento di un servizio le capacità di risposta degli altri servizi rimane invariata
- Protegge tutto il sistema dai fallimenti in cascata in caso di traffico elevato o interruzioni di servizio

### Retry Pattern

- Strategia di design che permette la gestione di guasti momentanei mediante il retry delle operazioni fallite

- Non si utilizza per non errori momentanei (ad esempio quelli di autenticazione) ma per errori momentanei (ad esempio packet loss)
- Viene utilizzato un meccanismo di Exponential Backoff, ovvero l'aumento esponenziale del tempo che intercorre tra un tentativo e quello successivo
- È buona pratica implementare il circuit breaker per evitare i retry in caso di disservizi non momentanei

## Service Discovery

Il Service Discovery è un meccanismo che ha l'obiettivo di individuare e connettere dinamicamente i microservizi. È necessario in quanto con i microservizi si lavora utilizzando il cloud e i container pertanto gli indirizzi IP potrebbero cambiare spesso.

## Service Registry

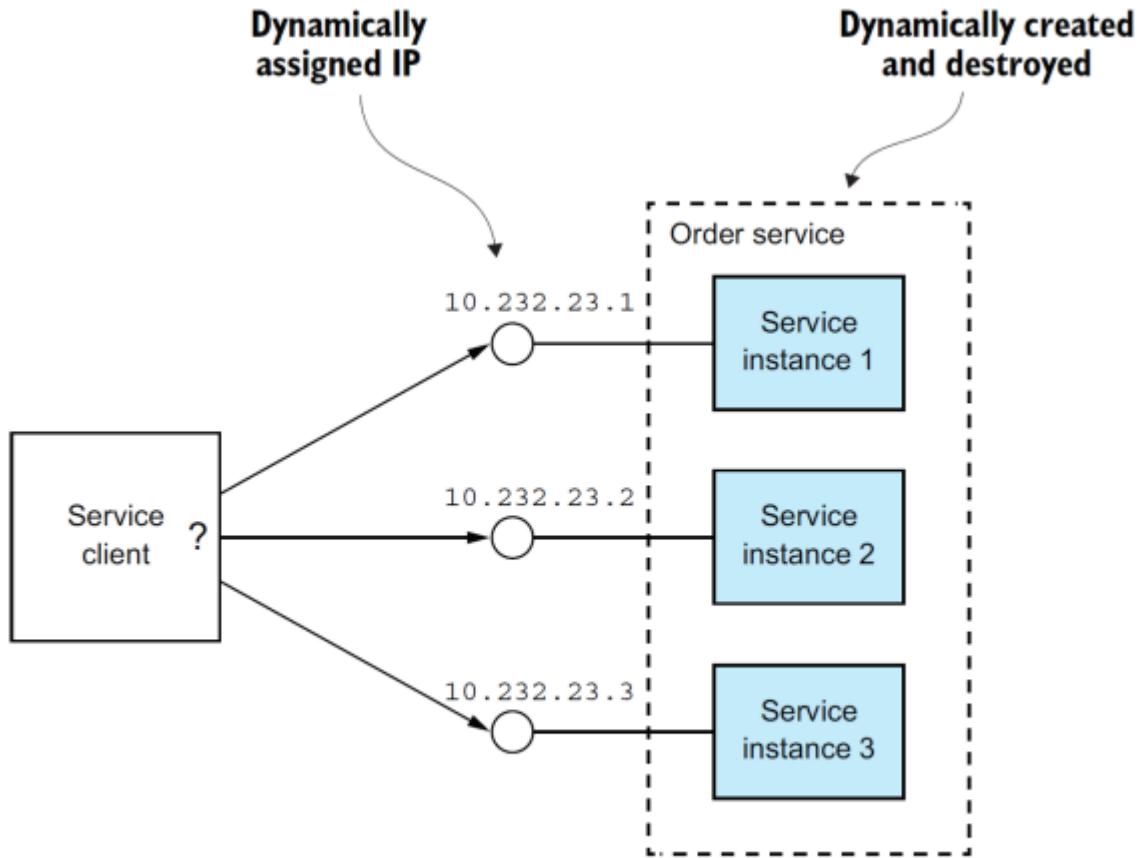
Database che contiene le informazioni sugli indirizzi di rete relativi alle istanze di tutti i servizi. Ogni volta che un servizio viene avviato (o riavviato) vengono salvati i dati relativi a indirizzo IP e porta

## Dynamic Updates

Questa è una capacità chiave del Service Registry, in quanto gli permette di rispecchiare i cambiamenti che avvengono ai microservizi, sapendo se qualcuno di questi viene avviato, arrestato o fallisce

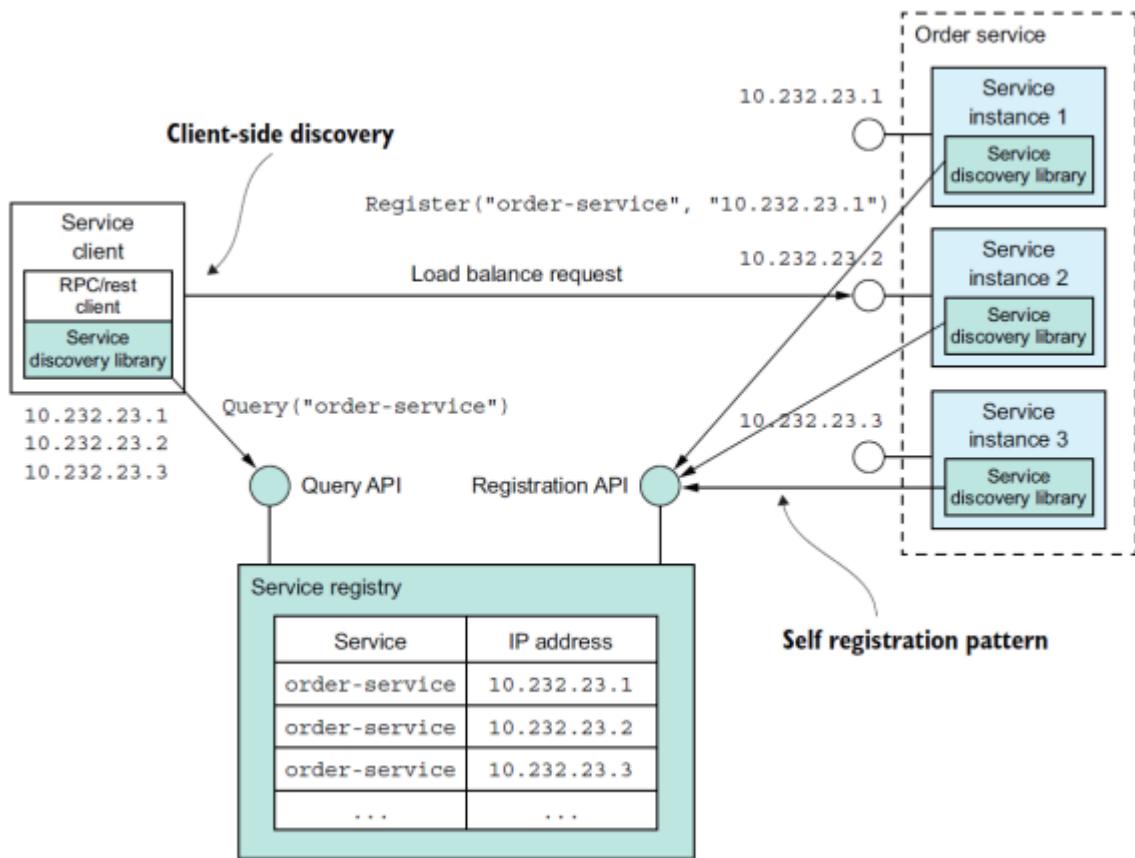
## Client Role

Ogni client, ovvero ogni microservizio, qualora dovesse avere bisogno di comunicare con qualche altro microservizio interroga il Service Registry e ottiene gli indirizzi delle relative istanze. Appena ottengono questi dati possono provvedere a mandare le richieste



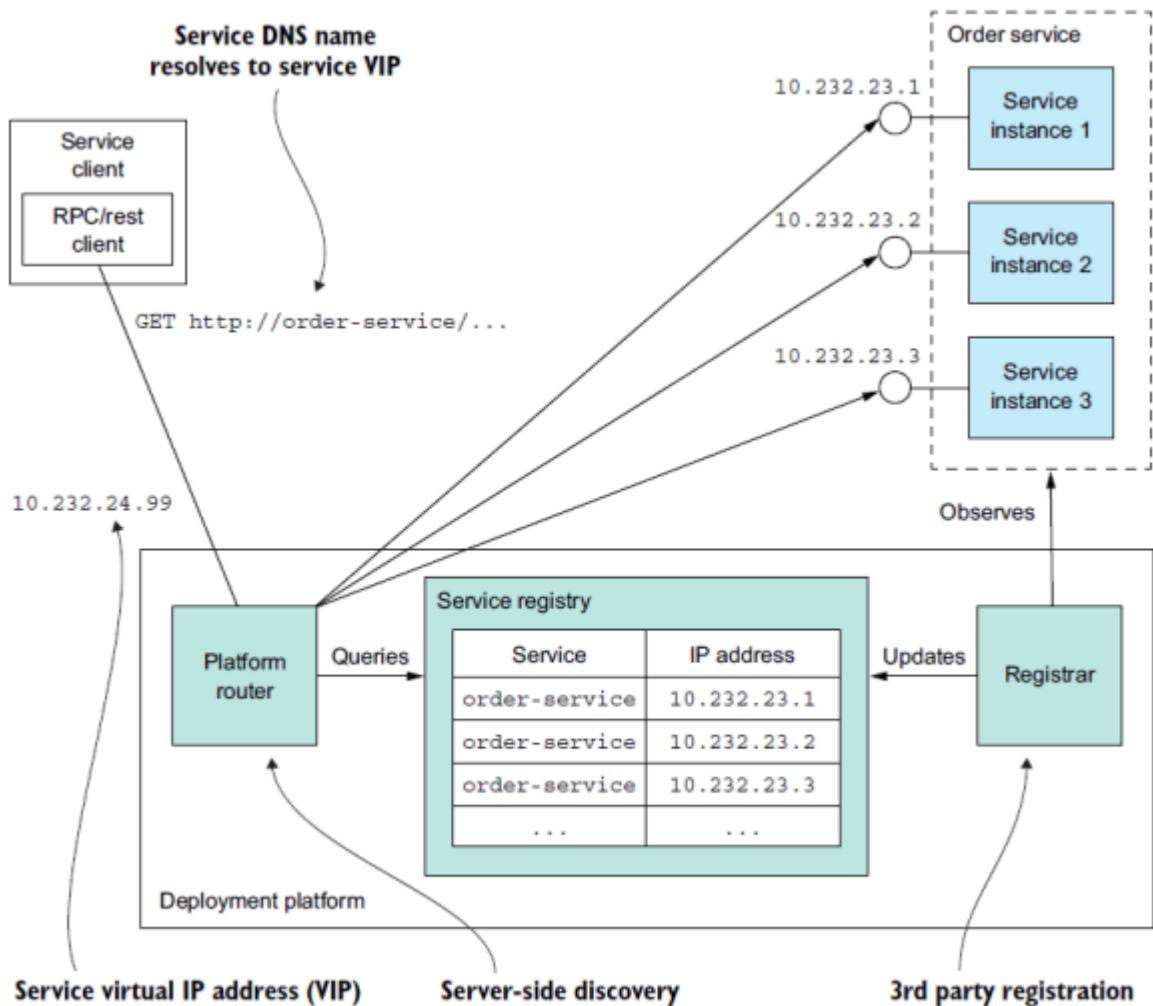
### Application-Level Service Discovery Patterns

- Ogni istanza dei servizi di regista al Service Registry con una chiamata API fornendo indirizzo IP e porta (**Self Registration**)
- È anche possibile implementare un health-check URL per verificare lo stato di disponibilità dei servizi
- Prima di fare una richiesta il client provvede a richiedere al Service Registry una lista delle istanze dei servizi disponibili (**Client-Side Discovery**)



## Platform-Provided Service Discovery Patterns

- Presenza di un componente dedicato per gestire le registrazioni all'interno del Service Registry, spesso parte della piattaforma di deployment (**3rd Party Registration**)
- Riduce significativamente l'onere dei servizi individuali relativamente all'aggiornamento del loro stato
- I client fanno la richiesta al componente apposito (o a un indirizzo DNS o a un IP virtuale) e questo si occupa di interrogare il Service Registry e bilanciare le richieste (**Service-side Discovery**)



## Asynchronous Messaging

### Messaggi

Il messaggio è l'unità di comunicazione all'interno di un sistema di messaggistica asincrona, è composto da:

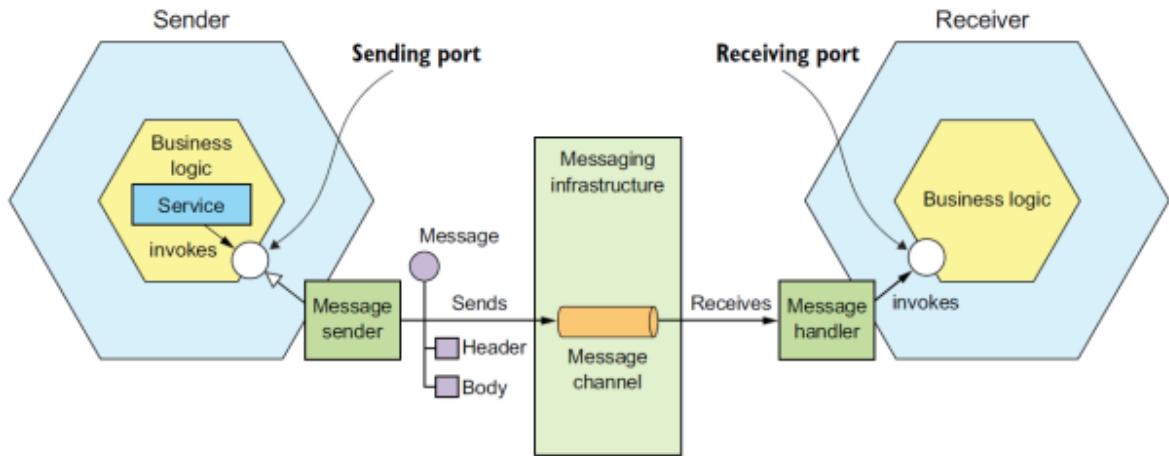
- Header: contiene i metadati del messaggio (ID, indirizzo di ritorno e altre informazioni utili alla gestione o al routing)
- Body: contiene i dati effettivi del messaggio (possono essere in formato testo o binario)

Esistono diverse tipologie di messaggi, ognuno con uno scopo diverso:

- Document: contiene solo dati che il ricevente deve interpretare (è come un documento che viene inviato tra un sistema e un altro)
- Command: È equivalente a una richiesta RPC, specifica un'operazione che il ricevente deve eseguire
- Event: indica un evento, spesso rappresenta un cambiamento di stato

## Canali Punto-Punto

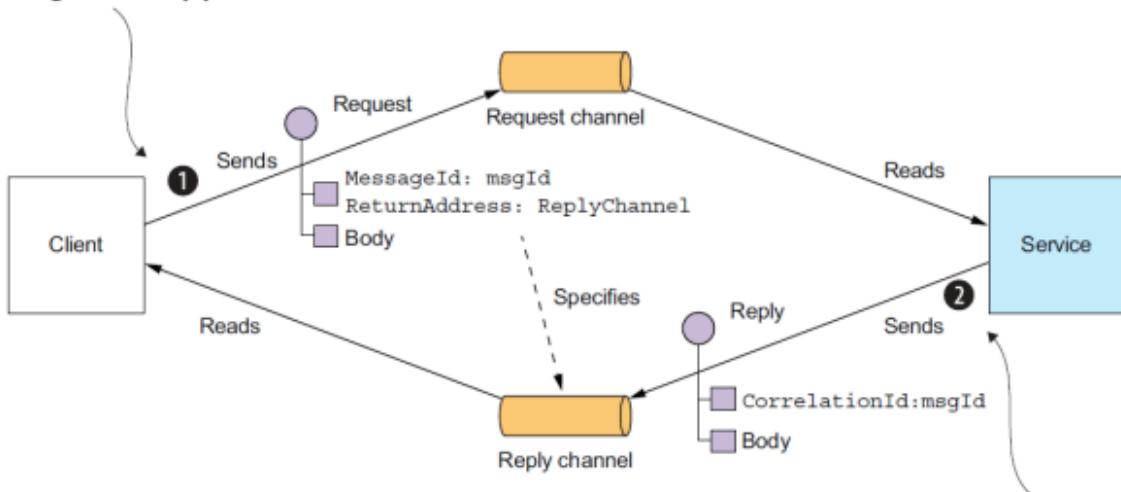
Possono essere sincroni o asincroni, nel primo caso il client si blocca aspettando una risposta (caso non molto comune)



Nel caso asincrono, invece:

- Il client manda un messaggio contenente un messageId e un indirizzo di ritorno. Il primo è un ID specifico che il client assegna per capire a quale richiesta farà capo la risposta da parte del servizio, il secondo si riferisce al canale che deve utilizzare il servizio per fornire la risposta
- Una volta che il servizio riceve ed elabora la richiesta la inoltra indietro secondo il canale di risposta specificato dal client. La risposta avrà anche un campo CorrelationID che conterrà l'ID del messaggio

**Client sends message containing messageId and a reply channel.**



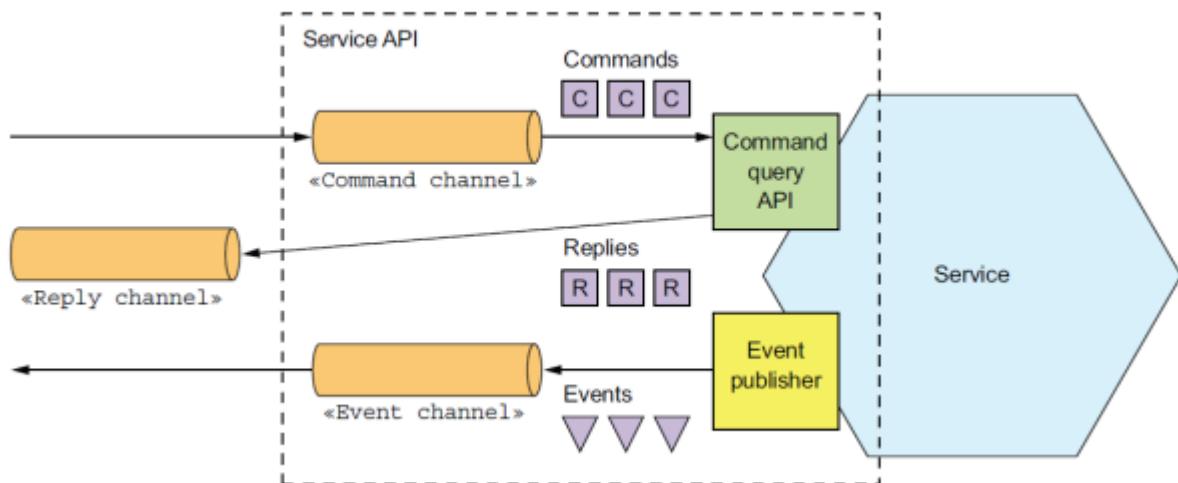
**Service sends reply to the specified reply channel. The reply contains a correlationId, which is the request's messageId.**

In questa tipologia di comunicazione è possibile avere delle One-Way Notification, ovvero comunicare senza aspettarsi delle risposte (questo è il caso in cui si comunica un cambiamento di stato)

## Canali Publish-Subscribe

Il produttore pubblica dei messaggi su un canale e più consumatori possono leggerlo. Questo caso è particolarmente utile per pubblicare degli eventi di dominio (Es.: in FTGO possiamo pubblicare gli ordini o gli stati delle consegne).

È possibile anche ricevere delle risposte asincrone, per farlo è necessario che il client specifici un canale su cui ricevere la risposta



## Brokerless Messaging

I servizi scambiano messaggi senza l'utilizzo di un intermediario

- Bassa latenza
- Traffico più semplice da gestire e minore congestione di rete
- Necessità di implementazione di Service Discovery
- Sia chi invia che chi riceve deve essere attivo per comunicare

## Broker-based Messaging

Tra i servizi opera un broker che fa da intermediario, occupandosi di mandare i messaggi

- Basso accoppiamento, i servizi non devono sapere la posizione degli altri servizi
- Presenza di un buffer di messaggi
- Supporta sia req/res che pub/sub
- Il broker potrebbe essere motivo di bottleneck se non scalato correttamente
- Il broker è un point of failure, se c'è un problema crolla tutto

## AMQP

### Componenti chiave

#### Broker

Il broker si occupa di ricevere i messaggi e instradarli ai consumatori. Tra questi abbiamo RabbitMQ

## Exchange

Gli Exchange ricevono messaggi dai produttori e li instradano alle code basandosi sulle regole di binding

## Queue

Le code memorizzano i messaggi fino a quando non vengono letti dai consumatori

## Binding

I binding definiscono le regole che collegano gli exchange alle code

## Message

I messaggi sono composti da un header (che contiene una serie di metadati) e un body (i dati effettivi)

## Routing Key

Stringa utilizzata da AMQP per instradare i messaggi dagli exchange alle code, utile quando si vuole instradare seguendo dei pattern o stabilendo precisamente il binding

## Flusso

### 1 - Produzione (Producer → Exchange)

Un produttore invia un messaggio a un exchange

### 2 - Instradamento (Exchange → Queue)

L'exchange riceve il messaggio e lo instrada a una o più code in base alle regole di binding

### 3 - Consumo (Queue → Consumer)

Il consumatore riceve il messaggio dalla coda e lo rielabora

## Tipi di Exchange

### Direct Exchange

I messaggi vengono instradati direttamente alle code con una routing key esatta

### Fanout Exchange

I messaggi vengono instradati a tutte le code collegate non considerando la routing key

### Topic Exchange

I messaggi vengono instradati alle routing key in base ad alcuni pattern di routing key

Es.: una routing key come `*.log` instraderà il messaggio a `error.log`, `info.log`

Es.: `order.*` instraderà a `order.new`, `order.update`

## Headers Exchange

I messaggi vengono instradati in base ai valori degli header piuttosto che le routing key

## Messaggistica Punto-Punto

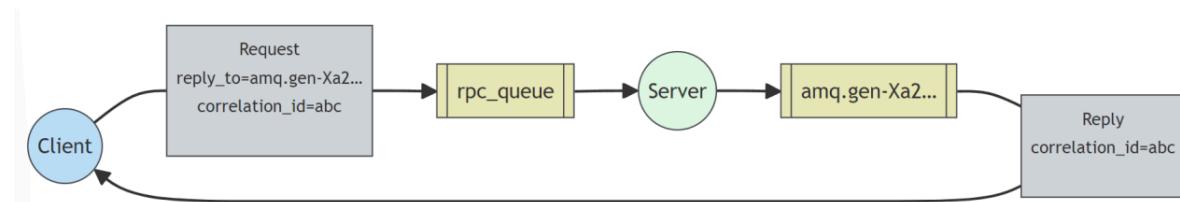
È un pattern dove i messaggi vengono mandati a una coda specifica e un consumer riceve e processa il messaggio. La comunicazione avviene in questo modo:

- Il producer manda un messaggio con una routing key
- Il messaggio viene conservato nella coda
- Il consumatore “consuma” il messaggio nella coda

In AMQP, il meccanismo built-in per il direct exchange si utilizza il nome della coda come routing key (**Default Exchange**)

## Implementazione di RPC su RabbitMQ

- L’interazione comincia con il Client che manda un messaggio di richiesta a una coda specifica (**rpc\_queue**) e ne aspetta il risultato
- Il Client stabilisce una specifica coda di callback e ne riceve le risposte, questa viene indicata nella proprietà **reply\_to** della richiesta
- Ogni richiesta ha come proprietà un **correlation\_id** che servirà al client per capire a quale richiesta corrisponde una specifica risposta appena ricevuta



## AMQP con Publish/Subscribe

In questo caso si utilizza il protocollo AMQP per mandare messaggi da un produttore a una serie di consumatori in modo simultaneo.

È particolarmente utilizzato con Fanout Exchange (distribuisce i messaggi a tutti a prescindere dalle routing key) o Topic Exchange (distribuisce basandosi sulla routing key).

Utilizzato in vari scenari:

- Necessità di logging, ovvero i servizi aspettano dei messaggi di log
- Architetture event-driven, dove i moduli ricevono degli aggiornamenti
- Sistemi quali dashboard real-time o notifiche

## AMQP Routing su RabbitMQ

Il routing permette di mandare messaggi a uno o più destinatari in base ai criteri stabiliti.

Questo fa sì che un consumatore riceva solo i messaggi a cui è interessato e non tutti.

- Binding: relazione tra un Exchange e una coda, indica che la coda è interessata ai messaggi che riceve dall’exchange
- Binding key: parametro opzionale che specifica le condizioni sotto le quali i messaggi vengono direzionati a una coda collegata (*bound*)

## Direct Exchange

I direct Exchange instradano i messaggi alle code la cui binding key corrisponde esattamente alla routing key del messaggio

## Multiple Bindings

Più code possono essere legate con la stessa binding key. In questo caso il messaggio con questa routing key sarà consegnata a tutte le code che soddisfano questa condizione

## Topic Exchange

I Topic Exchange permettono la redirezione di messaggi a una serie di code utilizzando le wildcard.

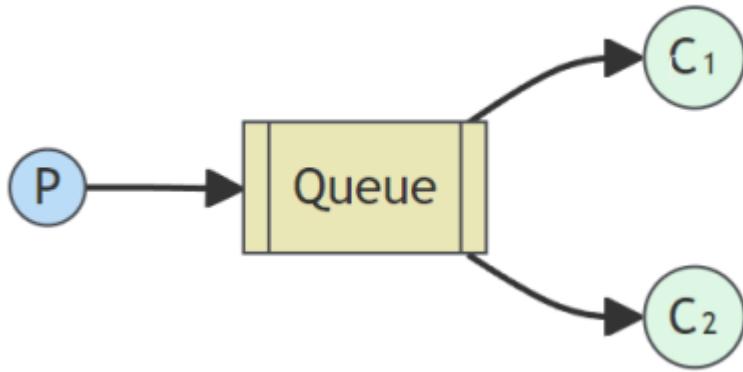
Es.: `stock.usd.nyse`, `quick.orange.rabbit`

- Wildcard \*: Corrisponde esattamente a una parola.
- Wildcard #: Corrisponde a zero o più parole.

Utilizzando `*.orange.*` andremo a selezionare tutti gli animali arancioni (Es.: `quick.orange.rabbit`, `lazy.orange.elephant`).

## Esempio di coda di lavoro

- **Produttore:** Responsabile dell'invio di **compiti** (tasks) a una coda. Questi rappresentano dei **lavori** che **richiedono tempo** (Es.: elaborazione immagini o analisi di dati)
- **Consumatore:** **Lavoratori** che si occupano di **prelevare** dei dati dalla coda. Per ogni coda possono esserci più consumatori, questo permette un **throughput** più elevato nel sistema
- **Equa distribuzione:** i **compiti** vengono **distribuiti equamente** tra i consumatori, facendo sì che non ci siano sovraccarichi e ognuno di questi abbia un numero simile di richieste da elaborare
- **Durabilità:** I messaggi e la coda possono **sopravvivere** a degli eventuali crash del server (**persistenza della coda e dei messaggi**)
- **Manual Acknowledgment:** Meccanismo che assicura che nessun compito venga perso se un lavoratore fallisce prima di completarlo. È infatti necessario **riconoscere** esplicitamente il messaggio dopo l'elaborazione altrimenti sarà inserito nuovamente nella coda



Pertanto, approfondendo ulteriormente potremo dividere in:

Logica del producer:

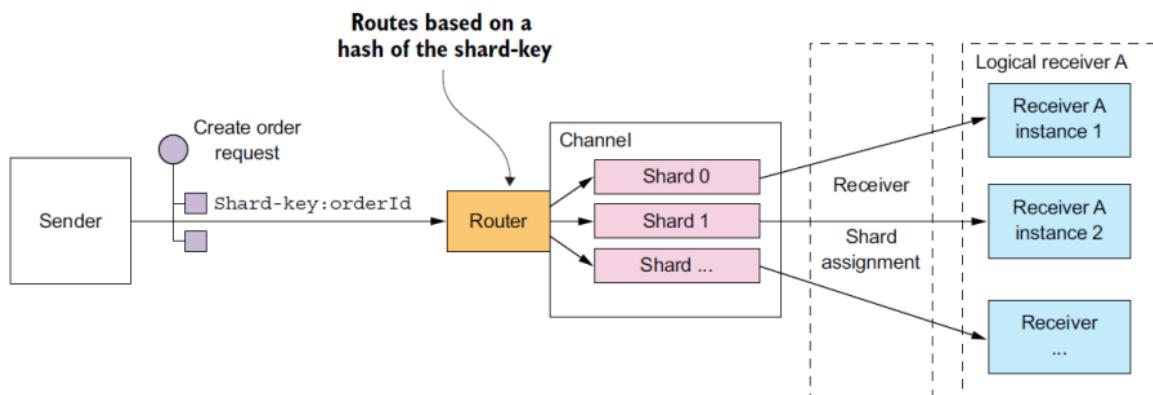
- 1) Invia un nuovo ordine a una coda
- 2) Passa i dettagli dell'ordine tramite linea di comando e li rende persistenti

Logica del consumer:

- 1) Il consumer consuma gli ordini della coda e li elabora
- 2) Tempo di elaborazione
- 3) Manual acknowledgment per assicurarsi che il messaggio una volta elaborato venga tolto dalla coda

Canali partizionati (Sharded)

Se per aumentare il throughput utilizziamo più istanze del servizio potrebbe capitare che l'ordine di elaborazione dei messaggi non sia rispettato, pertanto si utilizza un canale "shardato" (sharded), dove i messaggi vengono suddivisi tra partizioni basate su una chiave di shard (in generale un ID, come quello del cliente o dell'ordine), per permettere l'elaborazione indipendente di ogni partizione, pur mantenendo un ordine per i messaggi



Quindi:

- 1) Ogni messaggio viene associato a una chiave di shard (per capire a quale partizione appartiene il messaggio)
- 2) I messaggi vengono distribuiti tra diverse partizioni basate sulla chiave di shard (ogni partizione è una coda separata)
- 3) Ogni partizione viene elaborata da un set specifico di istanze del servizio (e questo fa sì che i messaggi vengano elaborati nell'ordine corretto)

## Problemi di duplicazione

In un sistema di messaggistica è possibile che un messaggio sia consegnato più volte (che sia per guasti o per crash). Questo crea particolari problemi se gli eventi sono sequenziali (si fa l'esempio di un messaggio per eliminare un ordine esattamente dopo uno per creare un ordine)

In questo caso esistono due soluzioni:

- Handler idempotenti: più invocazioni della stessa operazione non creano effetti aggiuntivi (l'ordine già annullato non subisce effetti se viene rieseguita l'operazione per annullare)
- Tracking di messaggi: implementazione di un meccanismo che si occupi di memorizzare e controllare gli ID dei messaggi per non elaborare lo stesso messaggio più di una volta

## Comunicazione asincrona

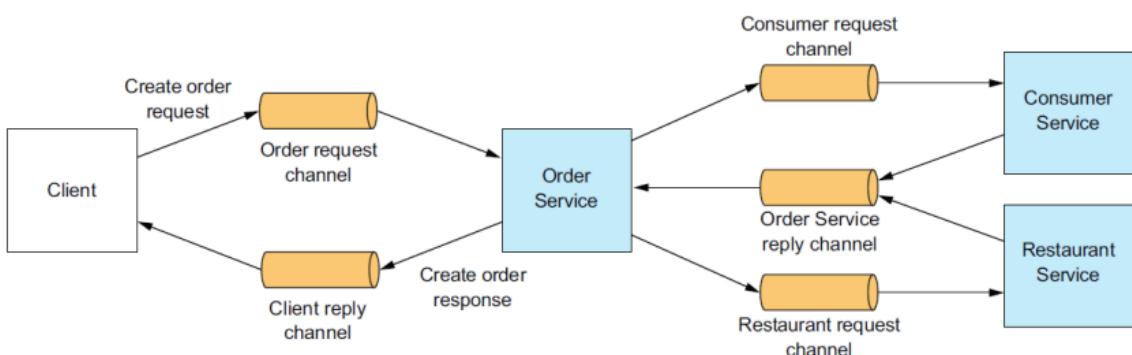
La comunicazione sincrona tra servizi ne richiede la disponibilità in tempo reale. Questo può essere estremamente problematico, soprattutto perché la disponibilità totale si calcola come il prodotto delle disponibilità di tutti i servizi. Se abbiamo un buon numero di servizi questa tende a calare drasticamente

Es.: Avere 10 servizi, ognuno con 99,5% di disponibilità ci farà avere una disponibilità totale del 95,11%

La soluzione è creare una comunicazione asincrona e la replicazione dei dati per cercare di minimizzare le richieste sincrone.

## Interazione asincrona

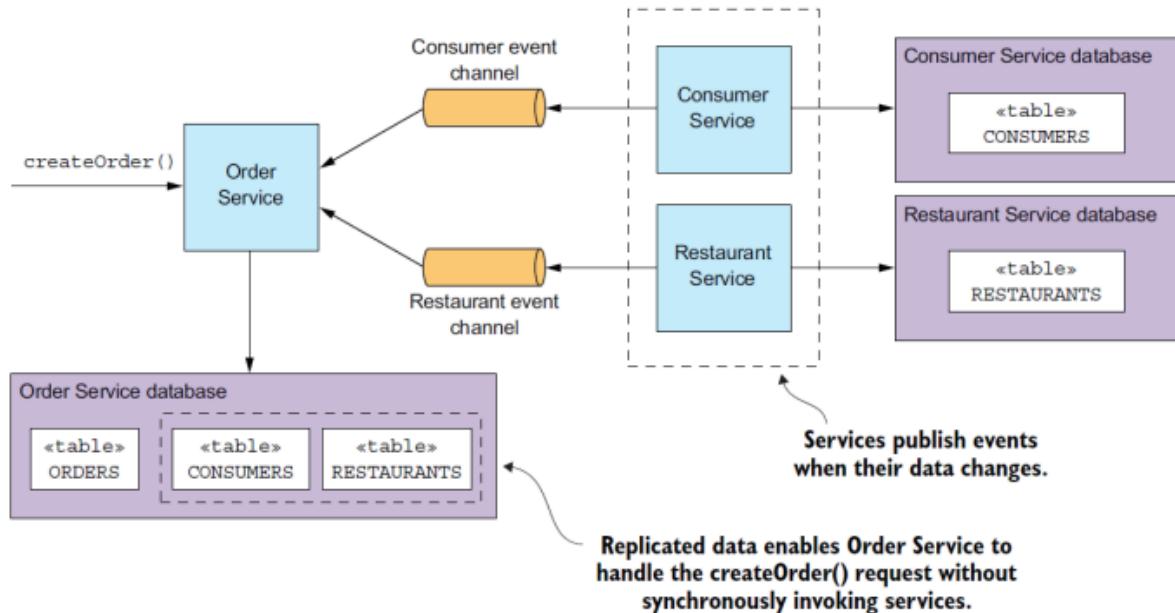
- I servizi non sono bloccati ad aspettare una risposta
- Il broker conserva i messaggi, si assicura che i servizi rimangano resilienti
- Le API esterne utilizzano protocolli sincroni (come le REST) quindi c'è bisogno di una risposta veloce



## Data Replication

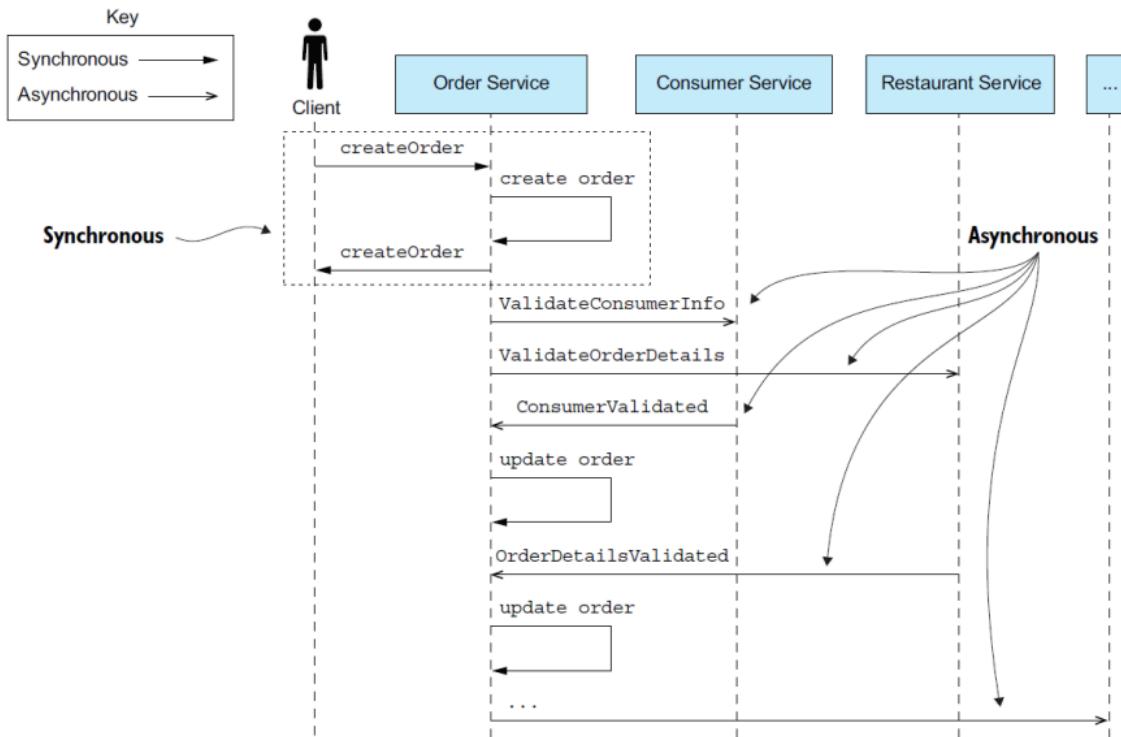
- 1) Si valida la richiesta usando solo dati disponibili localmente
- 2) Si aggiorna il database aggiungendo i dati anche alla tabella OUTBOX (che contiene i messaggi che devono essere mandati agli altri servizi)

- 3) Viene mandata immediatamente una risposta al client, senza aspettare altre elaborazioni



Analizziamo il flusso:

- 1) Order Service crea un ordine in stato PENDING inviando anche dei messaggi di validazione al Consumer Service e al Restaurant Service
- 2) Consumer Service valida il consumatore e invia una risposta ConsumerValidated  
Restaurant Service valida i dettagli dell'ordine e invia una risposta OrderDetailsValidated
- 3) Dopo che Order Service ha ricevuto le risposte aggiorna l'ordine assegnandogli lo stato VALIDATED

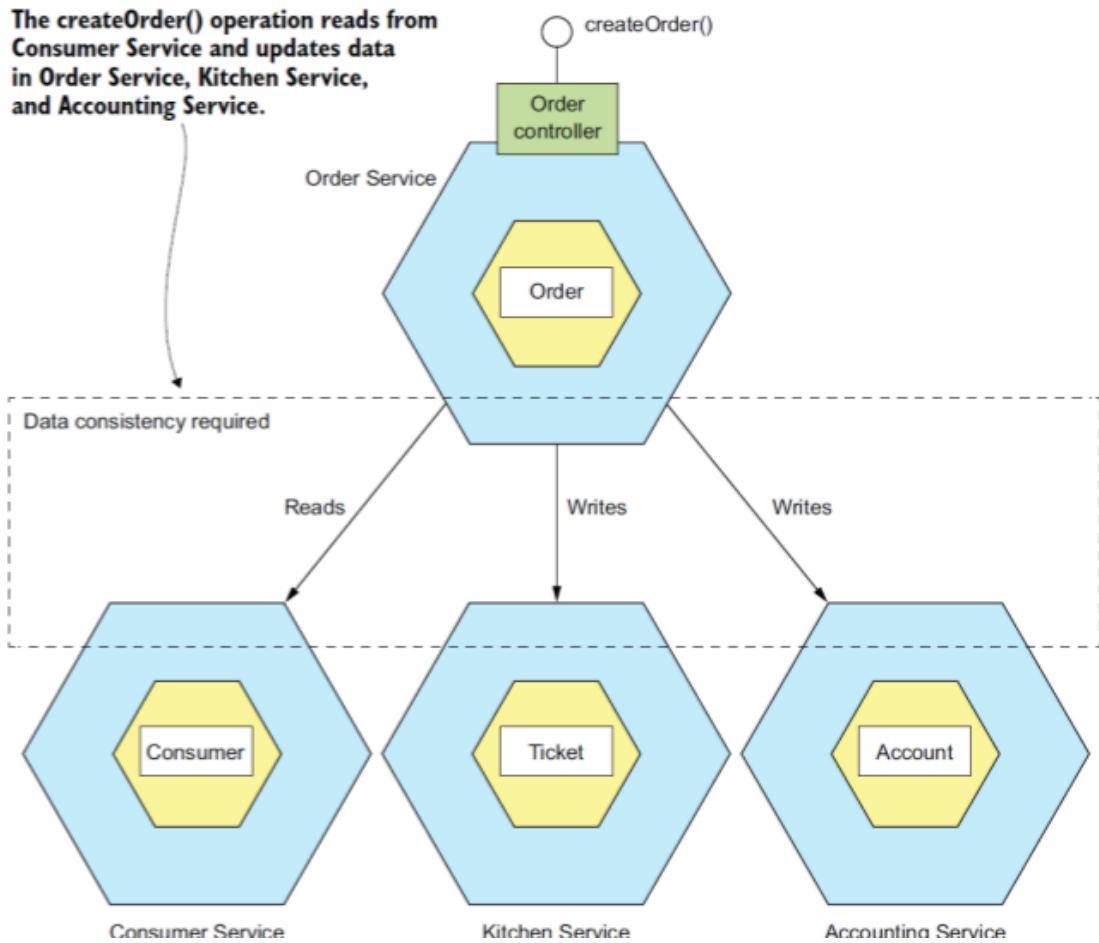


## Coordinazione e orchestrazione

- I dati necessari per le nostre operazioni sono divisi tra più servizi
- È possibile avere solo 2 requisiti tra Consistenza, Disponibilità e Tolleranza alle partizioni
- In generale si preferisce optare per la Disponibilità per motivazioni legate ai cali di disponibilità (che si fanno sentire particolarmente con tanti servizi)
- Il requisito che si sacrifica è la Consistenza

## Sistemi di transazioni

- I dati sono “divisi” tra i diversi servizi
- Ogni servizio ha il suo database
- È presente un meccanismo per assicurare la consistenza globale
- Si mantengono le ACID globalmente grazie all'utilizzo di transazioni distribuite



Com'è possibile vedere ogni servizio necessita di **aggiornare il database** (quindi creare o aggiornare entità) e **pubblicare messaggi** (eventi di dominio). Tuttavia se un servizio dovesse **crashare** dopo l'aggiornamento del database ma prima dell'invio del messaggio, il sistema finirebbe in uno **stato inconsistente**

Però:

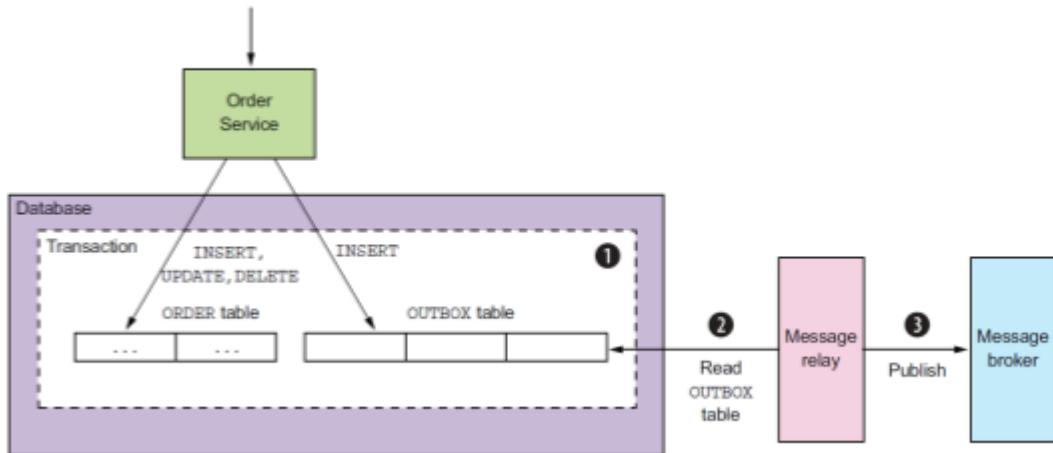
- Le transazioni distribuite non sono adatte ai microservizi
- I moderni message broker non supportano le transazioni distribuite
- Si utilizzano altre soluzioni per assicurarsi che il database e i messaggi siano sincronizzati

## Outbox Pattern

In questo pattern utilizziamo:

- Outbox Table: **Database** dedicato che funge da **coda temporanea** di messaggi, ogni operazione che lavora con dei dati di business passa di qui
- Message Relay: Componente che **legge i messaggi** dalla Outbox Table e li **invia** al broker

Questo ci permette di garantire atomicità alle operazioni



## Polling Publisher Pattern

Il message Relay interroga periodicamente la Outbox Table per far pubblicare messaggi al broker, una volta inoltrati li elimina.

- Il polling è inefficiente, soprattutto se il sistema è particolarmente grande

Il tutto funziona in questo modo:

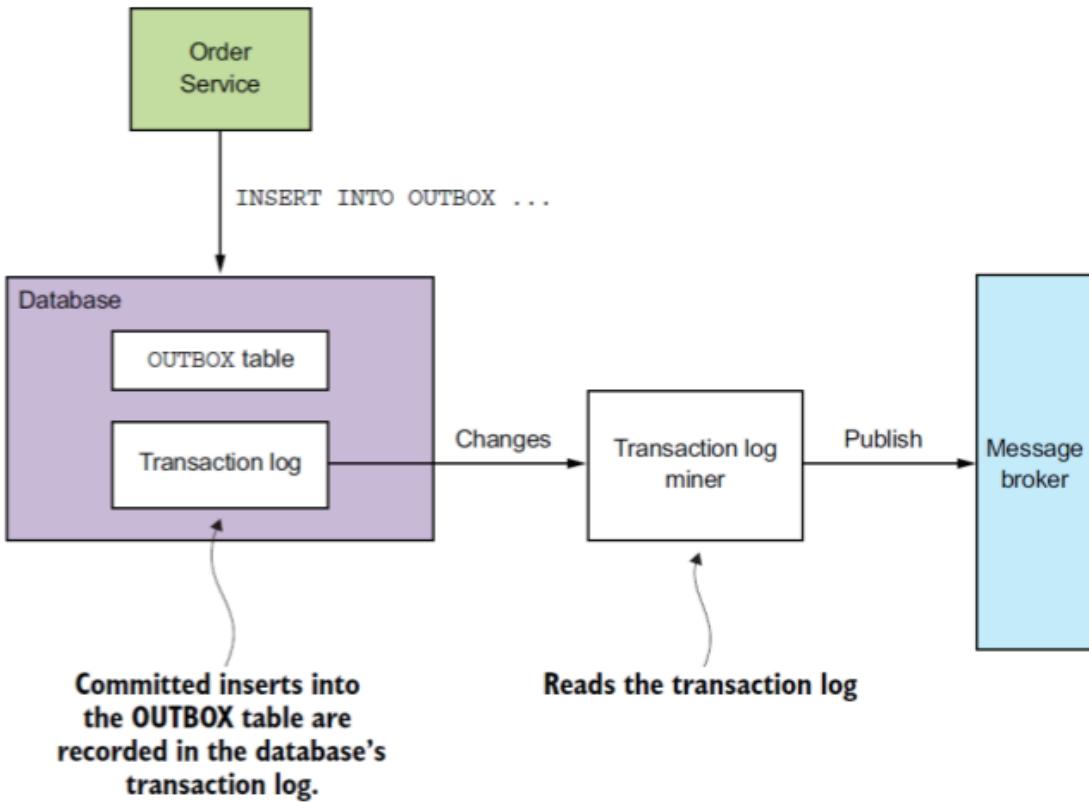
- 1) Il message Relay fa una **query** alla Outbox Table, chiedendo che sia **ordinata in modo ascendente**
- 2) **Inoltra i messaggi** al broker (che li pubblica ai canali appositi)
- 3) **Cancella i messaggi** dalla Outbox Table dopo che sono stati **pubblicati**

## Transaction Log Tailing Pattern

Un'idea simile è quella di fare polling non su tutto un database di transazioni ma su un database di log di transazioni.

In questo caso quindi, c'è un componente che si occupa di catturare i cambiamenti dal database dei log e inoltrare i messaggi al broker.

- Scala meglio del polling (soprattutto in sistemi con traffico elevato)
- Particolari difficoltà implementative



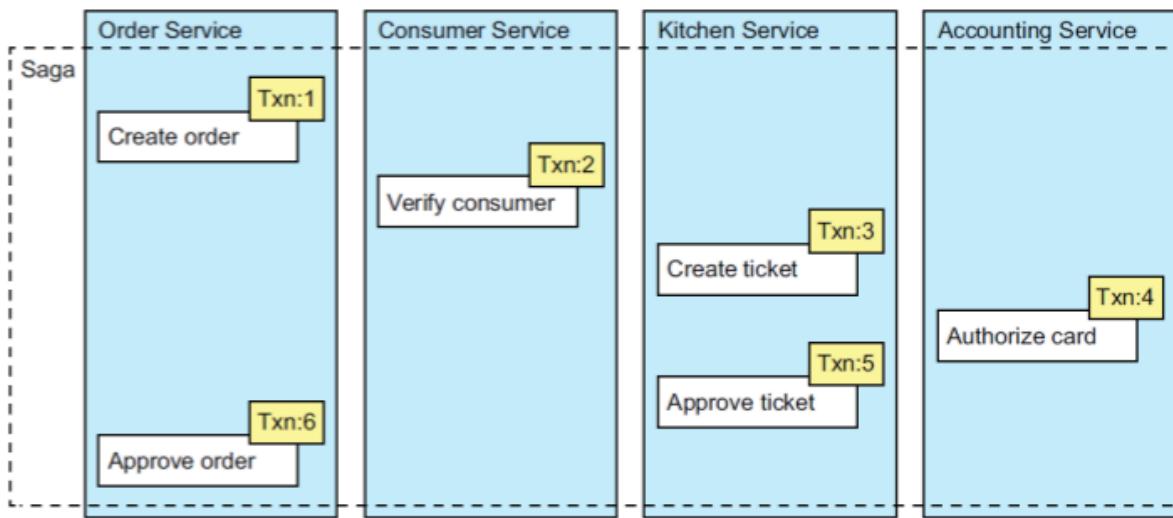
## SAGA Pattern

Design pattern per gestire transazioni tra microservizi basato sull'idea di basso accoppiamento e servizi asincroni per garantire la consistenza senza usare le transazioni distribuite. Utilizza:

- **Transazioni locali** (per ogni servizio)
- **Comunicazione asincrona** (con eventi o comandi)
- **Transazioni di compensazione** (in caso di fallimento, per garantire la consistenza)

## Esempio di implementazione

- 1) **Order Service** crea un ordine in stato `APPROVAL_PENDING`
- 2) **Consumer Service** verifica il consumer
- 3) **Kitchen Service** valida i dettagli dell'ordine e crea un ticket in stato `CREATE_PENDING`
- 4) **Accounting Service** autorizza la carta di credito
- 5) Kitchen Service aggiorna lo stato del ticket in `AWAITING_ACCEPTANCE`
- 6) **Order Service** aggiorna lo stato in `APPROVED`



## Transazioni di compensazione

Diversamente dai sistemi tradizionali, utilizzando la saga non è possibile utilizzare rollback, questo perché:

- Ogni servizio effettua delle transazioni locali (su database locali)
- Ogni modifica viene immessa direttamente nel database del servizio

Per questo motivo, determinati servizi devono provvedere a implementare delle operazioni per far tornare il sistema a uno stato consistente.

Vediamo cosa succederebbe se ci fosse un fallimento nell'esempio precedente:

- 1) **Order Service** crea un ordine in stato **APPROVAL\_PENDING**
- 2) **Consumer Service** verifica il consumer
- 3) **Kitchen Service** valida i dettagli dell'ordine e crea un ticket in stato **CREATE\_PENDING**
- 4) **Accounting Service** rifiuta la carta di credito
- 5) Kitchen Service aggiorna lo stato del ticket in **CREATE\_REJECTED**
- 6) **Order Service** aggiorna lo stato in **REJECTED**

Tutto questo è possibile perché il nostro sistema implementa dei metodi che permettono al sistema di tornare indietro, cioè

Step	Service	Transaction	Compensating Transaction
1	Order Service	createOrder()	rejectOrder()
2	Consumer Service	verifyConsumerDetails()	—
3	Kitchen Service	createTicket()	rejectTicket()
4	Accounting Service	authorizeCreditCard()	—
5	Kitchen Service	approveTicket()	—
6	Order Service	approveOrder()	—

Possono esserci 3 tipi di transazioni, ovvero:

- **Transazioni compensabili**: possono essere annullate utilizzando le transazioni di compensazione
- **Transazioni Pivot**: sono il punto di non ritorno, se hanno successo la saga sarà completato
- **Transazioni retrabili**: dopo la transazione pivot andranno sempre a buon fine

## Coordinazione

La coordinazione è un meccanismo fondamentale quando si tratta di gestire il flusso delle transazioni locali. Esistono due modi per gestire la coordinazione:

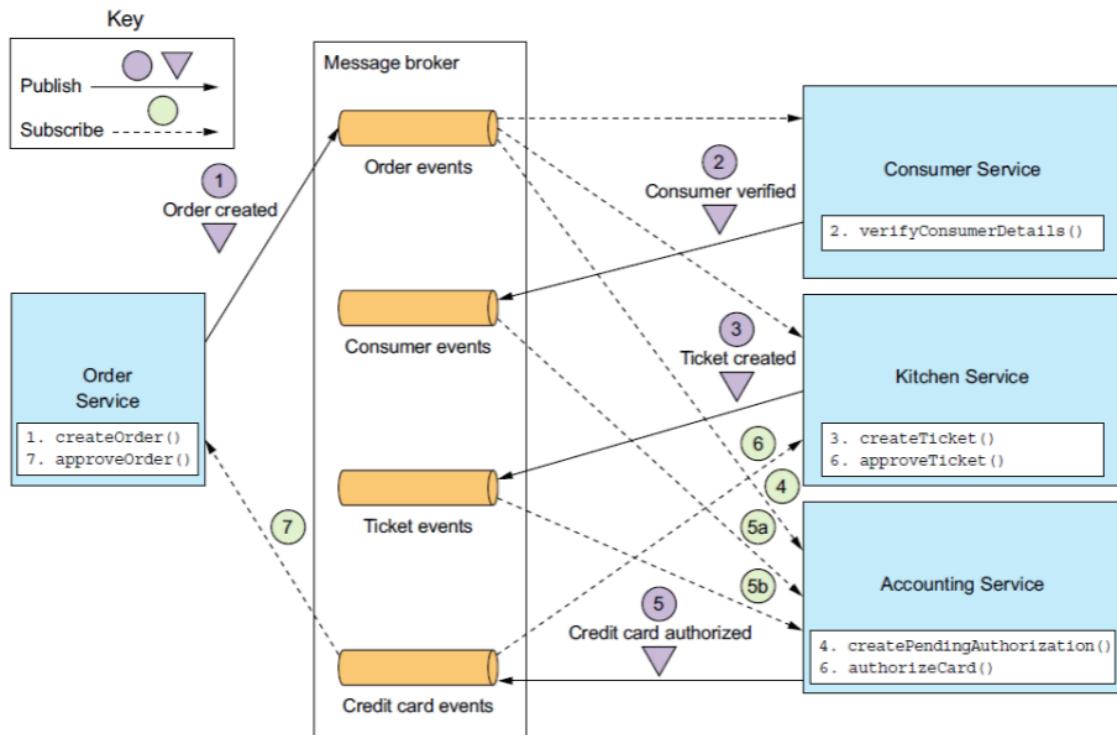
- **Coreografia**: controllo **decentralizzato**, i servizi si attivano tra di loro con gli eventi, ognuno di questi ascolta gli eventi e fa qualcosa a riguardo
- **Orchestrazione**: controllo **centralizzato** con un **orchestratore** che si occupa di gestire il flusso inviando messaggi in sequenza ai servizi

## Coreografia

- In ogni passaggio vengono aggiornati i database in modo atomico
- Viene utilizzato l'ID di correlazione tra tutti i servizi
- Permette di avere un **accoppiamento basso**, i servizi partecipanti si sottoscrivono solo ai topic che gli interessano e non sanno nulla degli altri
- Il sistema mantiene una certa **semplicità** in quanto si caricano gli eventi quando vengono creati o aggiornati gli oggetti
- I partecipanti devono sottoscriversi **SOLO** ai topic utili (potrebbe **aumentare l'accoppiamento**)
- **Difficoltà di comprensione** e possibilità di **dipendenze cicliche**

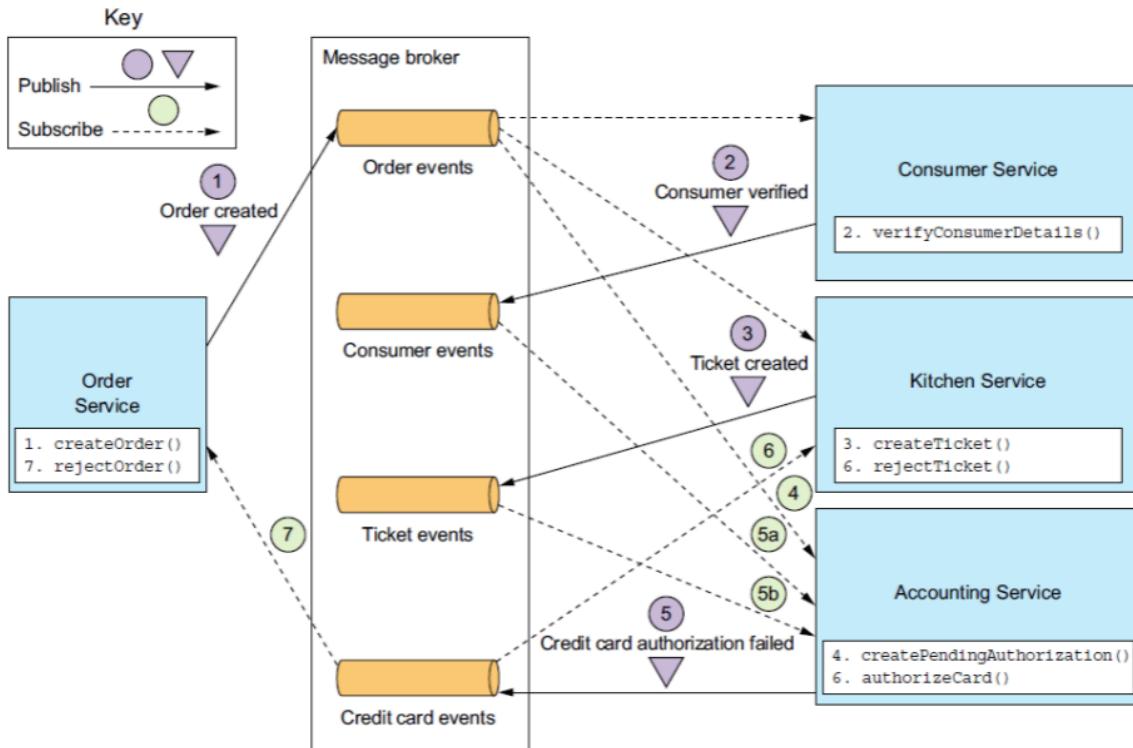
## Caso di successo

1. **Order Service**: Crea un ordine (**APPROVAL\_PENDING**) e pubblica **OrderCreated()**.
2. **Consumer Service**: Consuma **OrderCreated()**, verifica e pubblica **ConsumerVerified**.
3. **Kitchen Service**: Consuma **OrderCreated()**, valida l'ordine, crea un ticket (**CREATE\_PENDING**) e pubblica **TicketCreated()**.
4. **Accounting Service**: Consuma **OrderCreated()**, crea un'autorizzazione della carta (**PENDING**).
5. **Accounting Service**: Consuma **TicketCreated()** e **ConsumerVerified()**, autorizza la carta di credito e pubblica **CreditCardAuthorized()**.



## Caso di fallimento

1. **Order Service**: Crea un ordine (`APPROVAL_PENDING`) e pubblica `OrderCreated()`.
2. **Consumer Service**: Verifica il consumatore e pubblica `ConsumerVerified()`.
3. **Kitchen Service**: Crea un ticket (`CREATE_PENDING`) e pubblica `TicketCreated()`.
4. **Accounting Service**: Consuma `OrderCreated()`, crea un'autorizzazione della carta di credito (`PENDING`).
5. **Accounting Service**: Tenta di autorizzare la carta di credito, fallisce e pubblica `CreditCardAuthorizationFailed()`.



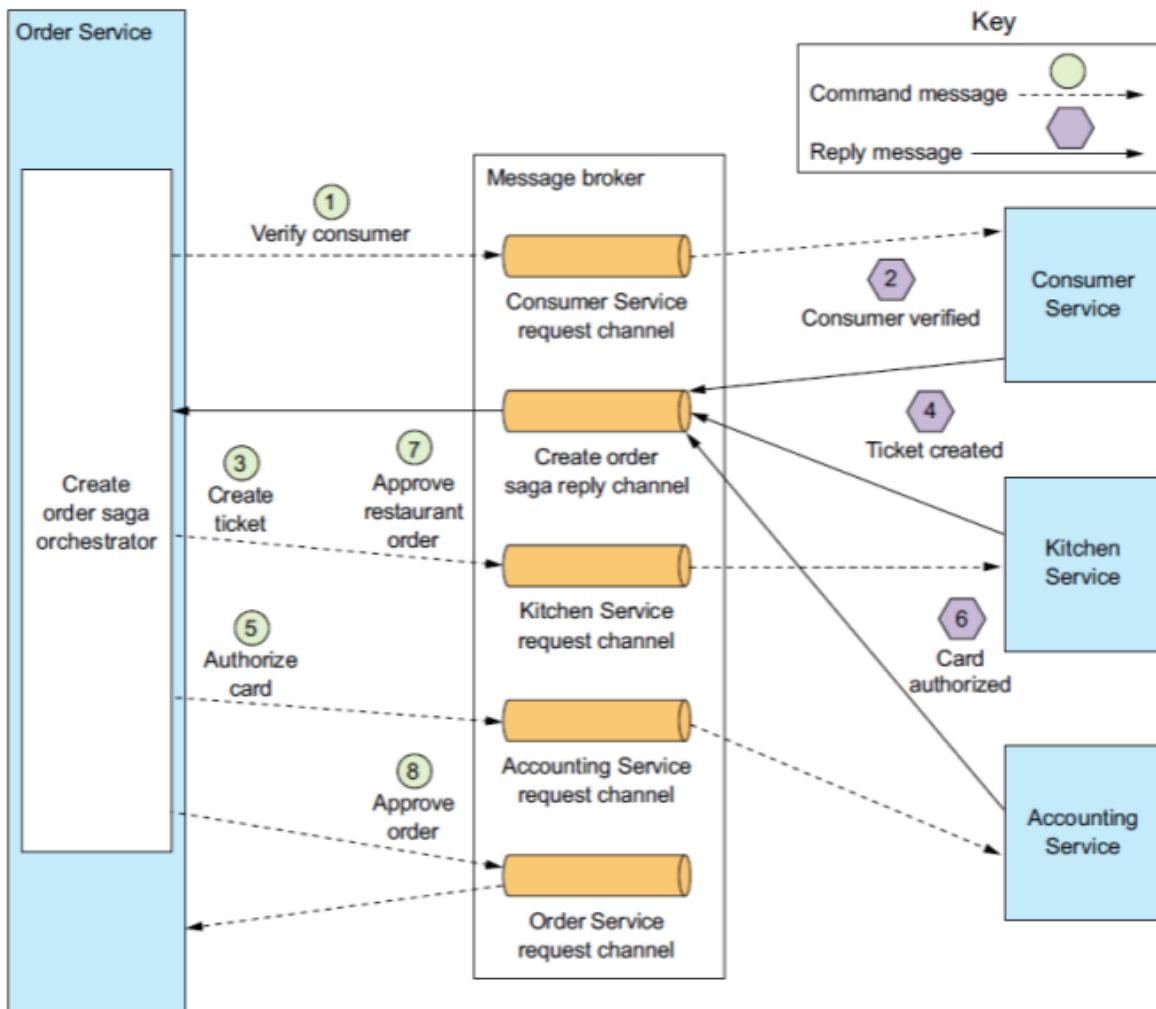
## Orchestrazione

Tipologia di coordinazione con un attore che si occupa di gestire tutto il flusso (orchestratore).

- **Dipendenze più semplici**, si evitano le dipendenze cicliche
- **Accoppiamento più basso**, i servizi espongono solo le API e non conoscono gli eventi tra di loro
- **Separazione netta delle “preoccupazioni”**
- **Rischio di centralizzazione**, l'orchestratore potrebbe diventare molto complesso
- L'orchestratore si cerca di utilizzarlo **SOLO** per gestire la **sequenza**

## Caso di successo

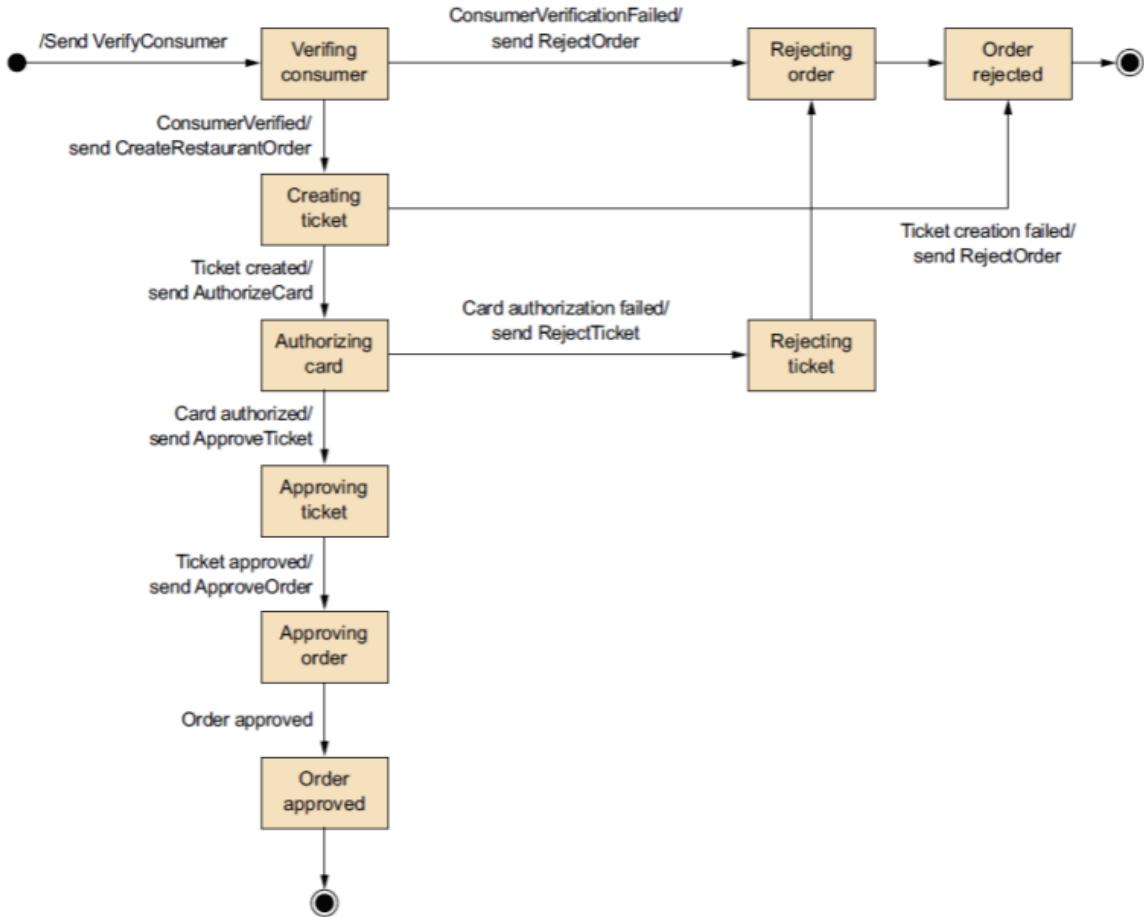
- 1) **Saga Orchestrator** invia il comando **VerifyConsumer()** al Consumer Service.
- 2) **Consumer Service** risponde con un messaggio **ConsumerVerified()**.
- 3) **Saga Orchestrator** invia il comando **CreateTicket()** al Kitchen Service.
- 4) **Kitchen Service** risponde con un messaggio **TicketCreated()**.
- 5) **Saga Orchestrator** invia il comando **AuthorizeCard()** al Accounting Service.
- 6) **Accounting Service** risponde con un messaggio **CardAuthorized()**.
- 7) **Saga Orchestrator** invia il comando **ApproveTicket()** al Kitchen Service.



Orchestratore e Macchina a stati

È possibile vedere un orchestratore SAGA come una macchina a stati finiti in quanto composta da stati e transizioni. Ogni transizione viene attivata dal completamento di una transazione locale. Questo ci permette di avere una visione chiara di tutti i possibili scenari (anche quelli in caso di fallimento).

Qui di seguito un esempio di una macchina a stati:



## Mancanza di Isolamento

Utilizzando la saga pattern il nostro sistema **NON** rispetterà le **proprietà ACID** ma solo **ACD**, quindi non sarà rispettato **l'isolamento**. Questo perché gli altri saga possono accedere ai dati durante l'esecuzione, causando anomalie

## Anomalie nel SAGA

### Lost updates

Una SAGA **sovrascrive** le **modifiche** apportate da un'altra SAGA **prima di leggerle**.

Es.: Create Order approva un ordine che era stato cancellato da Cancel Order

### Dirty reads

Una SAGA **legge dati** da un altro SAGA che **non** ha **ancora** completato i suoi **aggiornamenti**.

Es.: Create Order legge la disponibilità di credito dopo che Cancel Order l'ha incrementata, permettendo un ordine che superi il credito effettivo qualora la Cancel Order viene annullata

Fuzzy/Nonrepeatable reads

In diversi passaggi di una SAGA si leggono gli **stessi dati** ma si ricevono **valori diversi** perché un'altra SAGA ne ha aggiornato i dati.

Es.: Check Inventory legge i livelli di stock in momenti diversi; nel caso in cui lo stock viene modificato da un'altra saga tra una lettura e l'altra i dati potrebbero essere incoerenti

## Contromisure per la mancanza di isolamento

Lock semantico

È possibile implementare questa soluzione utilizzando delle transazioni compensabili, che impostano un flag (come `*_PENDING`) nel record. Questo serve per segnalare che il record non è completamente confermato, agendo di fatto da blocco. Questo flag viene poi rimosso quando la SAGA viene terminata (sia positivamente che negativamente)

Es.: Nella Create Order Saga possiamo utilizzare lo stato `APPROVAL_PENDING` per bloccare l'ordine. Una volta completata potremo cambiare lo stato in `APPROVED` (o se negativamente, in `REJECTED`)

Si può decidere di optare per due metodi:

- **Fallire e richiedere un nuovo tentativo:** semplice ma sposta la complessità al client
- **Bloccare fino al rilascio del blocco:** fornisce isolamento ma richiede la gestione dei deadlock

Aggiornamenti commutativi

**Operazioni** che possono essere eseguite in **qualsiasi ordine** senza influenzare il risultato finale

Es.: In un sistema per gestire dei conti, le operazioni di credito e debito sono commutative se possono essere usate in qualsiasi ordine senza conflitti

- **Permette di annullare** in sicurezza le azioni delle saghe **senza rischiare sovrascritture**, previene **lost updates**

Vista pessimistica

**Riordinamento** dei **passaggi** di una saga per assicurare una **sequenza logica** delle operazioni. Riduce anche il rischio di **dirty reads**

Es.: Nella Create Order SAGA si aggiorna lo stato dell'ordine a `CANCELLED` prima di notificare gli altri servizi per evitare problemi con le transazioni concorrenti (che leggerebbero dati incoerenti)

- **Garantisce** che i passaggi seguano una **sequenza** che **minimizza** i rischi aziendali derivanti da problemi di **concorrenza**

Rilettura del valore

Si **ricontrolla** lo **stato** di un record prima di apportare delle modifiche

Es.: Nella Create Order SAGA il sistema può verificare se l'ordine è ancora APPROVAL\_PENDING prima di approvarlo, assicurandosi che nessun'altra SAGA l'abbia modificato

- Utile per **mantenere la coerenza confermando** l'integrità dei dati prima di applicargli degli aggiornamenti

File di versione

Utilizzo di un **file** che **mantiene** un **record** di ogni **aggiornamento**, permettendo di **riordinare** le **operazioni** per **correggere** degli eventi di errore

- Seleziona meccanismi di gestione della **concorrenza** basati sul rischio aziendale associato a ciascuna richiesta

## Aggregate DDD

Per implementare un sistema a microservizi è necessario che ogni servizio **incapsuli** una porzione di logica di business. Questo vuol dire quindi effettuare una **divisione coerente** con la logica complessiva, cercandone dei confini.

Si utilizza un **modello** che si basa sugli **Aggregate**, un gruppo di oggetti che vengono trattati come **unica unità di modifica** dei dati. Questo ci permette mantenere la **consistenza** all'interno di ogni Aggregate. Ognuno di questi ha un solo **punto d'accesso** chiamato **Aggregate root** e non si può accedere ad alcun oggetto interno senza passare dal root.

## Domain Driven Design tradizionale

- **Modello di dominio:** collezione di **classi interconnesse** che rappresentano oggetti di dominio
- I modelli tradizionali mancavano spesso di confini netti, questo creava delle situazioni in cui i confini e le relazioni erano impliciti
- Confini non netti potrebbero creare inconsistenze

Viste queste limitazioni implementiamo gli Aggregate in un modello DDD che si basa su:

### Entità

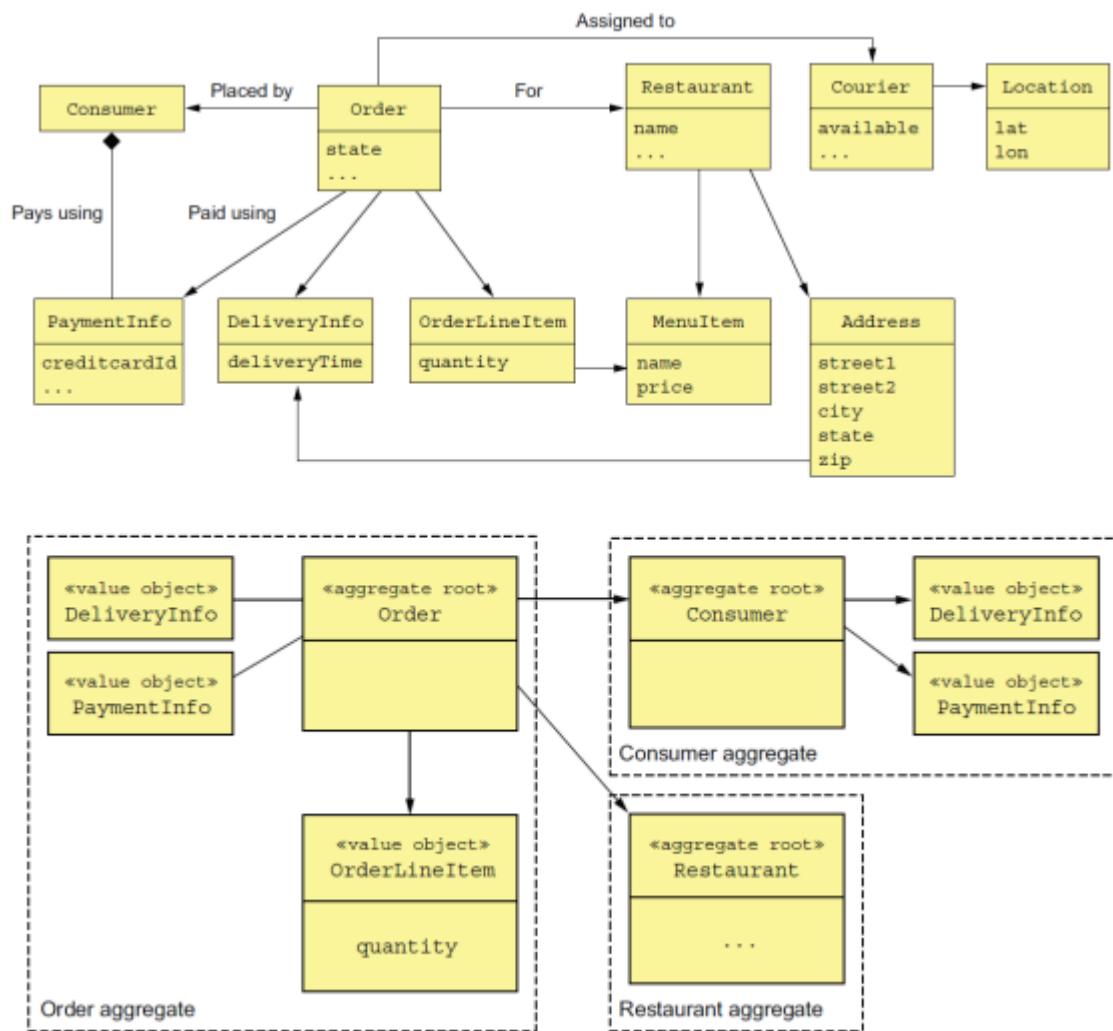
- Oggetto di dominio con identità unica e persistente
- Definiti dalla loro identità, non solo dai loro attributi
- Ognuna di loro ha degli identificatori univoci che la distingue dalle altre
- Possono cambiare attributi nel tempo ma rimangono la stessa identità
- Generalmente persistenti e memorizzate nel DB

Es.: In un sistema di gestione di studenti ogni studente è un'entità con un identificatore unico (anche se due studenti hanno lo stesso nome e cognome sono distinti tra di loro)

## Value Object

- Oggetto di dominio definito solo dai suoi attributi, non ha identità propria
- Immutabili e confrontati in base ai propri valori, se serve un cambiamento se ne crea un altro
- Due value object sono considerati uguali se tutti i loro attributi sono uguali
- Non hanno identificatori univoci, la loro identità dipende solo dai loro attributi

Es.: in un sistema di gestione degli ordini un indirizzo potrebbe essere un value object in quanto viene definito dai suoi attributi e si cambia indirizzo semplicemente se ne crea un altro



## Vantaggi degli Aggregate

- I **confini** delle transazioni sono importanti perché vogliamo che ognuna di queste sia **isolata** all'interno di Aggregate e che si eviti un'eccessiva complessità nella coordinazione
- Si cerca di avere **consistenza locale** per ogni Aggregate
- I cambiamenti vengono o **TUTTI applicati** o **NESSUNO** applicati, questo garantisce **la consistenza**

- Aggregate più piccoli consentono di ridurre la competizione per le risorse e aumentano l'efficienza, Aggregate più potrebbero causare perdita di performance

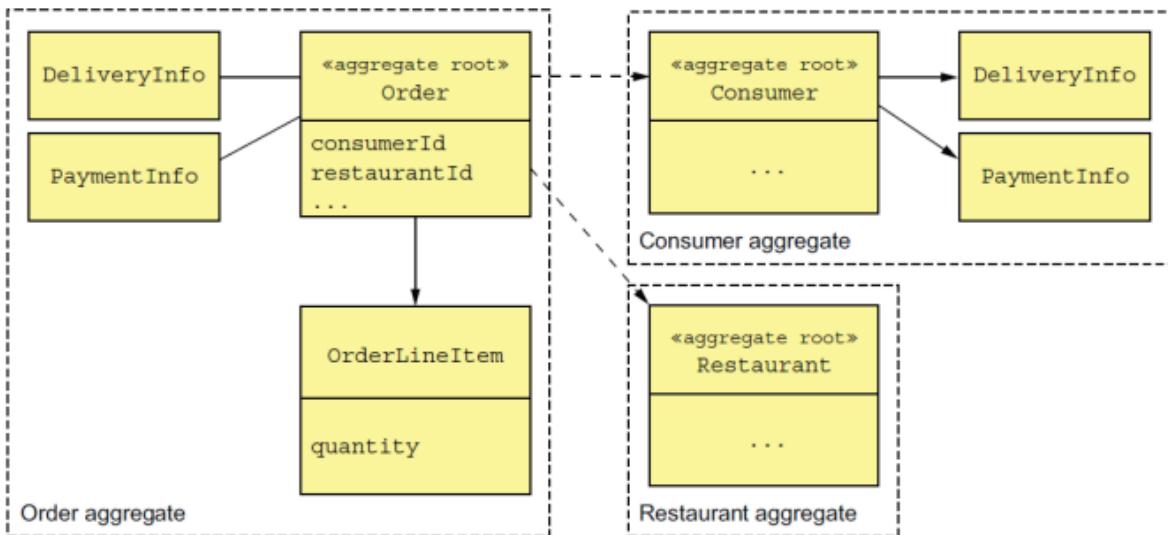
## Regole degli Aggregate nella DDD

Riferimento solo alla Aggregate Root

Solo la Aggregate Root può essere direttamente accessibile e modificabile da altre parti del sistema. Ogni modifica agli oggetti interni dell'Aggregate deve passare attraverso la Root Aggregate

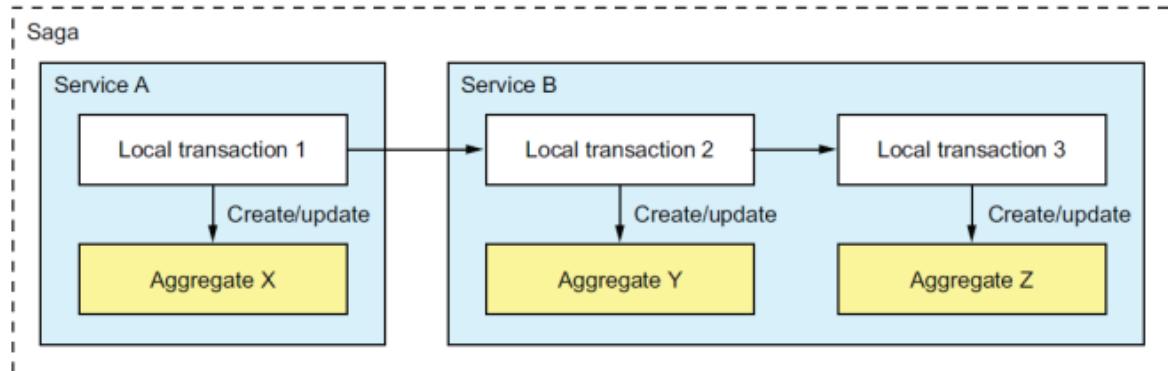
Riferimenti inter-aggregati per chiave primaria

Gli Aggregate si dovrebbero riferire tra di loro utilizzando chiavi primarie (come consumerId) e non riferimenti diretti agli oggetti (ad esempio l'oggetto Consumer). Questo serve a mantenere un accoppiamento basso tra gli Aggregate



Una transazione per Aggregate

Ogni transazione dovrebbe coinvolgere un solo Aggregate. Qualora per una transazione fossero necessarie modifiche a più Aggregate si utilizzerebbero dei SAGA



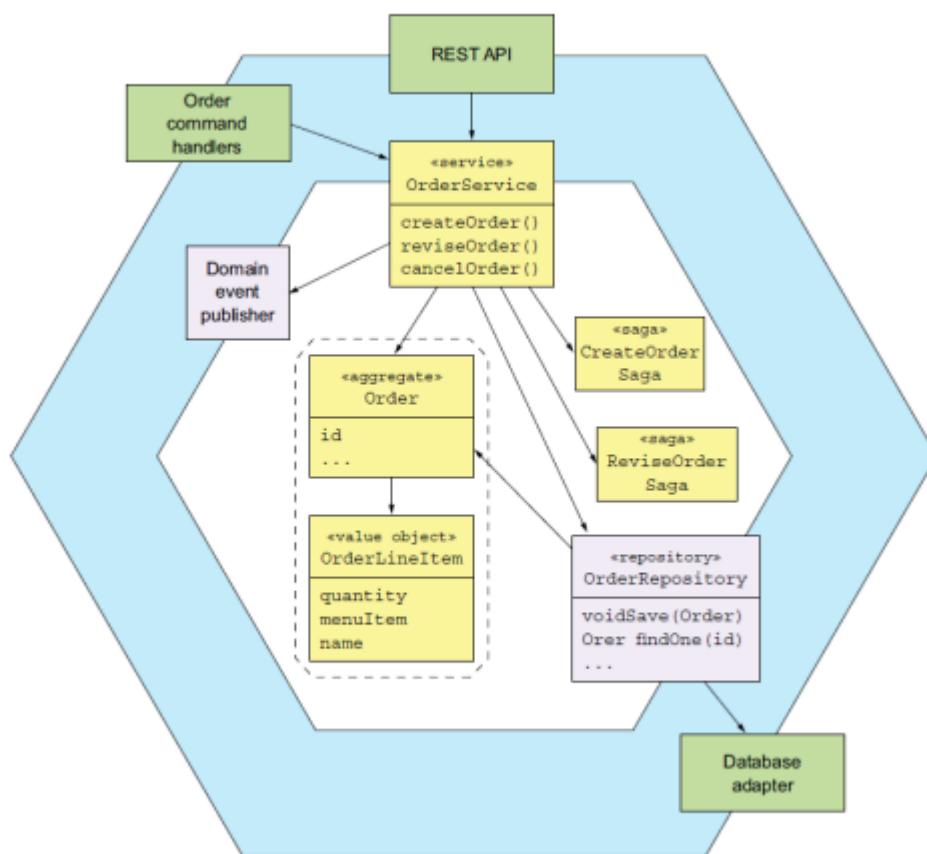
## Granularità

Come detto prima è fondamentale avere degli **Aggregate piccoli** per un discorso di **competizione**. Oltre a questo avere un Aggregate troppo grande vorrebbe anche dire la necessità di **serializzare** gli aggiornamenti (cioè creare una sequenza ordinata e assicurarsi che siano eseguite una per volta).

## Design basato su Aggregate nei microservizi

In questo caso abbiamo:

- **Servizi di dominio** che agiscono come punto di ingresso per la logica di business, si occupano anche dei comandi e delle query
- **SAGA** che gestisce le sequenze delle transazioni locali



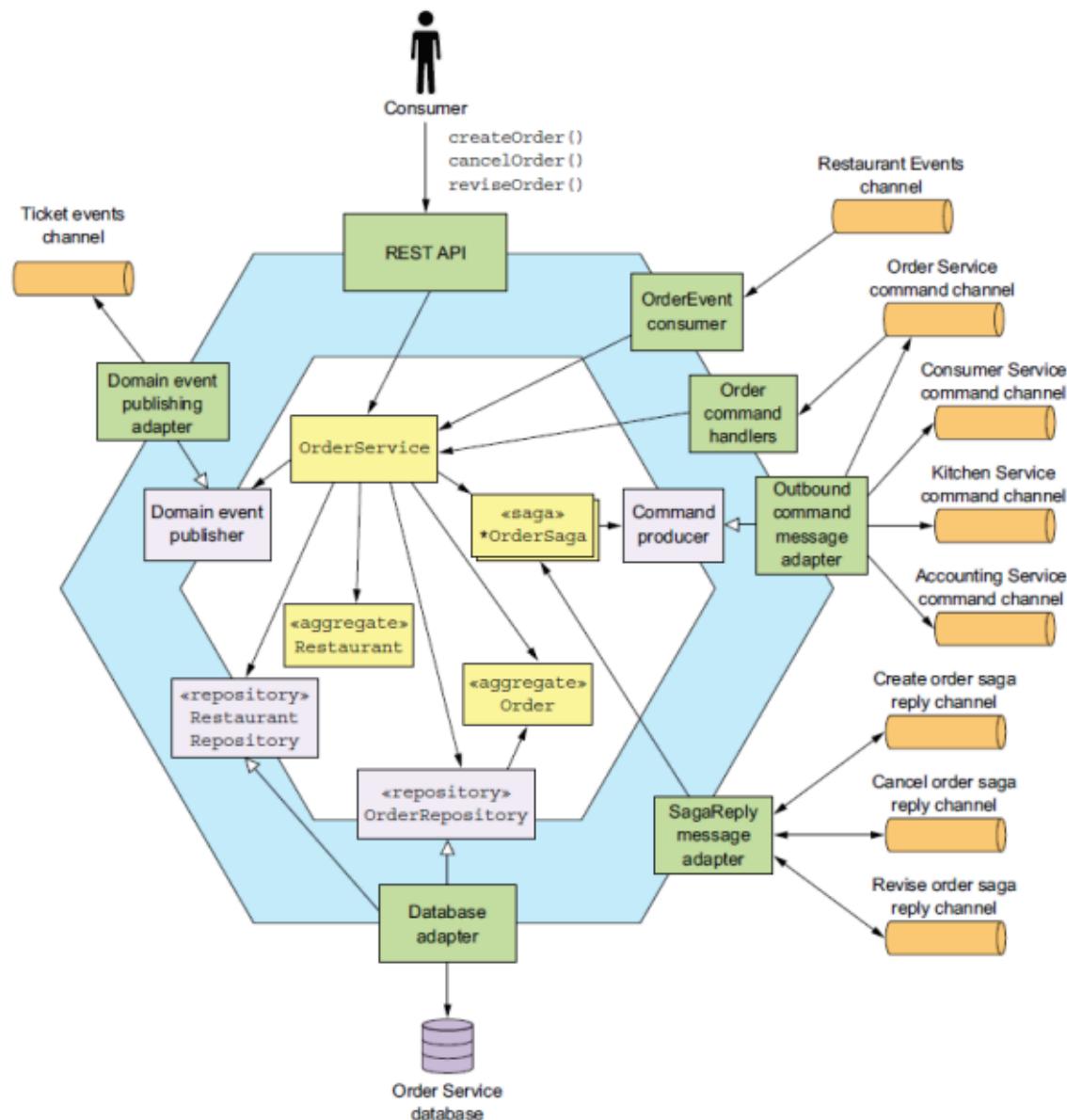
- **Order Aggregate** gestisce lo stato dell'ordine e le operazioni
- **OrderService** gestisce i comandi (come `createOrder()`)
- **OrderRepository** gestisce la persistenza degli ordini
- Le **SAGA** gestiscono le transazioni

## Eventi di dominio

Un evento di dominio è un cambiamento significativo di stato in un Aggregate e vengono utilizzati per comunicare l'avvenimento di qualcosa di importante all'interno del dominio.

- Il nome delle classi degli eventi sono al **participio passato** per indicare che l'evento è già accaduto (OrderCreated)
- Gli eventi contengono proprietà con **dati significativi**, quali valori primitivi o Value Object (come orderId)
- Sono anche inclusi dei **metadati** come ID dell'evento o timestamp

Generazione e pubblicazione



In questo grafico possiamo suddividere tutte le componenti in:

- **Componenti Core**
  - **OrderService**: entry point primario
  - **Order Aggregate**: gestisce i dati degli ordini e le operazioni
  - **Restaurant Aggregate**: replica parziale dei dati di Restaurant Service, serve per confermare la validazione e il prezzo degli item degli ordini
- **Persistenza**
  - **OrderRepository**: gestisce la persistenza

- **RestaurantRepository**: gestisce la persistenza delle repliche di Restaurant Aggregate per le validazioni più vecchie
- **Adattatori verso l'interno**
  - **REST API**: gestisce le richieste da interfaccia utente invocando OrderService per la gestione degli ordini
  - **OrderEventConsumer**: Ascolta gli eventi da Restaurant Service per aggiornare i dati di Restaurant
  - **OrderCommandHandlers**: processa i comandi asincroni dalle SAGA per gestire gli aggiornamenti agli ordini
  - **SagaReplyAdapter**: gestisce le risposte delle SAGA e avvia le corrette azioni SAGA
- **Adattatori verso l'esterno**
  - **DB Adapter**: da accesso a OrderRepository
  - **DomainEventPublishingAdapter**: pubblica gli eventi del dominio di Order agli altri servizi
  - **OutboundCommandMessageAdapter**: Manda comandi agli altri SAGA supportando la coordinazione inter-servizi

## Architettura Event-Driven

In un sistema **distribuito asincrono** si comunica con **eventi**. Le componenti principali sono:

- **Producer**: genera e detecta gli eventi
- **Broker**: direziona gli eventi ai consumer corretti
- **Consumer**: reagisce agli eventi di processo
- **Persistenza**: conserva gli eventi

Volendo strutturare un workflow:

- 1) Il **producer** pubblica un **evento** al **broker** (o un router)
- 2) Il **broker** filtra e **direziona** l'**evento** ai **consumer** iscritti
- 3) Il **consumer** processa l'**evento** in modo **asincrono**

Tutto questo è utile per una serie di motivi:

- I sistemi operano in modo **indipendente** e non si bloccano uno con l'altro quindi si riduce la latenza e si aumenta la responsività
- Diminuisce **l'accoppiamento**, manca proprio quello temporale
- Aumenta la **scalabilità**, possibilità di scalare orizzontalmente

Altre caratteristiche utili sono:

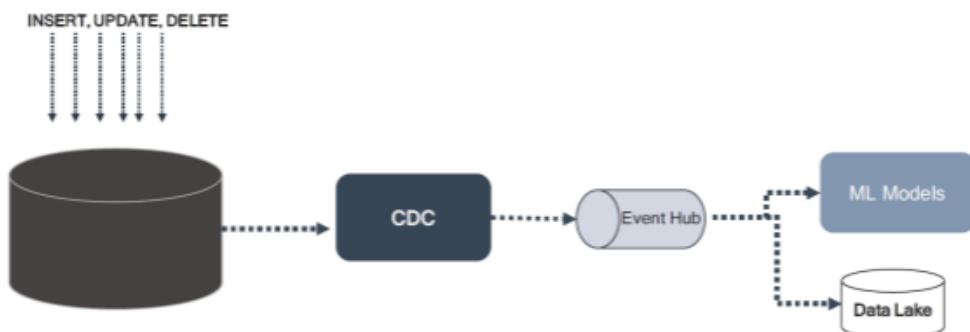
- Gli **eventi** sono tipicamente **immutabili** e **conservati** in un **log** di **eventi** per la **durabilità** e il **riutilizzo**
- I **consumatori** **reagiscono** agli eventi in **real-time**
- Esistono diversi **pattern** per applicare questa strategia:
  - **Event Stream**: **flusso continuo** di eventi per processing sequenziale
  - **Event Sourcing**: gli **eventi** sono la **source of truth** per ricostruire lo **stato**
  - **CQRS**: **separazione** della **lettura** e della **scrittura** per ottimizzare il processing degli eventi

Esistono tre stili per processare gli eventi:

- **Simple Event Processing**: un evento **immediato** che **triggera un'azione** (Es.: sensore IoT che rileva un cambio di temperatura o del fumo)
- **Event Stream Processing**: **filtraggio e processing** di eventi in **real-time** (spesso si usa Kafka)
- **Complex Event Processing**: **analisi su più eventi**, si catturano e analizzano stream di eventi per identificare determinati **pattern**

## Event Streaming

Una definizione utile è quella di **Change Data Capture** ovvero un **processo** che **identifica e cattura** eventuali **cambiamenti** in una collezione di dati. Questo viene fatto con il fine di **sincronizzare i cambiamenti** tra repository e permettere eventualmente di effettuare delle analisi. I cambiamenti vengono **catturati** con **log di transazione**, **timestamp** o **trigger**. In particolare **tracciano i cambiamenti incrementali** (INSERT, UPDATE, DELETE) e utilizzano i **log come source of truth**.



Nell'event Streaming si effettua data processing di stream triggerati da eventi CDC pertanto ne permette delle analisi in real-time e di tracciare la storia completa basandosi sulle modifiche. Abbiamo tre attori:

- **Event Storage**: **conserva** i dati **cronologicamente** (basandosi sul timestamp)
- **Tecnologie di streaming**: Kafka, Kinesis per la **persistenza** degli eventi
- **Stream Processor**: **arricchisce, trasforma e analizza** i dati in arrivo

## Integrazione con Message Queues

- Permette di **riutilizzare** architetture di MQ **già esistenti** senza dover riprogettare tutto il sistema
- Permette di gestire comunicazioni sia **sincrone** che **asincrone**

Sono tuttavia necessari i **connettori** che si occupano di **semplificare la comunicazione** tra MQ e stream di eventi. Nel dettaglio, si possono occupare di leggere messaggi da una coda MQ e inviarli allo stream (o viceversa). Tutto questo permette di creare un'architettura **disaccoppiata**

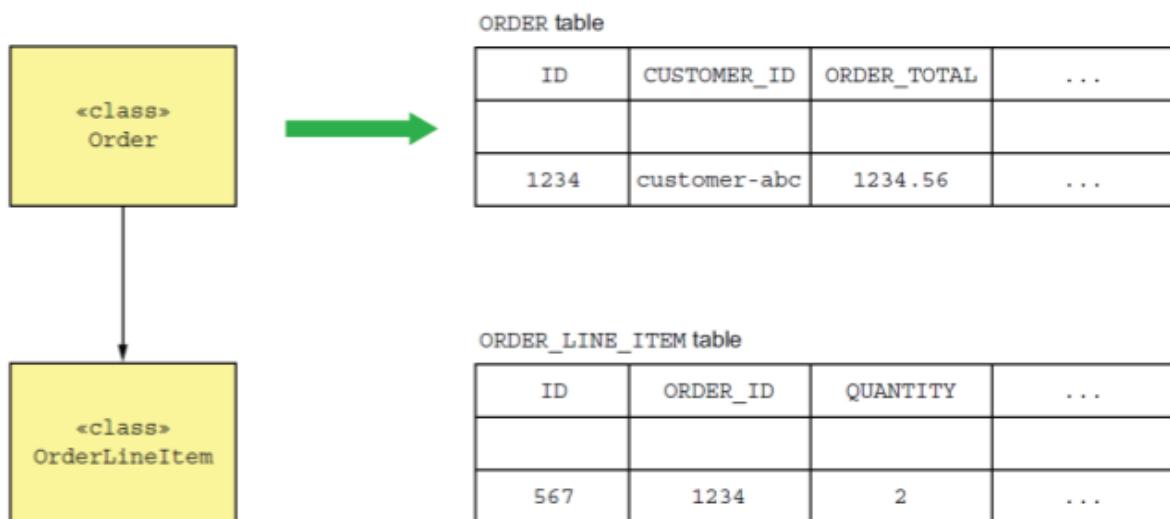
## Event Sourcing

L'event Sourcing è un **pattern architetturale** in cui i **cambiamenti di stato** vengono **conservati** come una **serie di eventi** (al posto di conservarne lo stato). Per avere lo stato attuale di un'entità, il sistema deve **applicare in sequenza** tutti i **cambiamenti** che sono stati effettuati.

- Ci da tutta la **cronologia** dei **cambiamenti**, utile per auditing e troubleshooting
- Ci **semplifica** molto il processo di **revert**

## Approccio tradizionale alla persistenza

- Ogni **classe** viene **mappata** come una **table**
- I **campi** corrispondono alle **colonne**
- Le **istanze** corrispondono a **righe**



Avere un approccio del genere pone delle sfide significative relativamente alla **consistenza** dei **dati** se il sistema sta **mappando entità complesse**, oltre a questo abbiamo:

- Se viene salvato solo lo stato corrente di un Aggregate perdiamo i **dati cronologici**
- **Difficoltà** nel fare **audit**
- I **meccanismi** di **event-publishing** sono aggiunti separatamente e potrebbero non integrarsi benissimo con la logica business
- **Differenza** enorme tra un **modello relazionale** di un database e uno basato su **grafi** di un **modello** di dominio **object-oriented**

## Persistenza con Aggregate

Ogni **ordine** viene **memorizzato** con più **righe** in una **tabella** degli **eventi**. Ognuna di queste rappresenta un **cambiamento di stato** (e il tipo è specificato direttamente nel campo **event\_type**). Questa tabella si **aggiorna** ogni volta che un **Aggregate** viene **creato** o **aggiornato**

The diagram illustrates the structure of a database table for event storage. The table has five columns: `event_id`, `event_type`, `entity_type`, `entity_id`, and `event_data`. Annotations with arrows point from the column headers to specific cells in the table:

- Unique event ID**: Points to the `event_id` column.
- The type of the event**: Points to the `event_type` column.
- Identifies the aggregate**: Points to the `entity_type` column.
- The serialized event, such as JSON**: Points to the `event_data` column.

The table data is as follows:

event_id	event_type	entity_type	entity_id	event_data
102	Order Created	Order	101	{ ... }
103	Order Approved	Order	101	{ ... }
104	Order Shipped	Order	101	{ ... }
105	Order Delivered	Order	101	{ ... }
...	...	...	...	...

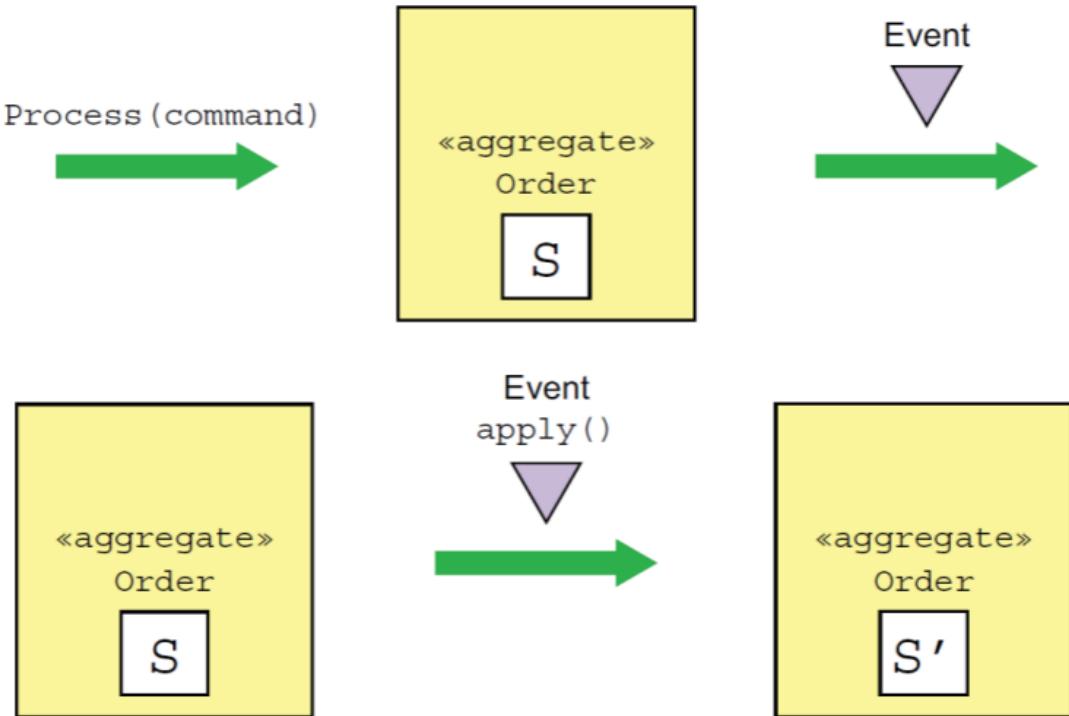
## Applicazione di eventi

`process()`

- Responsabile **dell'elaborazione** di un **comando**
- Appena invocato **esegue la logica di business** necessaria per determinare se il comando può essere eseguito (Es.: verifica se si può ordinare o meno un prodotto in base alla sua disponibilità)
- Se eseguito correttamente fa avviare la routine

`apply()`

- Responsabile **dell'applicazione** di un **evento** per **modificare lo stato** di un'entità
- Appena invocato **aggiorna lo stato** dell'entità in base ai dati contenuti nell'evento



## Aggiornamenti contemporanei

Una transazione potrebbe sovrascrivere i cambiamenti effettuati da un'altra transazione perché più richieste provano ad aggiornare lo stesso Aggregate.

Per questo si utilizza la colonna **VERSION**, che viene **aggiornata** ogni volta che viene **modificato** l'Aggregate. Ogni volta che si legge da un Aggregate si **memorizza** il valore della colonna **VERSION** e quando lo si vuole aggiornare si **include** questo valore all'interno della **condizione** dell'istruzione **UPDATE**.

Quindi:

- 1) Si **legge** dall'Aggregate il **valore** della colonna **VERSION**
- 2) Si **esegue l'aggiornamento** (con la condizione sulla **VERSION** che può essere massimo **VERSION+1**)
- 3) La **prima transazione** che esegue l'aggiornamento **viene effettuata** con successo (e aumenta la **VERSION**)
- 4) Tutte le **altre transazioni** che avevano il valore precedente di **VERSION** vengono **ignorate**

## Event Sourcing e Pubblicazione di eventi

Per la pubblicazione degli eventi si potrebbe pensare a un publisher di eventi che interroga periodicamente la tabella EVENTS per cercare delle nuove voci da pubblicare sul broker.

Questo non è particolarmente conveniente in quanto gli eventi potrebbero non apparire nell'ordine in cui sono avvenuti, creando problemi di coerenza.

Per risolvere utilizziamo il **flag PUBLISHED** nella tabella EVENTS:

- gli **eventi** NON pubblicati avranno PUBLISHED=0, quelli pubblicati avranno PUBLISHED=1
- Il **publisher interroga** la tabella EVENTS e **seleziona** quelli con il flag a 0
- Una volta **pubblicati** si mette il flag a 1

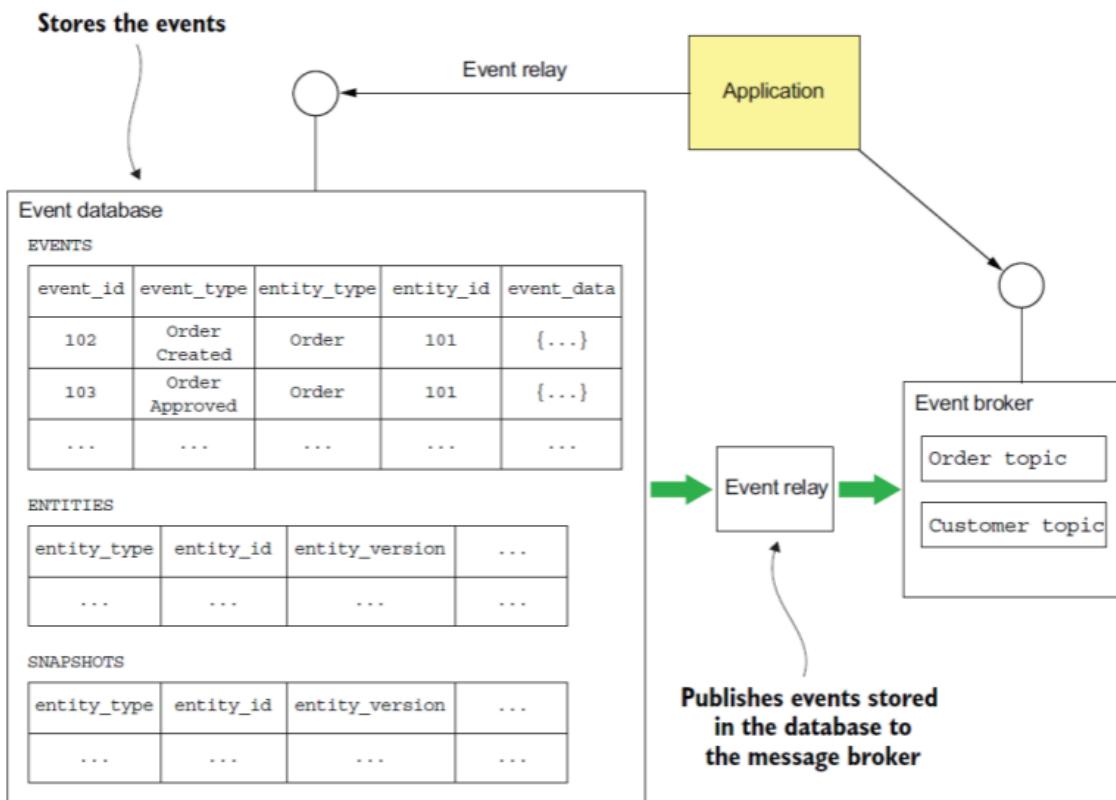
Un approccio più affidabile sarebbe quello con **Log Tailing**, ovvero **leggere** gli eventi direttamente dal **database** con i **log delle transazioni**. In questo modo ci assicuriamo che gli eventi siano **pubblicati in ordine** e che non siano stati saltati degli eventi

## Snapshot

Uno **snapshot** è lo **stato** di un **Aggregate** **fino a uno specifico evento**. Sono fondamentali perché ci possono essere degli Aggregate che accumulano una **quantità notevole di eventi** e diventa inefficiente caricare e riprodurre tutti gli eventi, pertanto **periodicamente si cattura lo stato** dell'Aggregate. Di solito gli **snapshot** vengono **serializzati** in **JSON**

Nel caso in cui si volesse **caricare** un **Aggregate**:

- 1) Si **recupera** la **versione** più recente (versione N)
- 2) Si **caricano** solo gli **eventi** che sono **successi dopo** lo **snapshot** (quindi eventi N+1, N+2)
- 3) Si **ripristina** lo **stato** dell'Aggregate **applicando** allo snapshot tutti gli **eventi avvenuti dopo** lo snapshot



## Processing di messaggi idempotenti

Può succedere che i broker mandino lo stesso messaggio più volte pertanto bisogna pensare ai **consumer** come dei **servizi idempotenti**.

Si utilizza anche una tabella **PROCESSED\_MESSAGES** per **registrare** gli **ID** dei **messaggi già elaborati**, in questo caso se un ID di un messaggio esiste già nella tabella vuol dire che è un duplicato e può essere scartato. Per questo motivo è importante **incorporare** gli **ID** dei messaggi negli **eventi**.

## Domain Events in Event Sourcing

Tutti gli eventi vengono **salvati indefinitamente**, dando la possibilità di effettuare una **ricostruzione storica** partendo dai log. Questo serve per il debugging, l'audit e la conformità normativa.

Con l'evoluzione dei requisiti o la raffinazione del modello di dominio potrebbe cambiare la struttura degli eventi:

- **Livello di Schema**: aggiunta o rinomina di Aggregate, generalmente retrocompatibile quando si tratta di aggiunte
- **Livello di Aggregate**: aggiunta o rimozione di tipi di eventi emessi dagli Aggregate, influenza la comunicazione
- **Livello di Evento**: aggiunta, rimozione o modifica dei campi all'interno di un evento, necessaria una gestione delle versioni degli eventi per la compatibilità

## Upcasting

Lavorando con più versioni di componenti è necessario l'utilizzo di un **Upcaster**. Questo è un componente che **trasforma** delle **versioni più vecchie** di eventi in una **versione** dello **schema corrente** appena vengono caricate. Serve per assicurarsi che il codice dell'applicazione interagisca solo con lo schema più recente degli eventi

- 1) Gli **eventi** vengono **caricati** dai log, l'upcaster **controlla** la **versione** dello schema
- 2) Se l'evento è in una versione più vecchia l'upcaster lo **trasforma** nello schema corrente
- 3) Il codice dell'applicazione interagisce **solo** con eventi nella **versione più recente** dello schema

## Pro e Contro

- Garantisce che i **componenti** siano **aggiornati** con le modifiche più recenti, **mantenendo coerenza e sincronizzazione**
- **Salva** tutti i **cambiamenti** di un Aggregate **come eventi** e permette di effettuare delle analisi temporali
- Eventi **facili da serializzare**, (JSON/XML) riduce la complessità della mappatura degli oggetti in schemi relazionali
- Ogni **modifica** viene **tracciata** con **informazioni** sull'utente che l'ha effettuata e quando è avvenuta
- È possibile "guardare indietro" per analizzare come e perché sono avvenute certe modifiche

- L'event Sourcing si basa spesso sui **message broker** per la **pubblicazione** e la **distribuzione** degli eventi
- Utilizzo di **handler idempotenti** per evitare problemi con i duplicati
- Per ottenere lo stato corrente di un Aggregate è necessario “**foldare**” gli **eventi** (cioè partire da uno snapshot e applicargli tutti i cambiamenti successivi), questo può essere molto **complesso** e richiedere molto tempo
- Per query particolari può essere necessario utilizzare il pattern **CQRS**

## Implementazione con SAGA

Nell'ambito di utilizzo **dell'Event Sourcing** si ricorre spesso alla **coordinazione** utilizzando il **SAGA** pattern. Questo garantisce la **consistenza** di **dati** tra tutti i servizi. Chiaramente ogni passo nel SAGA deve essere **atomico** e bisogna gestire in modo molto preciso i workflow multi-step.

- Per gli **RDBMS** bisogna sono **supportate** le transazioni **ACID**
- Per i **NoSQL** bisogna trovare degli **approcci diversi** per mantenere **atomicità** e **consistenza**

### Coreografia

- Gli Aggregate **emettono** degli **eventi** quando vengono **aggiornati**
- I **consumer consumano** l'evento, **aggiornano** i relativi Aggregate ed **emettono** nuovi **eventi**

Questo permette di:

- **Semplificare la comunicazione** e usare un workflow **event-driven**
- Utilizzare **messaggi** basati su **IPC** e **aggiornamenti atomici**

Chiaramente le sfide sono relative a:

- **Rappresentare dei cambi di stato**
- Far funzionare la **coordinazione** anche quando **non** ci sono **cambiamenti di stato**

### Orchestrazione

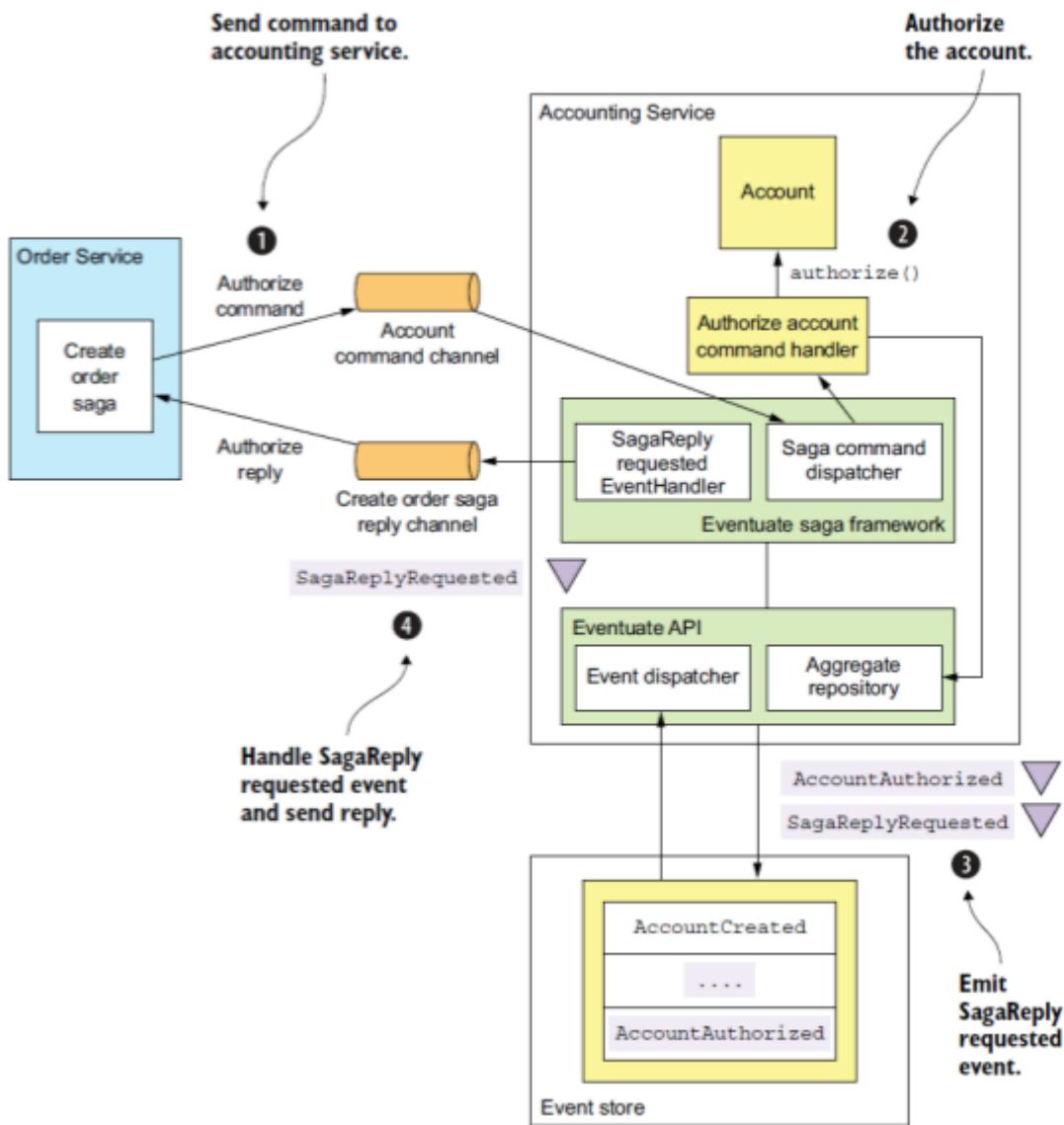
L'**orchestrazione** viene usata per **coordinare** workflow multi-step, in questo caso:

- **RDBMS-based event store**: usano le **transazioni ACID** per aggiornare gli Aggregate e creare gli orchestratori **atomicamente**
- **NoSQL-based event store**: richiedono un **gestore di eventi** per creare l'orchestratore SAGA dagli eventi emessi

È anche necessario utilizzare degli **ID univoci** per assicurarsi che non vengano creati **duplicati** nelle SAGA

### Esempi di applicazione

- 1) **Create Order SAGA** invia il comando **AuthorizeAccount** attraverso il **canale di messaggistica** (Account command channel)
- 2) **Accounting Service** aggiorna l'Aggregate Account
- 3) **Accounting Service** emette **AccountAuthorized** e **SagaReplyRequestedEvent**
- 4) **SagaReplyRequestedEventHandler** manda una **risposta** tramite il **canale** per le **risposte** della **SAGA**



Quindi, riprendendo le sfide:

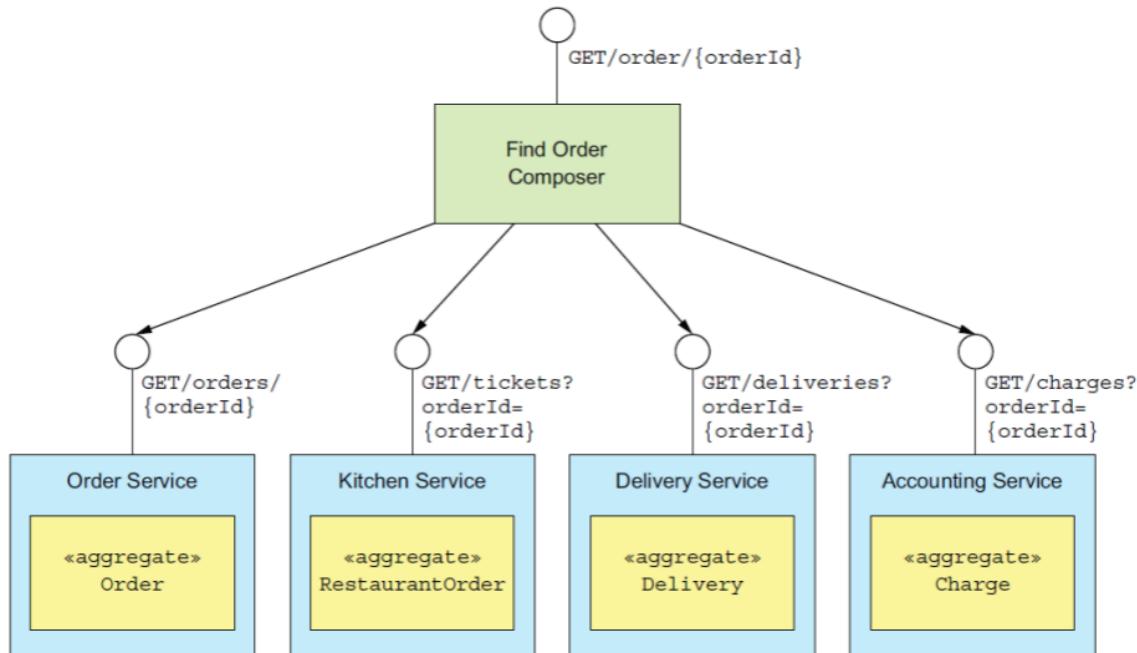
- Bisogna assicurarsi che la gestione dei messaggi-comando sia idempotente per la questione dei duplicati
- Bisogna mandare messaggi di risposta atomici all'orchestratore SAGA
- Bisogna memorizzare il messageld degli eventi generati dopo il processamento di un comando
- Prima di aggiornare l'Aggregate bisogna controllare che il messageld non esista già negli eventi
- Il SAGA non dovrebbe cambiare lo stato dell'Aggregate

## API composition pattern

Nelle applicazioni **monolitiche** tutti i dati sono all'interno di un **singolo database** quindi le query hanno bisogno di un solo SELECT (anche per unire le tabelle). In questo caso, in un'architettura a **microservizi**, i dati sono **divisi** tra vari servizi pertanto la query deve andare a prenderli in parti diverse. L'**API composition pattern** ci permette di **implementare una query** che “raccoglie” i dati da servizi diversi, **combinandone i risultati**.

In questo esempio vogliamo eseguire una query per cercare l'ordine partendo dal suo ID:

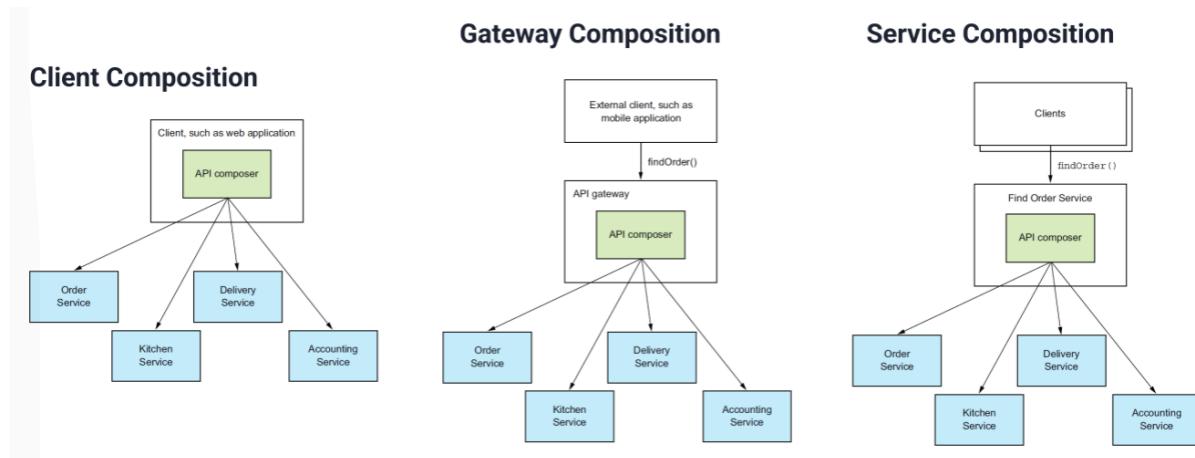
- Viene esposto solo un endpoint REST, ovvero **GET /order/{orderId}**
- Questo chiama 4 servizi:
  - **Order Service**: restituisce i dettagli di base dell'ordine
  - **Kitchen Service**: restituisce lo stato dell'ordine relativamente al ristorante
  - **Delivery Service**: recupera i dettagli della consegna
  - **Accounting Service**: si occupa delle informazioni relative al pagamento



## Pro e contro dell'API Composition pattern

- **Modo intuitivo** di implementare le operazioni di **query** in un'architettura a microservizi
- Permette la **composizione** dei **dati** da più servizi, supportando anche dei **requisiti flessibili**
- Consente una **logica di aggregazione** affine ai bisogni dell'applicazione
- **Più** servizi che fanno **chiamate** e query al database richiedono un quantitativo notevole di risorse
- Più sale il numero di **servizi** e più diminuisce la **disponibilità**
- Fare query su più database potrebbe restituirci **dati inconsistenti**

## Schemi principali di API compositor



### Client Composition

- Il client **invia richieste separate** a ciascun microservizio per ottenere i dati necessari
- Il client **riceve le risposte** dai vari microservizi e le **combina** per formare una **risposta unificata** (grazie al composer)
- **Semplice** da implementare, non richiede particolari modifiche ai microservizi esistenti
- **Aumenta la latenza** e il carico sul client

### Gateway Composition

- Il **client** invia una **singola richiesta** al **gateway API**
- Il **gateway API** invia le **richieste** ai vari microservizi, le **aggredisce** e le **restituisce** come **risposta unificata** al client
- Riduce la **latenza** per il client e centralizza la logica di composizione delle API
- Il gateway API potrebbe diventare un **collo di bottiglia**, va gestito bene per la **scalabilità**

### Service Composition

- Il client invia una **singola richiesta** all'API **composer**
- L'API composer **invia le richieste** ai microservizi ed **esegue** un **join** in memoria dei dati, restituendo una **risposta unificata** al client
- Permette di **riutilizzare** determinate **query** o farne di **complesse**
- **Centralizza** la logica di **composizione**, riducendo la **complessità** per il client
- Potrebbe aumentare il **traffico di rete** per via delle numerose richieste ai database

### Chiamate Parallelle e Sequenziali

Utilizzare **chiamate parallele** è particolarmente utile se si vuole **ridurre la latenza** in quanto quello che si fa è fare **richieste parallelamente** tra più **servizi**. In alcuni casi però questo non è possibile, qui entrano in gioco le **chiamate sequenziali**, ovvero quando la **chiamata** di un **servizio** dipende da quella **precedente** (e così via). Queste andrebbero utilizzate solo nel caso in cui i risultati di una chiamata dipendano strettamente da quelli della chiamata precedente.

In questi casi si cerca di rispettare un **paradigma** di programmazione chiamato **Reactive Programming**. I capisaldi su cui si basa sono:

- Esecuzioni **non bloccanti** per aumentare la scalabilità
- Gestione efficiente delle **operazioni concorrenti**

È possibile perché si effettuano più **chiamate ai servizi senza il bisogno di aspettare** la fine delle altre e perché le dipendenze vengono **automaticamente aggiornate**.

## Esempi di applicazione

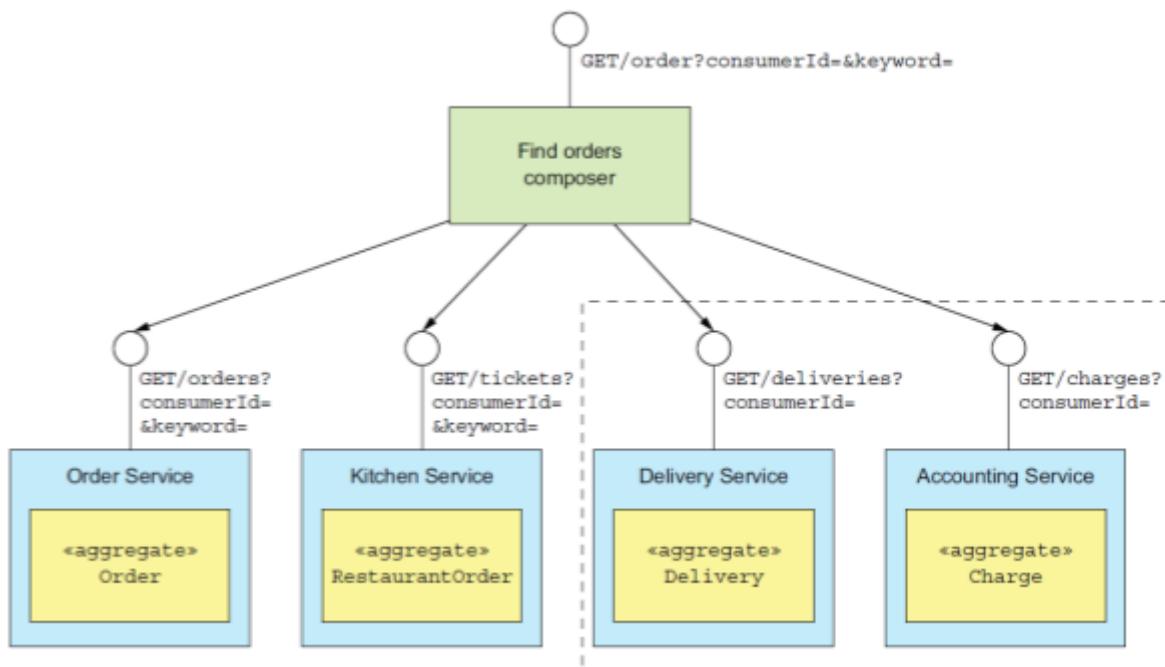
findOrderHistory()

Vogliamo creare un metodo che accetti come parametri in input:

- **consumerId**: l'ID che contraddistingue l'utente
- **pagination**: la pagina dei risultati
- **filter**: campi come il tempo massimo degli ordini, il nome del ristorante, ecc...

Ci deve restituire un elenco con tutti gli ordini effettuati. Ci sono dei problemi però:

- Solo i servizi degli **Ordini** e della **Cucina conservano** gli item del menu per keyword
- Solo il servizio degli **Ordini** può **ordinare cronologicamente** in base alla creazione dell'ordine



Abbiamo due possibili approcci:

- **Join nella memoria**: prendere **tutti gli ordini** degli utenti e effettuare un **join** con il **Composer** (cosa molto difficoltosa con dataset molto grandi)
- **Bulk fetch in base all'ID**: prendere prima gli ordini da Ordini e Cucina in base all'ID dell'utente e poi **filtrare** negli altri servizi in base agli ID degli ordini (in questo caso bisogna che i servizi supportino le bulk fetch)

findAvailableRestaurants()

Vogliamo creare un metodo che ci **restituisca** una **lista di ristoranti** che spediscano a un certo indirizzo in una certa tempistica. Le query saranno fatte in base alla **distanza**, quindi abbiamo bisogno di un database compatibile con l'output:

- **Database ottimizzato**: un normale db con le **estensioni geospaziali**
- **Altri Database**: con **repliche** o **librerie** specializzate

Quindi i problemi principali sono:

- **Sincronizzare le repliche** con i dati originali
- **Sovraccarico delle responsabilità**, il servizio Ristorante dovrebbe occuparsi per primo della gestione dei menu e dei profili, troppo traffico potrebbe abbassare l'affidabilità del sistema

Le soluzioni sono:

- **Delegare l'implementazione** della query in un altro servizio
- **Utilizzare CQRS** per la gestione delle repliche

## CQRS

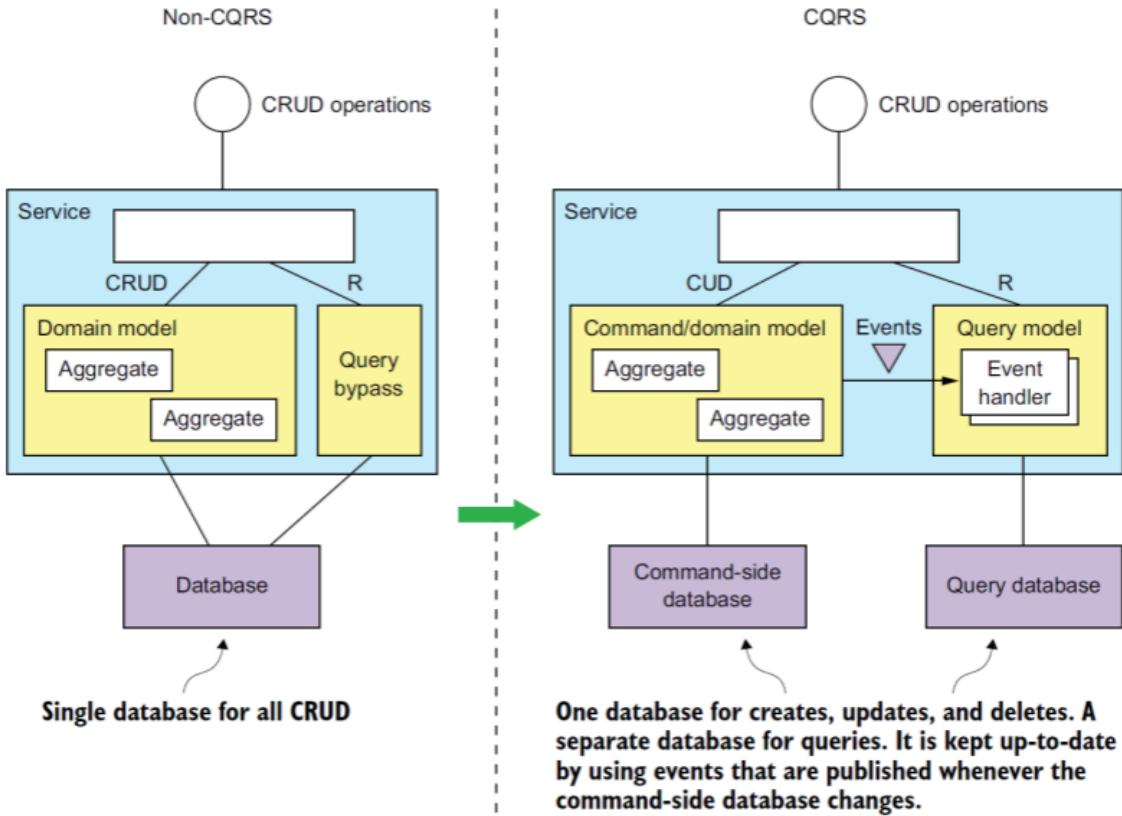
**Design Pattern** che **separa i comandi (CUD)** dalle **query (R)** per migliorare la scalabilità e l'efficienza

Ogni lato ha il suo modello dedicato:

- Il **lato dei comandi** gestisce operazioni di **CREATE, UPDATE** e **DELETE**, pubblica eventi di dominio quando ci sono cambiamenti ai dati
- Il **lato delle query** si occupa di **gestire le query** e utilizza dei database **ottimizzati** per quello

C'è anche da dire che:

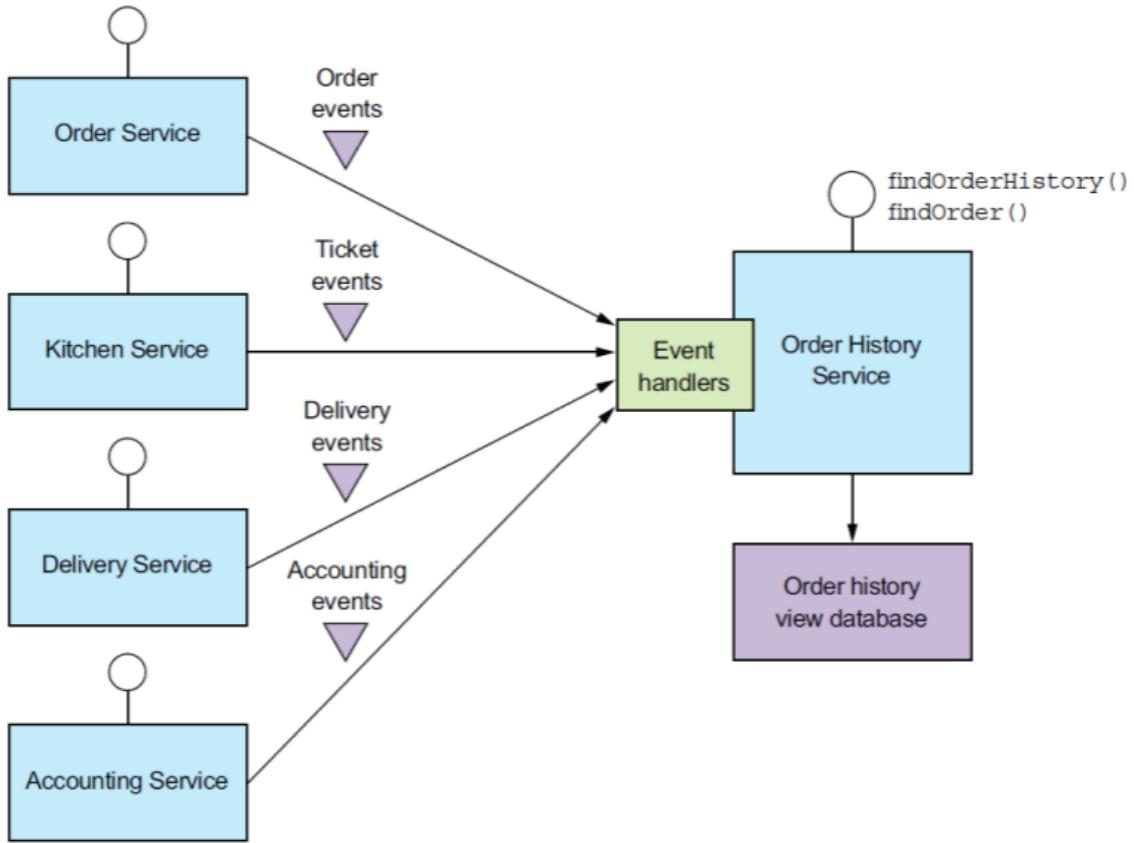
- La **sincronizzazione** del lato delle query avviene perché l'altro lato **invia** degli **eventi** appena vengono **modificati** dei **dati**
- Il lato delle **query** può utilizzare **qualsiasi tipo di database**, in generale si sceglie il migliore in base ai **requisiti**



- I **servizi** pensati per questo utilizzo **espongono soltanto le API, non i comandi**
- I **databases** per le query vengono **sincronizzati** perché c'è un **event handler** che si **sottoscrive** agli eventi dei **servizi** che gli interessano e viene **notificato** per ogni ordine rilevante

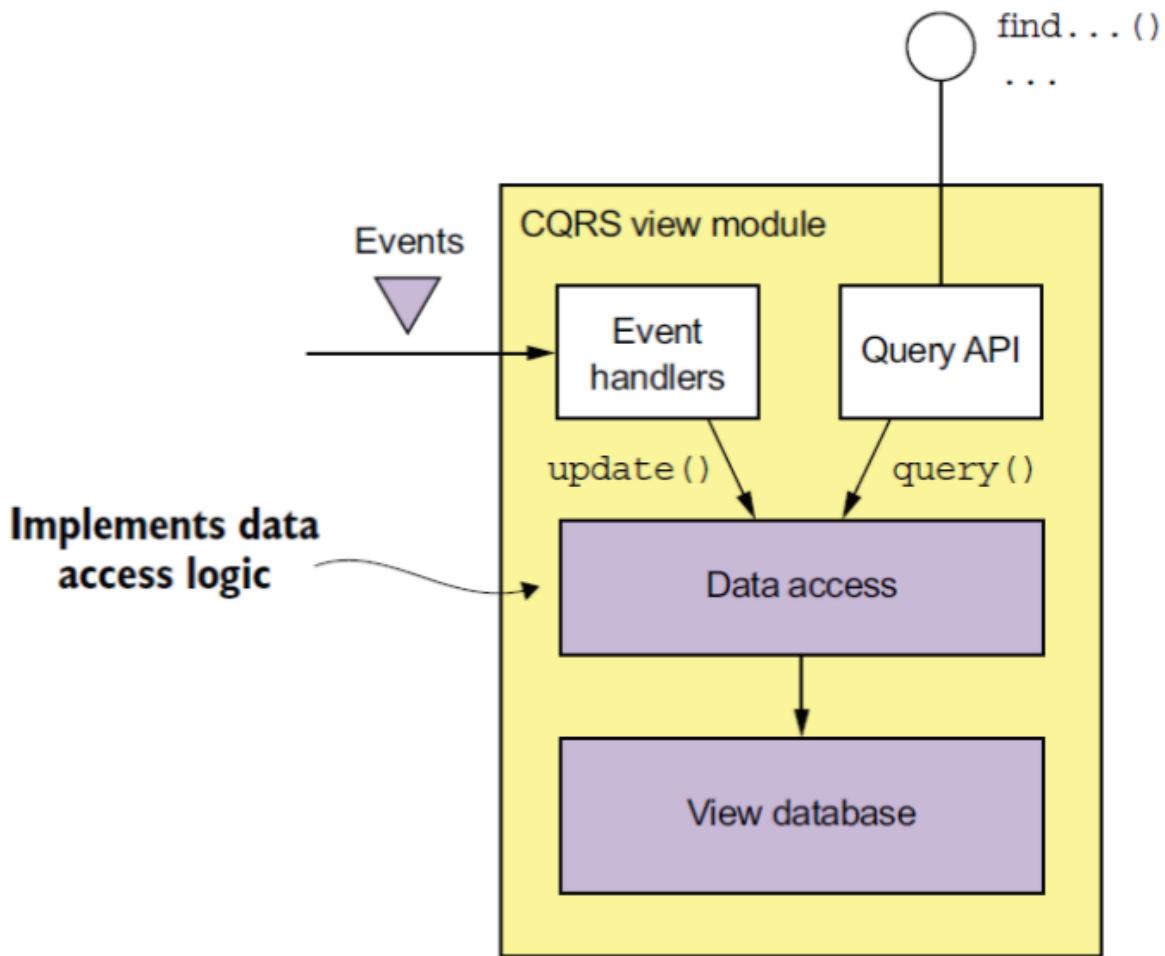
Riprendendo gli esempi di prima abbiamo:

- Una **view cross-service**, ovvero il caso in cui vogliamo avere tutta la cronologia di ordini effettuata da un cliente e in questo caso il servizio apposito si iscrive a tutti gli eventi dei servizi Order, Delivery e Accounting, mantenendo un database appositamente ottimizzato per gestire le query che ci aspettiamo da `findOrderHistory()`
- Una **view standalone** che ci dia il risultato la lista di ristoranti in base alla distanza e in questo caso abbiamo un servizio che si occuperà di mantenere i dati geospaziali grazie a un database appositamente ottimizzato per questo. Anche qui, il servizio in questione si dovrà sottoscrivere agli eventi del servizio Restaurant



## Architettura di una View CQRS

- **View Database:** conserva i dati ottimizzati in base alle operazioni di query
- **Data access module:** gestisce la logica di database per aggiornamenti e query
- **Event Handlers:** si sottoscrivono agli eventi e aggiornano il database
- **Query API module:** espone le API per le query ai client



Per progettare un sistema del genere bisogna:

- **Scegliere un database** in base alle **necessità** delle query (tipo geospaziali, ricerca di testo, ecc...)
- Progettare un sistema che **supporti le query** in modo **efficiente**
- Assicurarsi che gli **aggiornamenti** siano **idempotenti**
- Gestire efficacemente gli **aggiornamenti concorrenti**
- Implementare dei **meccanismi** per **costruire o ricostruire** le view in modo **efficiente**
- Provvedere alla creazione di strategie per i client per gestire eventuali **inconsistenze temporanee**

Esistono anche dei criteri per scegliere il database:

- **Ottimizzato per le query specifiche** in base alla view
- **Gestisce efficacemente gli aggiornamenti** dell'event handler

Possiamo anche optare per:

- **NoSQL**
  - Modello di dati **flessibile**
  - Buona **scalabilità orizzontale**
  - **Prestazioni elevate** per specifici tipi di query
- **SQL**
  - Supporto per **query complesse**

- **Estensioni avanzate (tipo geospaziali)**
- Ottimo per **reportistica e analisi**

Oltre a questo, c'è anche la possibilità di scegliere in base a come conserva i dati:

- **Document Store**
  - Utilizzo di MongoDB per view **basate su JSON**
  - Adatto a dati **non strutturati**
  - **Flessibile** e facile da usare
- **Graph database**
  - Utilizzo di Neo4j per rilevazione di frodi
  - **Ottimizzato per relazioni complesse**
  - Eccellente per **analisi di grafi e rilevazione di pattern**
- **SQL database**
  - Utilizzo per reportistica e analisi per l'intelligenza aziendale
  - Ottimo per **analisi e reportistica**
  - Supporto per query **complesse e transazioni**

## Data access Module

Precedentemente si è parlato del ruolo del **Data Access** module, questo serve per:

- Effettuare gli **aggiornamenti** al database, che vengono triggerati dagli eventi che arrivano agli event handler
- Eseguire le **operazioni di query** che gli arrivano dal query module
- **Mappare** i tipi di dati tra la **logica di business** e le **API** del database

Ci possono essere casi in cui l'event handler vuole effettuare degli **aggiornamenti contemporaneamente**, stiamo parlando di **aggiornamenti concorrenti**. Abbiamo due strategie con cui gestire questi comportamenti:

- **Locking pessimistico**: prevenire gli aggiornamenti concorrenti **bloccando** i record durante le operazioni
- **Locking ottimistico**: permettere l'accesso contemporaneo ma **verificare la consistenza** prima di applicare gli aggiornamenti

## Event Handler

- Bisogna gestire gli **eventi duplicati**, possibilmente avendo degli event handler che sono idempotenti, qualora non fosse possibile bisogna detectare e rimuovere i duplicati
- In base al tipo di database:
  - **SQL**: ID dell'evento viene salvato in una tabella **PROCESSED\_EVENTS** nella stessa transazione che aggiorna l'Aggregate
  - **NoSQL**: ID dell'evento viene salvato direttamente all'interno del record aggiornato
- L'ID degli ordini è un valore **monotono crescente** e si salva solo il **valore massimo** per le istanze degli Aggregate

Un client potrebbe fare una query subito dopo un comando e non vederne gli aggiornamenti per via di **ritardi relativi ai messaggi**, quindi:

- Dalla parte relativa ai **comandi** si restituisce l'**ID dell'evento** al client
- Dalla parte relativa alle **query** si controlla se la view ha già processato l'evento:
  - Restituisce **la query** se è stata correttamente **aggiornata**

- Ritorna un **errore** se la view **non** è stata **aggiornata**

## Aggiungere o modificare una view

### Aggiunta

- **Implementazione di event handlers e API** per le query pensate per la nuova view
- **Scelta del database** apposito in base ai requisiti delle query
- **Deployment** della piattaforma e sincronizzazione con gli eventi

### Modifica

- **Aggiornamento degli event handlers** per adattarsi a cambiamenti di schema o sistemare errori logici nella view
- **Ricostruzione** della **view** con unione dei dati storici e in real-time per minimizzare il downtime

### Implementazione efficiente

I broker non conservano dati indefinitamente (di solito si parla di limiti di 30 giorni), pertanto abbiamo bisogno di servizi che si occupino dello storage. Con il passare del tempo la creazione della view potrebbe essere molto **lenta e pesante**, di solito si utilizzano gli **snapshot** e gli si applicano gli ultimi eventi

## Pattern API esterne

Come abbiamo visto, l'app FTGO è pensata per essere utilizzata da un gruppo eterogeneo di client, tra questi abbiamo:

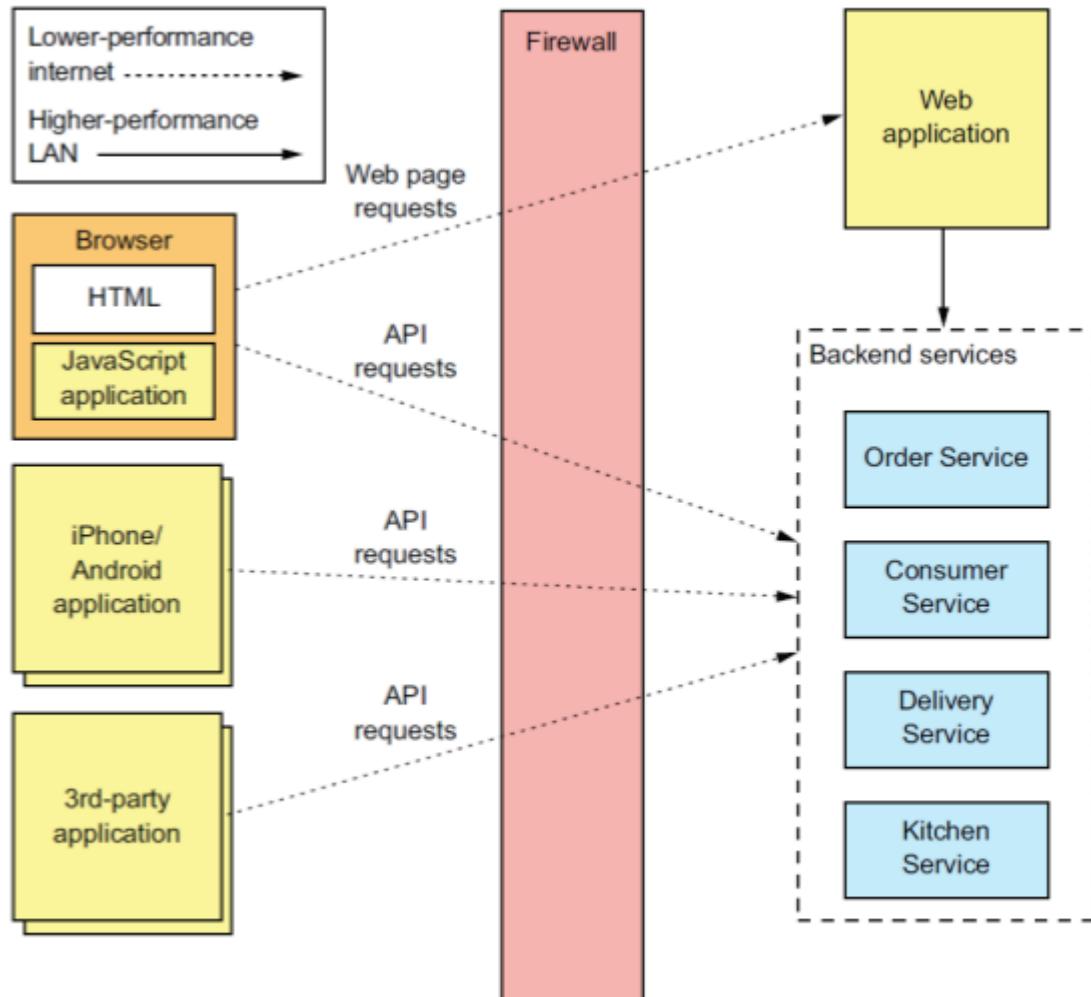
- Sito web interno per Admin e politiche di gestione
- App JS per interagire con i servizi utente dal browser
- App mobile per consumatore o corriere
- Applicazioni terze parti che utilizzano le API

Tutto questo è molto complesso da gestire perché:

- Alto numero di richieste da parte del client per ottenere i dati
- Alta latenza e bassa UX
- I client sono altamente accoppiati alle API perché queste ultime sono altamente specializzate in base al servizio (un cambio interno e bisogna stravolgere l'architettura)
- Meccanismo IPC poco pratico e poco accessibile per i client esterni (soprattutto fuori dal firewall)

Come abbiamo detto c'è un "interno" e un "esterno" rispetto al firewall. In questo caso abbiamo:

- Interno al firewall: web-app che accede ai servizi e connessa tramite una LAN ad alte performance
- Esterno al firewall: tutti i servizi pensati per gli utilizzatori finali, utilizzano reti a più basse performance



Com'è possibile notare il client fa da compositore, unendo tutte le risposte manualmente, questo sposta tutto il costo computazionale su di esso e aumenta la latenza.

Per una serie di motivazioni questo approccio è molto svantaggioso:

- C'è bisogno di effettuare molteplici richieste (alcune di queste potrebbero essere concorrenti) e aumentare la latenza di rete
- Lo sforzo computazionale richiede parecchie risorse in locale e farebbe consumare velocemente la batteria del dispositivo
- Gli sviluppatori mobile sarebbero sovraccaricati di responsabilità non proprie, spostando l'attenzione sulla creazione di API composito
- Ci sarebbe anche alto accoppiamento tra backend e frontend, in quanto quest'ultimo dovrebbe sempre essere aggiornato con il backend

Riguardo le IPC invece:

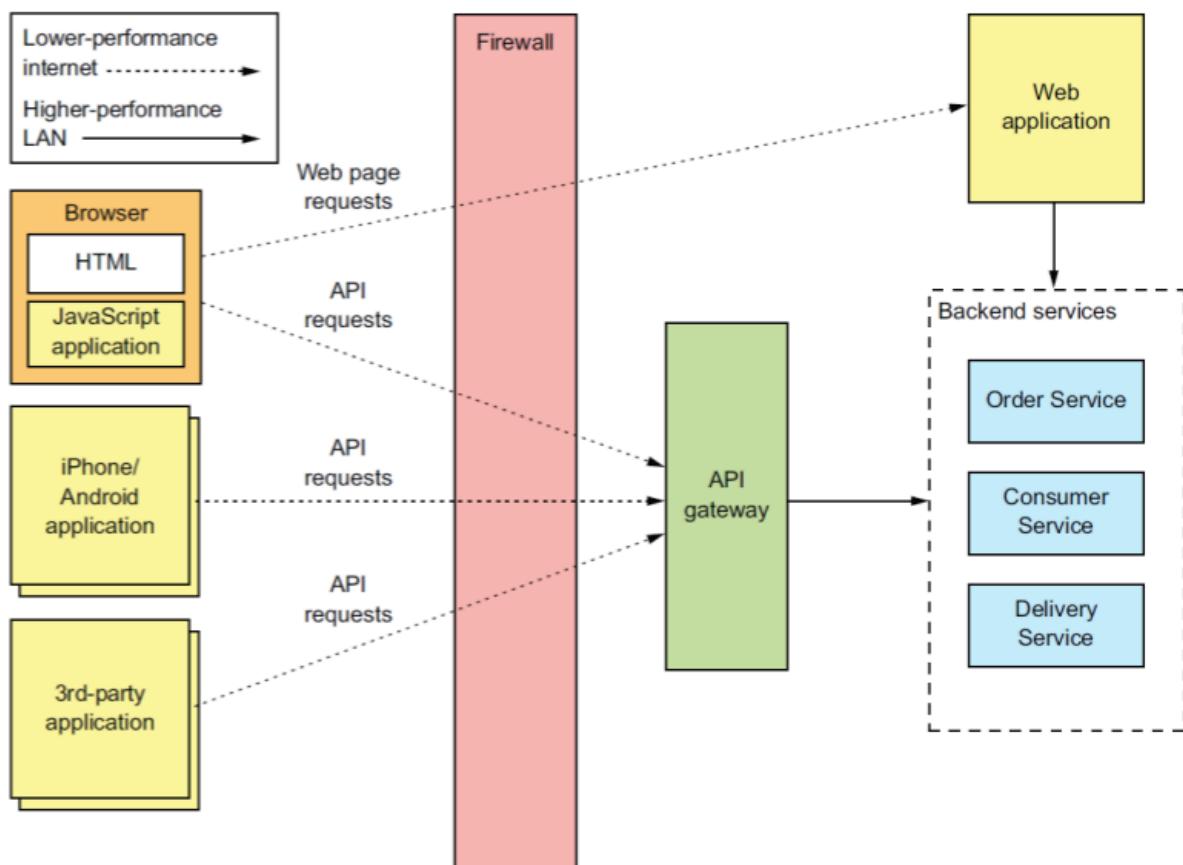
- AMQP e gRPC potrebbero non funzionare bene con i client mobile
- Alcuni protocolli di potrebbero dare problemi al di fuori del firewall

I bisogni pertanto sono differenti in base allo scopo del servizio. Questi infatti saranno

- Per l'accesso diretto (web-app) avremo bisogno di operare all'interno del firewall, sfruttando una rete con un'ottima larghezza di banda e bassa latenza
- Per i servizi vogliamo gli aggiornamenti dinamici per le app JS, bassa latenza e una efficienza tale da rendere accettabile l'esperienza

## Definizione e funzionamento

- Un servizio che fa da singolo entry-point per l'applicazione basata su microservizi
- Deve redirezionare le richieste, fare API composition e Protocol Translation
- Simile al facade, espone delle interfacce molto semplici che permettono di ignorare tutto ciò che c'è "sotto"



Riguardo al Routing utilizza delle routing map per mappare i metodi HTTP e i percorsi con gli URL dei servizi interessati (Es.:

`GET /orders/123` diventa `http://order-service/orders/123`), la sua funzione sarà molto simile a quella del proxy inverso, un servizio offerto anche da NGINX per fare da intermediario tra le richieste dei client e i server backend in base all'URL.

Poi c'è anche l'utilizzo per fare API composing, che come abbiamo già visto, servirà soltanto a raggruppare il risultato di query che vengono fatte a diversi servizi da una singola chiamata.

Infine si occupa anche di tradurre i protocolli esterni (REST) in protocolli interni e specifici per i servizi (come gRPC e AMQP), per trarne vantaggi da entrambi

## Responsabilità dell'API Gateway

- Si occupa di **verificare l'identità** dei client (**autenticazione**)
- Si **assicura** che i client abbiano un **accesso appropriato** (**autorizzazione**)
- Controlla che le **richieste non superino** un certo numero e creino **overload**
- Fanno **caching** per evitare di fare ripetute chiamate al backend per richieste frequenti

- Traggono l'utilizzo delle API per uso statistico
- Loggano le richieste che vengono fatte alle API per audit e debug
- Possono essere integrati con gli **edge service** (per motivazioni legate al centralizzazione e alla diminuzione di carico sui server backend)
- Possono fare addirittura da **edge service**, con il vantaggio di diminuire la latenza e eliminare a monte il problema delle dipendenze

## Architettura

Utilizza un'architettura a layer, modulare:

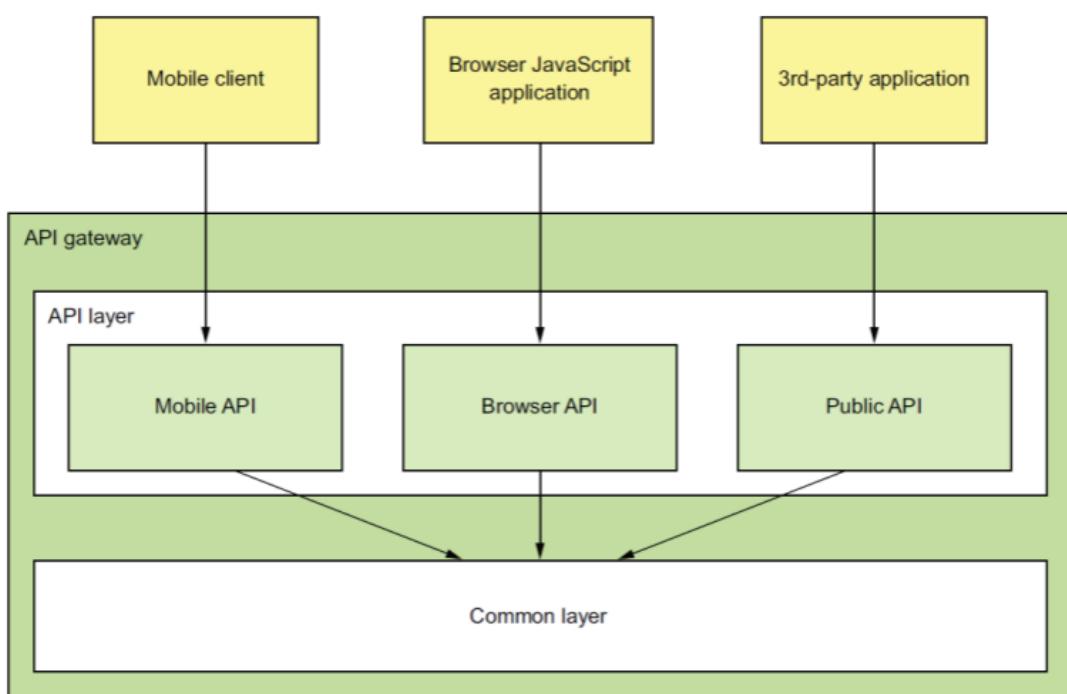
- Livello API: contiene moduli indipendenti in base alla tipologia di client
- Implementa funzioni condivise quali quelle per l'autenticazione e per il rate limiting

Ogni modulo API serve un client specifico, quindi abbiamo:

- **Mobile API**: pensato appositamente per il client mobile
- **Browser API**: pensato per le applicazioni JS su browser
- **Public API**: espone i servizi a sviluppatori di terze parti

Le operazioni che effettuano sono relative a:

- Instradare a servizi con regole basate su **configurazione**
- Utilizzare la **API composition** per richiedere dati da più servizi



## Ownership

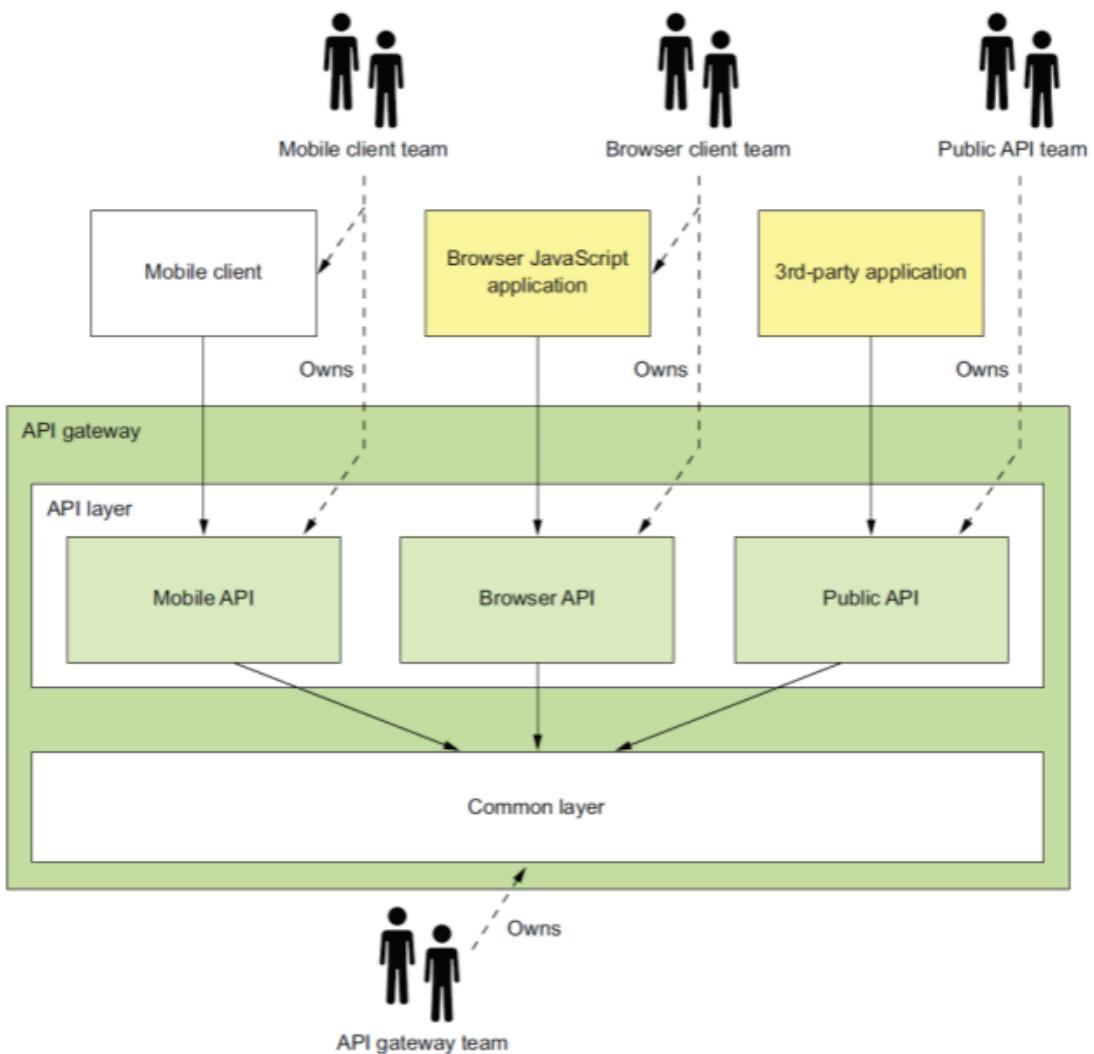
### Centralizzata

- Il Gateway è gestito da un singolo team o un gruppo ristretto

- La sicurezza, il routing e la trasformazione dei dati sono uniformi e gestite centralmente
- Semplice da gestire e monitorare, una sola source of truth
- Potrebbe diventare limitante relativamente alla scalabilità orizzontale

## Decentralizzata

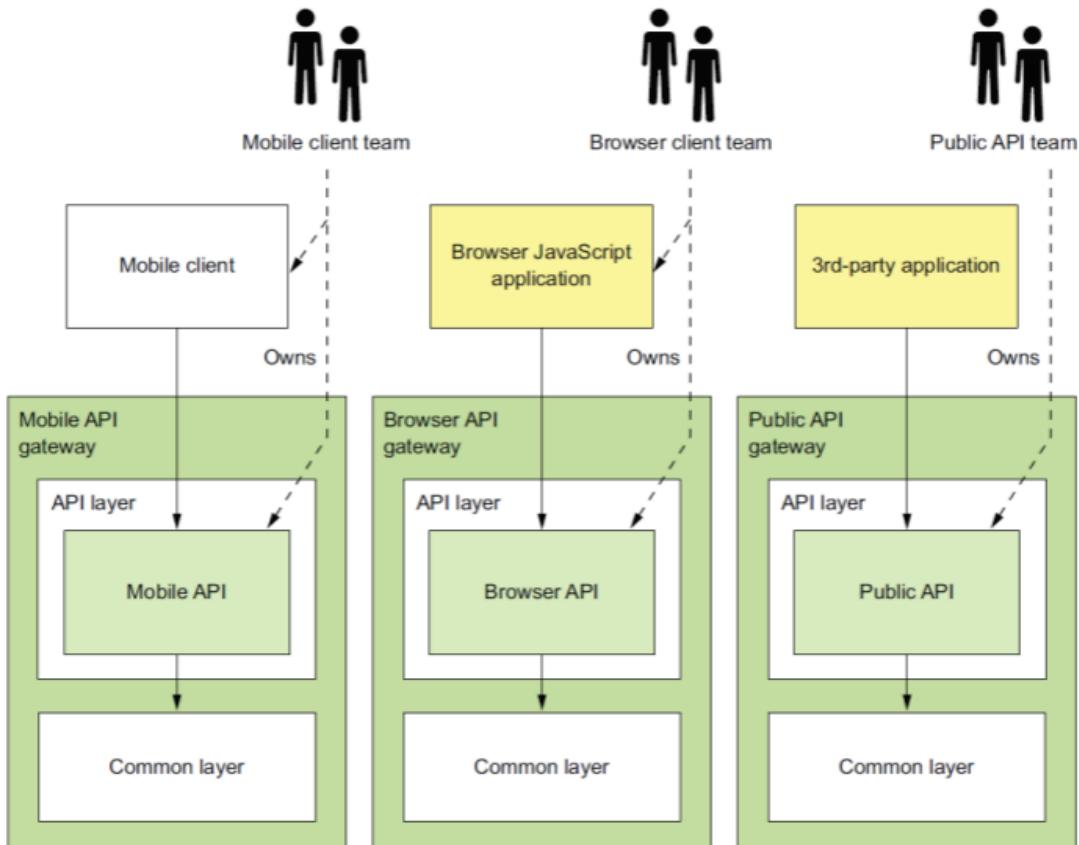
- Ogni team o unità può gestire il proprio Gateway
- I team possono adattare le configurazioni del Gateway in base alle necessità
- Maggiore resilienza, non c'è un unico point of failure
- Elimina tutti i possibili colli di bottiglia per la scalabilità
- Richiede una pipeline di deployment per gli aggiornamenti
- Più complesso da gestire
- Potenziali problemi di consistenza



## Backends for Frontends

- Ogni client ha il suo API Gateway
- Ogni **Gateway** è di proprietà di un **team**, che lo **mantiene**
- Ogni **modulo API** è **isolato**, previene i guasti a cascata

- **Scalabilità orizzontale** in base ai bisogni dei client
- **Complessità molto bassa**, ogni gateway fa poche cose
- Le librerie per funzioni comuni sono condivise (per l'autenticazione, per il logging, ...)



## Performance e scalabilità

Il Gateway API deve **gestire** tutte le **richieste API esterne** quindi le sue **performance** sono **fondamentali**, tutto si basa sulla sua **scalabilità**, se scala poco è un **collo di bottiglia** per l'intera applicazione.

Ci sono due approcci per gestire I/O:

- **Sincrono:**
  - **Facile** da programmare
  - **Limitato dall'overhead** di molteplici richieste
- **Asincrono:**
  - **Event loop single-thread** che gestisce tutte le connessioni
  - **Alta scalabilità**
  - **Ideale** per task **I/O-intensive**
  - **Non ideale** per task **Cpu-intensive**

## Considerazioni su implementazione

### Con Reactive Programming

L'obiettivo è avere dei **buoni tempi di risposta** e questo possiamo farlo utilizzando i cardini del **reactive programming**, ovvero **invocare parallelamente tutti i servizi** che ci servono.

Questo, oltre la **riduzione della latenza** ci permette di evitare il "callback hell" grazie all'utilizzo dello stile dichiarativo

NB.: Il callback hell è il caso in cui annidiamo tante callback rendendo il codice illegibile e difficile da manutenere. L'approccio preferibile è quello di usare dei gruppi di promise.

### Con guasti parziali

Alcune richieste potrebbero **fallire** o affrontare **latenze troppo elevate**, le cause potrebbero essere legate alle **risorse esaurite**. In questo caso è necessario implementare il pattern **Circuit Breaker** e gestire i casi di fallimento

### Integrazione con l'architettura

Il Gateway si deve far carico di **identificare dinamicamente la posizione** dei servizi e implementare strumenti di **monitoring e logging** per permettere la diagnosi

## Implementazione vera e propria

### Soluzioni di AWS

- **AWS API Gateway**: REST API con indirizzamento, autenticazione e scalabilità
- **AWS Application Load Balancer**: per indirizzare HTTP, HTTPS, WebSocket e HTTP/2

### Soluzioni open source

- **Kong**
- **Traefik**

### Proprio API Gateway

- Consigliato lo sviluppo utilizzando un **framework** per minimizzare l'effort
- Le regole di indirizzamento dovrebbero essere semplici

## Implementazione con GraphQL

- **Tecnologia API basata sui grafi** che vede i dati server-side come dei **grafi di oggetti con campi e relazioni**
- Il client **chiede esattamente** quello che gli serve (specifica i campi in modo preciso)
- Basta **solo una query** per ottenere tutti i dati di cui si ha bisogno
- Il grafo definito nello schema GraphQL è **direttamente collegato al database** quindi **il recupero e l'aggregazione** viene fatta in modo molto **efficiente**
- Un **unico endpoint** soddisfa tutti i bisogni dei client

- Riduce la la **complessità** di **sviluppo** degli sviluppatori perché integra già delle **automatizzazioni** legate alla gestione delle **query**

## GraphQL

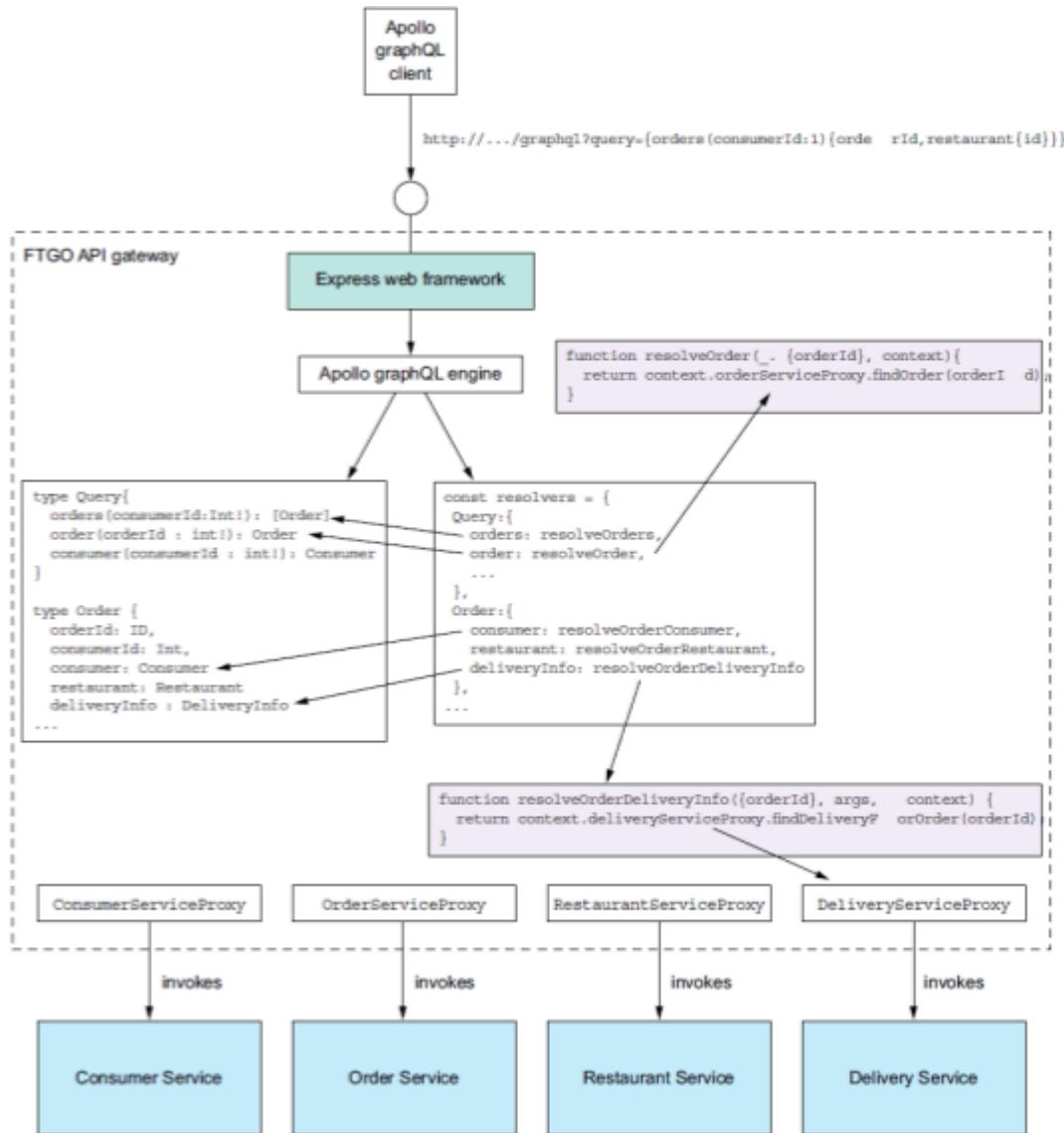
- Implementazione **open source** di Facebook
- **Supporta** un gran numero di **client**
- **Supporta query** (ottenimento dati) e **mutazioni** (creazione e aggiornamento dati)

## Falcor

- Implementazione **specificata, limitata** alla sua funzionalità e al suo ecosistema
- I client interagiscono con un'interfaccia JSON-like per ottenere o aggiornare le proprietà
- Il server mappa queste proprietà alle fonti di dati del backend

## Come funziona

- **Schema:** definisce il **modello** server-side dei dati e **specifica** le **query** supportate dall'API Gateway
- **Funzione Resolver:** gestisce le **richieste** che vengono fatte (con delle query), **interpreta** la **query** e capisce quali dati recuperare, li **recupera** (dal database) e li **aggredisce**
- **Classi proxy:** gestiscono l'**interazione** con i **servizi** di FTGO, permettono di invocare facilmente le funzioni resolver



## Fondamenti di Big Data

### Architettura Big Data

### Processing di Big Data offline