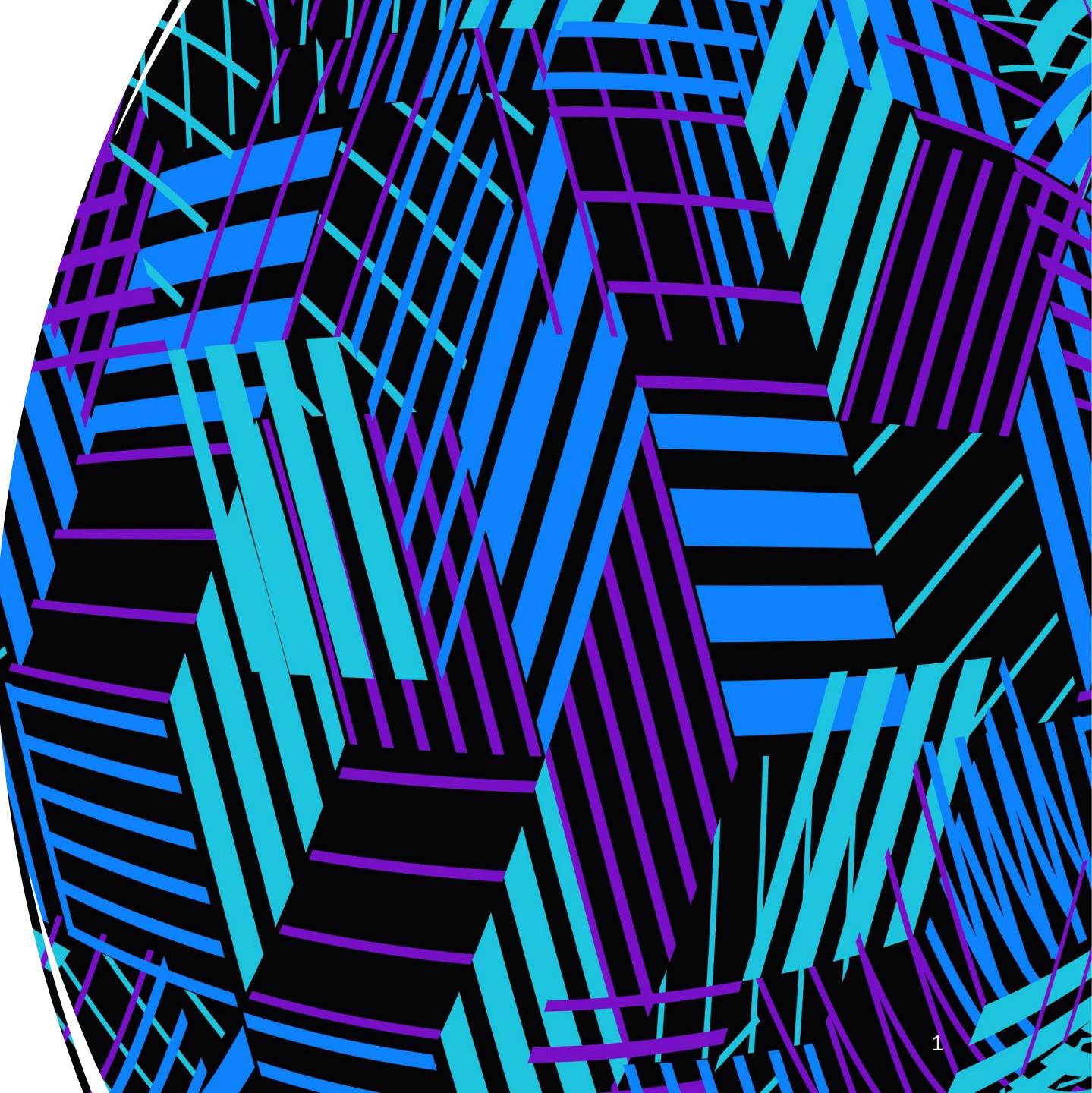


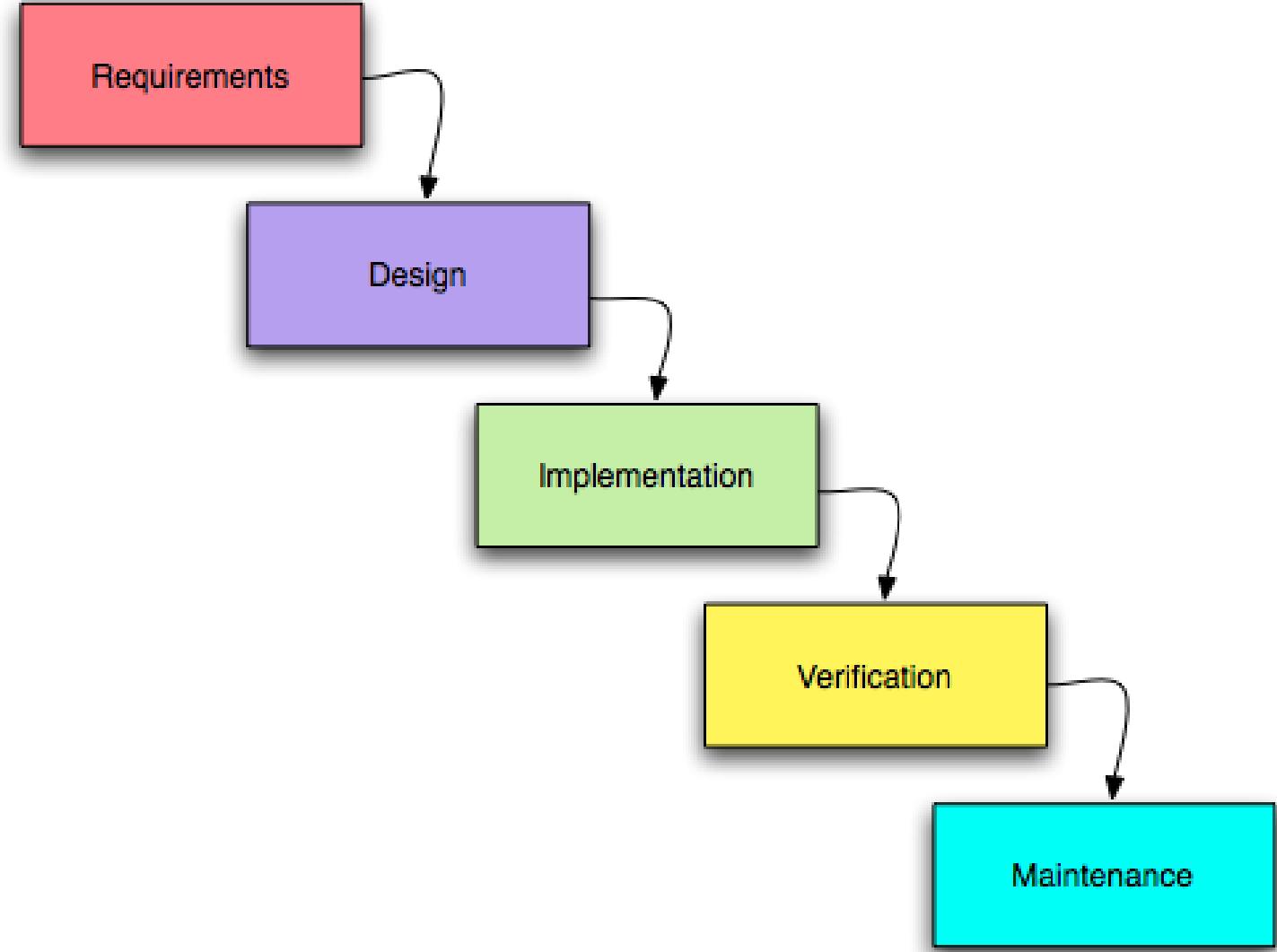
Introduzione ai Microservizi

Docente: ing.
Luca Lezzerini



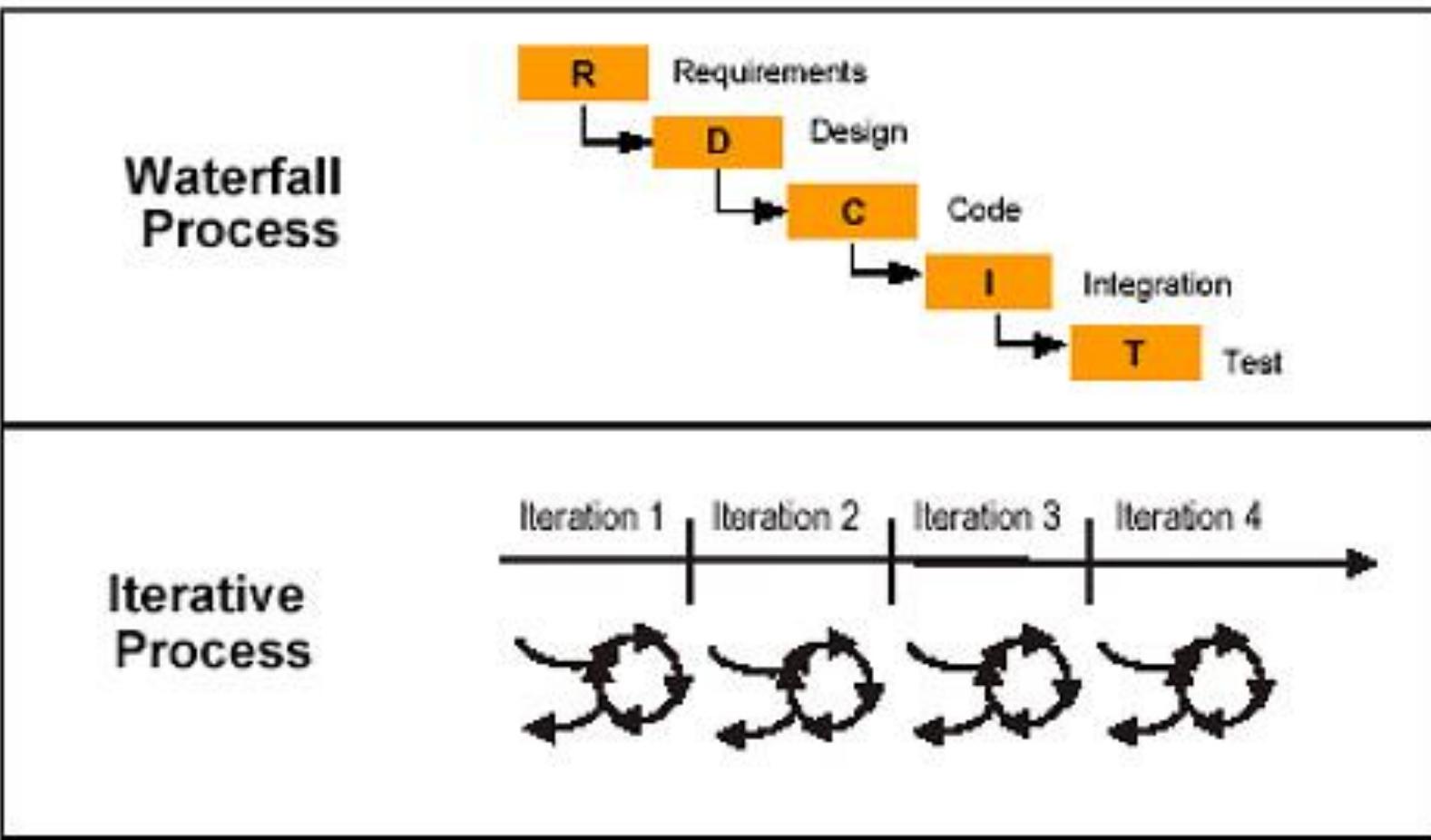
Cicli di sviluppo del software

Il Processo di Sviluppo a Cascata



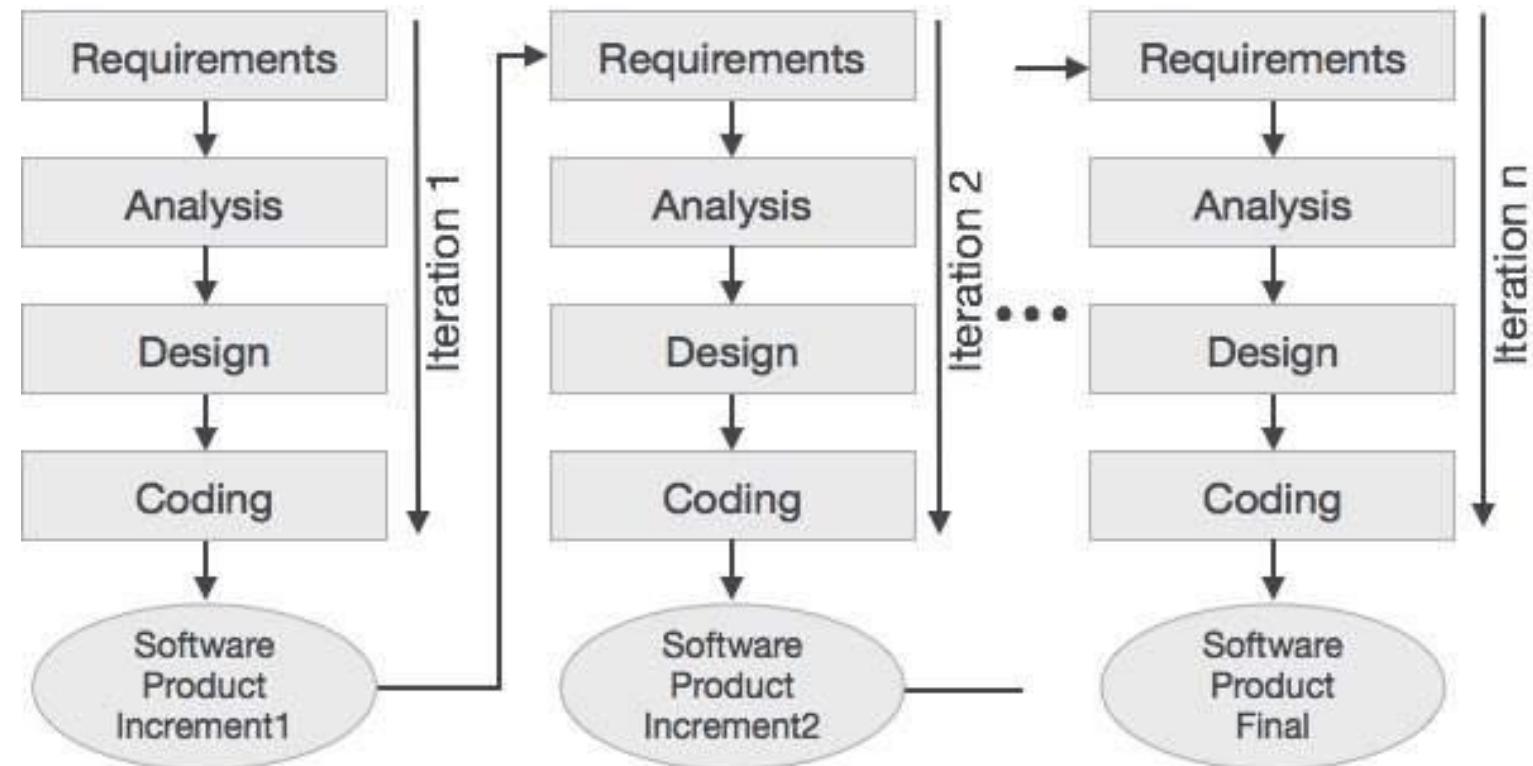
Struttura del Ciclo di Sviluppo a Cascata

- Il Ciclo di Sviluppo a Cascata, di norma, si suddivide nelle seguenti 4 fasi:
 - Analisi
 - Design
 - Codifica
 - Test



Processo Iterativo

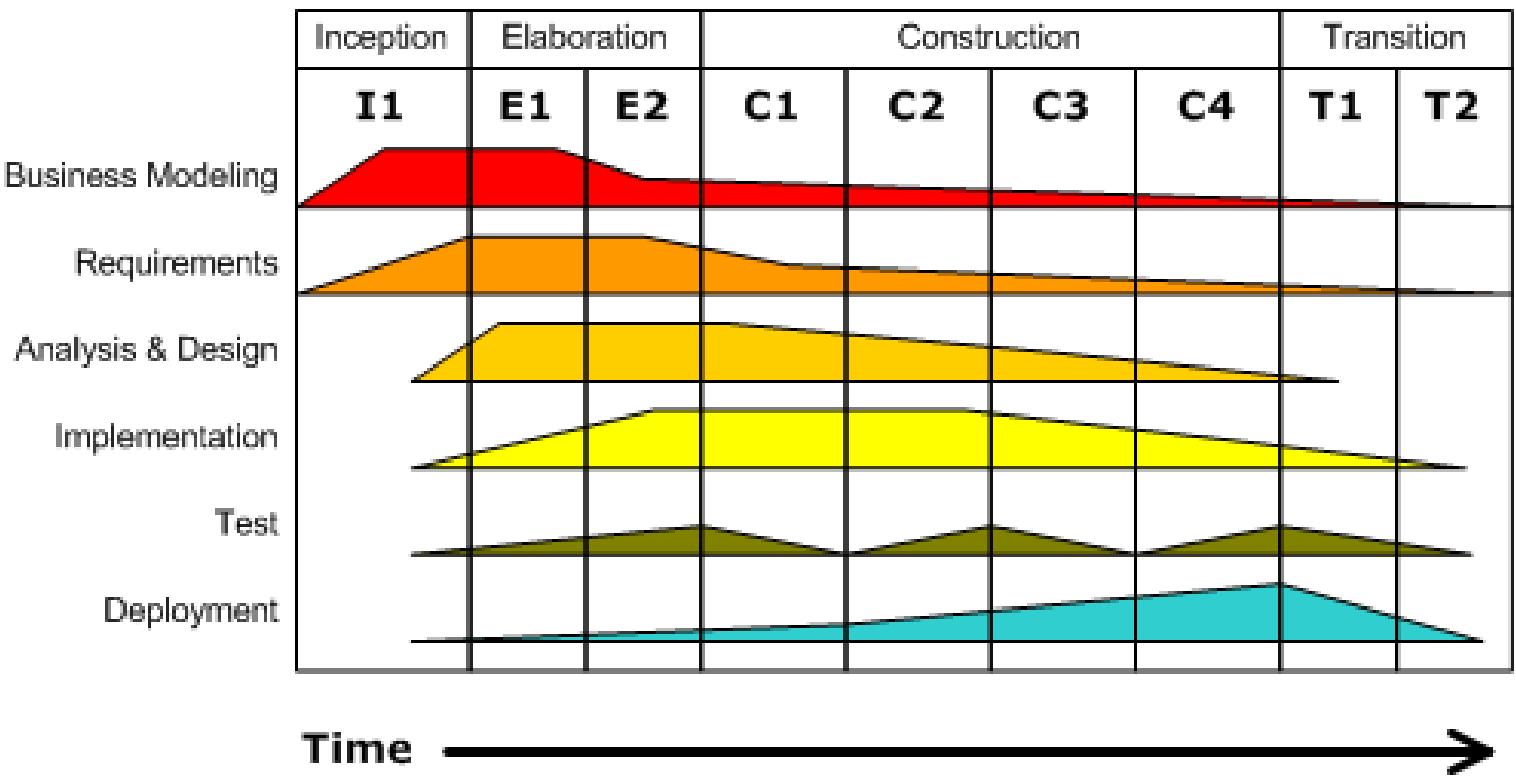
Processo Iterativo-Incrementale



Ciclo di Sviluppo Iterativo-Incrementale

Iterative Development

Business value is delivered incrementally in time-boxed cross-discipline iterations.



- Procedendo iterazione dopo iterazione non si ha mai una visione dettagliata dell'insieme del software. Ciò può portare all'**accumularsi di imperfezioni** nel progetto che, ad un certo punto, diventano incompatibili con lo sviluppo di una nuova iterazione.
- Soluzione: **iterazioni di refactoring**. Modificano il progetto ma non le funzionalità del software



La Somma delle
Tolleranze

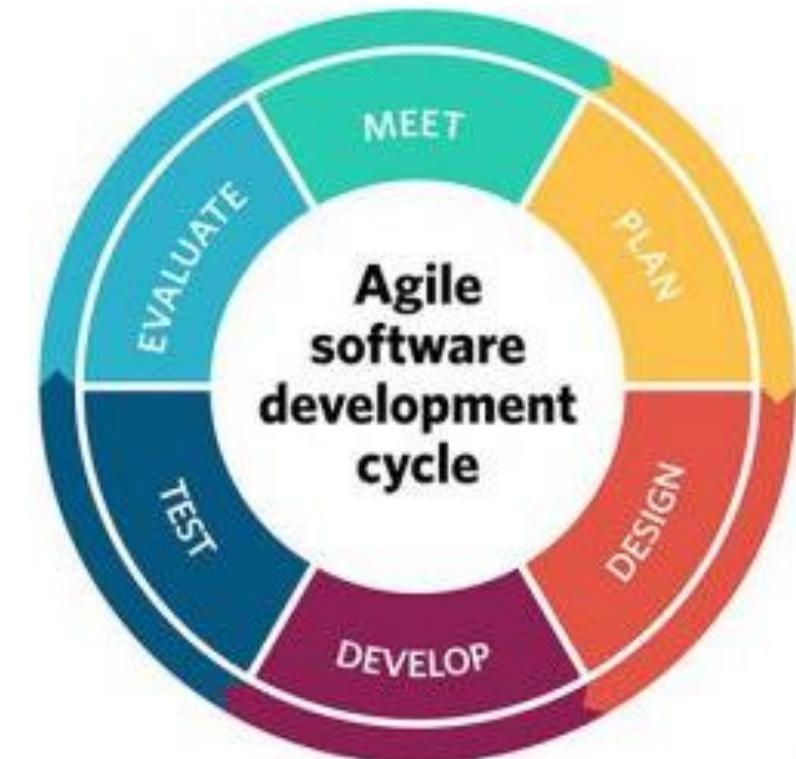
Metodologie agili/1

- Si basano su «sprint»
- Uno sprint è un periodo di tempo allocato per una particolare fase di un progetto. Gli sprint sono considerati completi quando scade il periodo di tempo. Potrebbero esserci disaccordi nel team se il livello di sviluppo sia accettabile o meno; in ogni caso, lo sviluppo termina per quella particolare fase del progetto. Le restanti fasi del progetto continueranno a evolvere entro i rispettivi intervalli di tempo.



Metodologie agili/1

- I principi generali delle metodologie Agile ...
 - Soddisfare il cliente e sviluppare continuamente il software.
 - Il cambiamento dei requisiti è accettato al fine di soddisfare maggiormente il Cliente.
 - Ci si concentra sulla consegna frequente e nel più breve tempo possibile.
 - Gli sviluppatori e gli esperti del business devono collaborare per l'intero progetto.
 - I progetti si basano su persone motivate.
 - La comunicazione faccia a faccia è il modo migliore per trasferire informazioni da e verso una squadra.
 - Il software funzionante (i.e. il grado di funzionamento) è la misura del reale progresso dello sviluppo.
 - Viene promosso un ritmo di sviluppo sostenibile.
 - Costante attenzione all'eccellenza tecnica e al buon design.
 - Semplicità (i.e. massimizzare il lavoro che non viene svolto).
 - Team auto-organizzati di solito creano i migliori design.
 - A intervalli regolari, il team rifletterà su come diventare più efficace e si adatterà e regolerà il proprio comportamento di conseguenza.





Manifesto per lo Sviluppo Agile di Software

Stiamo scoprendo modi migliori di creare software,
sviluppandolo e aiutando gli altri a fare lo stesso.

Grazie a questa attività siamo arrivati a considerare importanti:

Gli individui e le interazioni più che i processi e gli strumenti
Il software funzionante più che la documentazione esaustiva

La collaborazione col cliente più che la negoziazione dei contratti
Rispondere al cambiamento più che seguire un piano

Ovvero, fermo restando il valore delle voci a destra,
consideriamo più importanti le voci a sinistra.

Continuous Integration/1

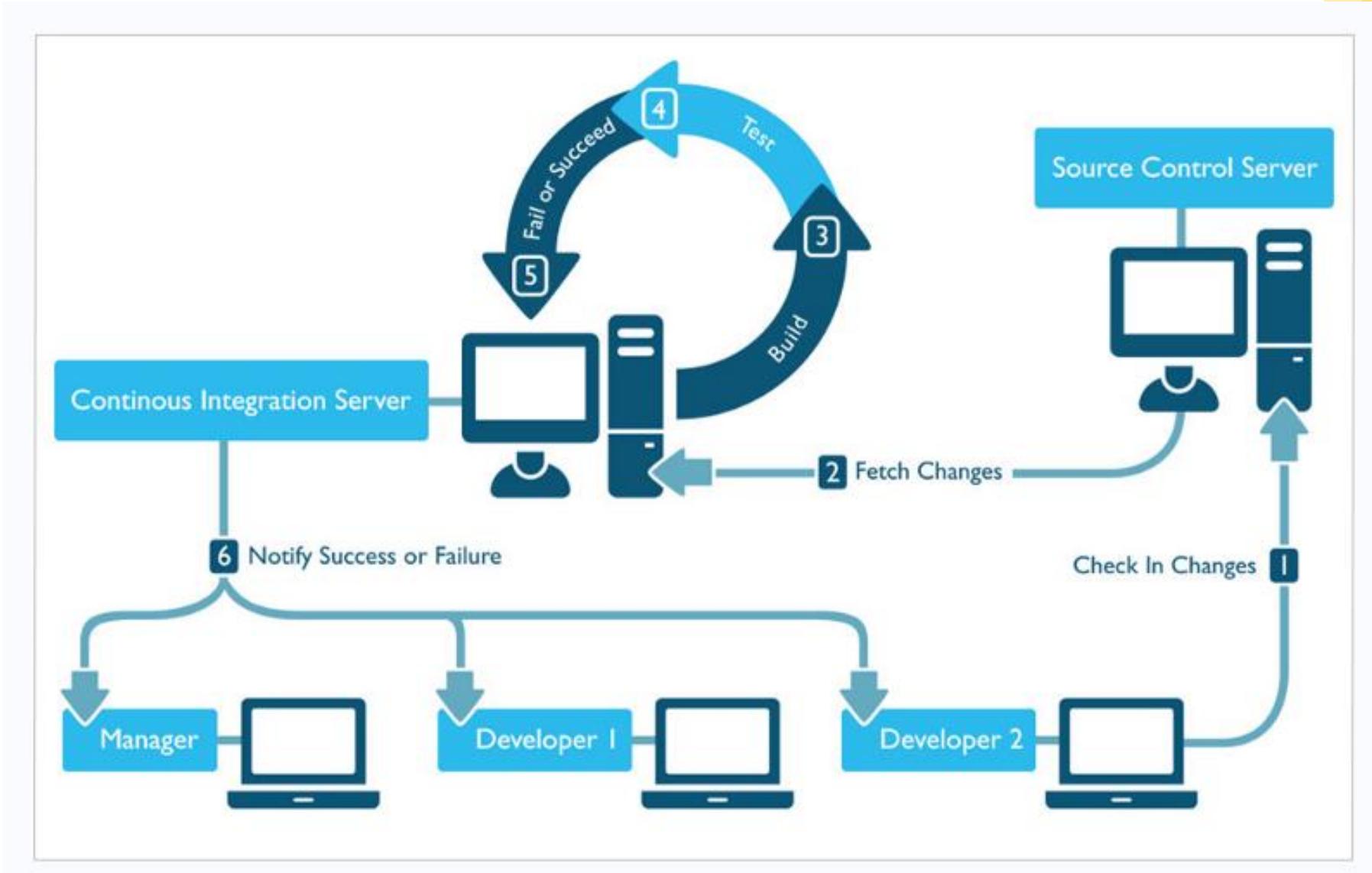
- La Continuous Integration (CI) è una pratica di sviluppo che richiede agli sviluppatori di integrare il codice in un repository condiviso più volte al giorno. Ogni check-in viene quindi verificato da un build automatizzato, consentendo ai team di rilevare i problemi in anticipo.
- Integrandosi regolarmente, è possibile rilevare rapidamente gli errori e individuarli più facilmente.
- Risolvi i problemi rapidamente
- Poiché ti stai integrando così frequentemente, c'è molto meno back-tracking per scoprire dove sono andate le cose, quindi puoi dedicare più tempo a sviluppare funzionalità.
- L'integrazione continua è economica. La mancata integrazione continua è costosa. Se non segui un approccio continuo, avrai periodi più lunghi tra le integrazioni. Questo rende esponenzialmente più difficile trovare e risolvere i problemi. Tali problemi di integrazione possono facilmente far fallire un progetto o far fallire del tutto.

Continuous Integration/2

- L'integrazione continua porta molteplici vantaggi:
 - Fine delle integrazioni lunghe e problematiche
 - Miglioramento della comunicazione
 - Anticipo dei problemi e loro risoluzione sul nascere
 - Meno tempo al debug e più tempo al reale sviluppo
 - Creare e mantenere una solida base su cui sviluppare
 - Conoscenza continua del livello di funzionamento del codice
 - Consegnare del software più veloce

Continuous Integration/3

- Best practice
 - Mantenere un repository centrale
 - Automatizzare i build
 - Fare test automatici ad ogni build
 - Ogni commit scatena un build su una macchina di integrazione
 - Build veloci
 - Testare in un clone dell'ambiente di produzione
 - Chiunque può ottenere l'ultima versione eseguibile
 - Tutti possono vedere cosa sta succedendo
 - Automatizzare il deployment



Continuous Integration/4

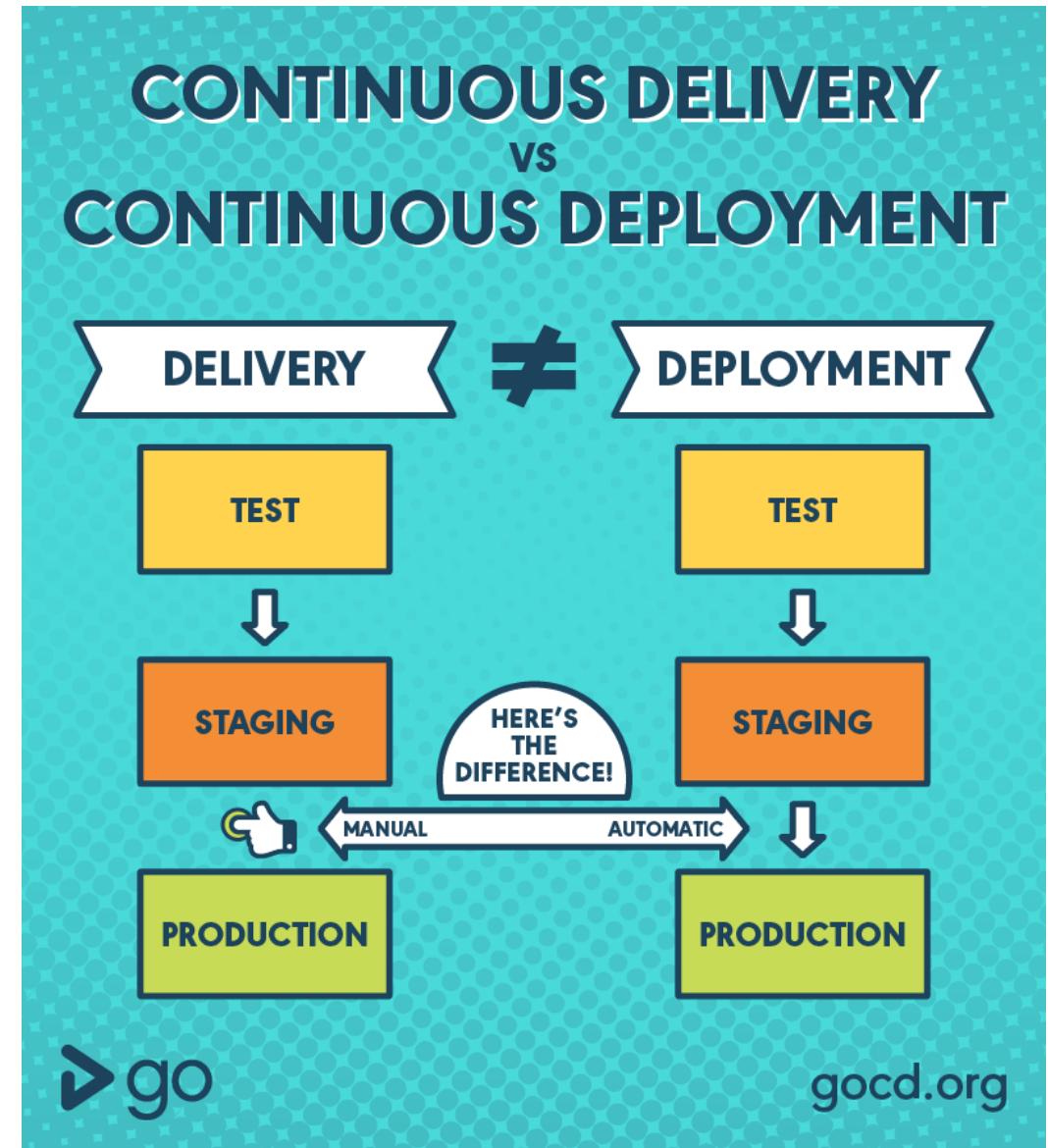
- Come realizzarle
 - Gli sviluppatori lavorano e testano il codice sulle loro copie locali
 - Al termine, committano le modifiche sul repository
 - Il server CI controlla il repository e rileva le modifiche quando si verificano
 - Il server CI esegue il build e lancia i test di unità e integrazione
 - Il server CI rilascia componenti deployabili (per il test)
 - Il server CI assegna un'etichetta di build ad ogni versione
 - Il server CI informa il team ad ogni build completato con successo
 - Se il build o i test falliscono, il server CI avvisa il team
 - Il team risolve il problema il prima possibile
 - Si continua ad integrare e testare continuamente per tutto il progetto

Continuous Integration/5

- Le responsabilità del team:
 - Check-in frequente
 - Non committare codice errato (i.e. non compila o non funziona palesemente)
 - Non eseguire il commit di codice non testato
 - Non effettuare il check-in quando il build precedente è fallito
 - Non andare a casa dopo il check-in fino a quando il build non ha successo

Continuous Integration/6

- La Distribuzione Continua
 - La Distribuzione continua è strettamente correlata all'integrazione continua e si riferisce al rilascio in produzione della versione che supera i test automatici.
 - "Essenzialmente, è la pratica di rilasciare agli utenti ogni build buono", (*Jez Humble, autore di Continuous Delivery*).
 - Adottando sia l'integrazione continua che la distribuzione continua, si riducono i rischi, si individuano rapidamente gli errori, si converge rapidamente su software funzionante.
 - Con rilasci a basso rischio, è possibile adattarsi rapidamente ai requisiti aziendali e alle esigenze degli utenti.



A silhouette photograph of a construction site against a vibrant orange and yellow sunset sky. In the foreground, several workers are silhouetted against the light, working on a complex steel scaffolding structure. A large construction crane is positioned on the left, its arm reaching diagonally across the frame. The overall atmosphere is one of industrial activity and architectural development.

Il paradigma di sviluppo a microservizi

- <https://microservices.io/patterns/microservices.html>

Cosa sono i microservizi

- Noti anche come architettura a microservizi, sono uno stile architettonico che struttura un'applicazione vedendola come una raccolta di servizi con le caratteristiche seguenti:
 - Altamente mantenibili e testabili
 - Debolmente accoppiati
 - Deployabili in modo indipendente
 - Organizzati in base alle possibilità/necessità aziendali
- L'architettura a microservizi consente la consegna / distribuzione continua di applicazioni grandi e complesse. Consente inoltre a un'organizzazione di far evolvere il proprio stack tecnologico con semplicità e in modo controllato.

Sono la soluzione migliore?

- I microservizi **non sono** la soluzione di ogni problema
- L'architettura a microservizi ha diversi contro. Inoltre, quando si utilizza questa architettura, ci sono numerose questioni che è necessario affrontare e risolvere.
- L'uso di pattern specifici ha due obiettivi:
 - decidere se i microservizi sono adatti al proprio caso
 - utilizzare correttamente l'architettura a microservizi

Preistoria (?): il pattern Monolitico/1

- **Contesto**
- Si sta sviluppando un'applicazione enterprise sul lato server. Deve supportare una varietà di client diversi tra cui browser desktop, browser mobile e applicazioni mobili native.
L'applicazione potrebbe anche esporre un'API per terze parti. Potrebbe anche integrarsi con altre applicazioni tramite i servizi Web o un broker di messaggi. L'applicazione gestisce le richieste (richieste e messaggi HTTP) eseguendo la logica di business; accede a un database; scambia messaggi con altri sistemi; e restituisce una risposta HTML / JSON / XML.
- Esistono componenti logici corrispondenti a diverse aree funzionali dell'applicazione.



Preistoria (?): il pattern Monolitico/2

- **Problema**
- Qual è l'architettura di distribuzione dell'applicazione?

- **Driver**
- Esiste un gruppo di sviluppatori che lavorano sull'applicazione
- I nuovi membri del team devono diventare rapidamente produttivi
- L'applicazione deve essere facile da capire e modificare
- Si desidera praticare la distribuzione continua (continuous integration) dell'applicazione
- È necessario eseguire più copie dell'applicazione su più macchine per soddisfare i requisiti di scalabilità e disponibilità
- Si vuole approfittare delle tecnologie emergenti (framework, linguaggi di programmazione, ecc.) -> evoluzione rapida dello stack tecnologico



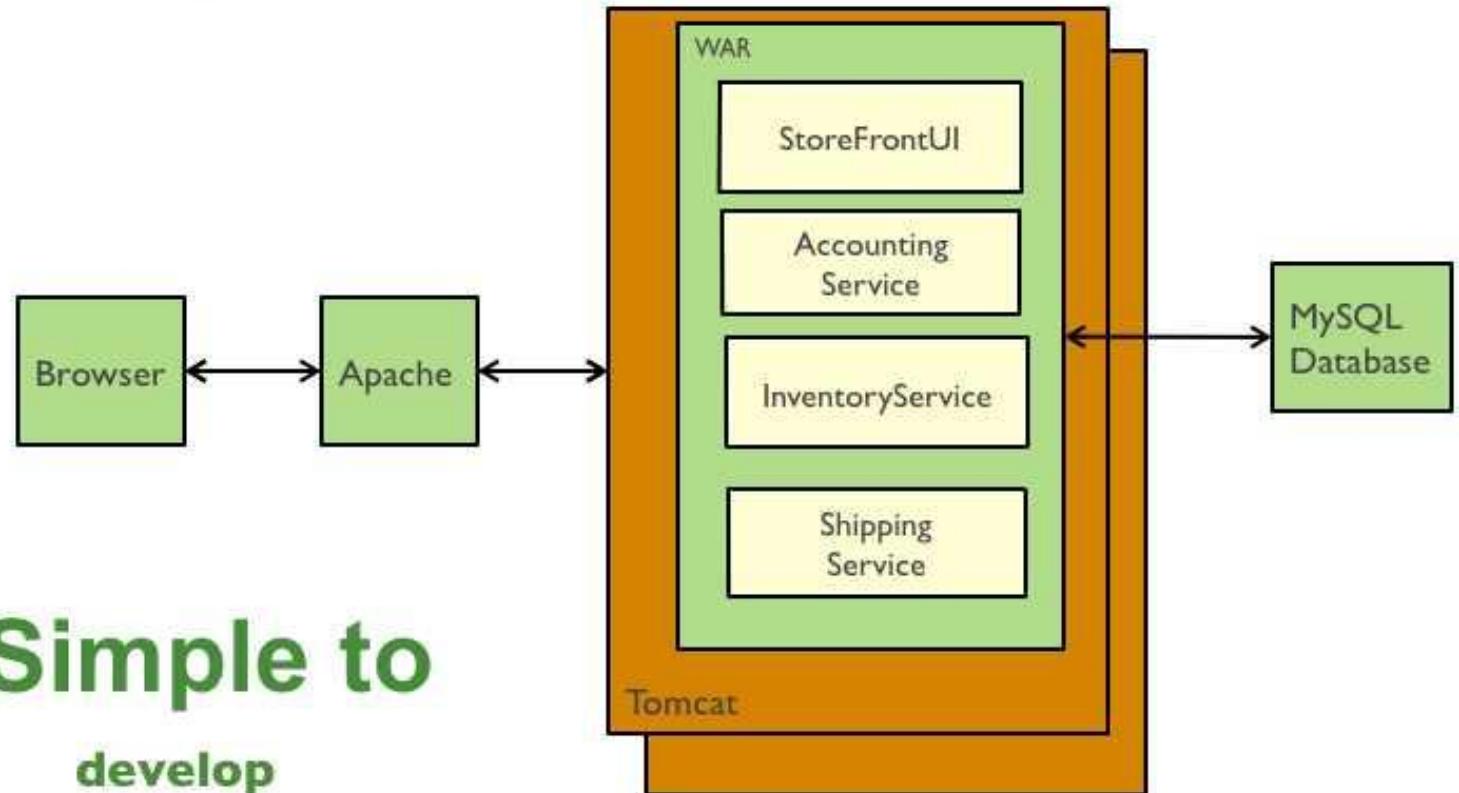
Preistoria (?): il pattern Monolitico/3

- **Soluzione**
- Costruire un'applicazione con un'architettura monolitica. Per esempio:
 - un singolo file Java WAR
 - una singola gerarchia di directory con il codice Angular



Esempio di Monolite

Traditional web application architecture





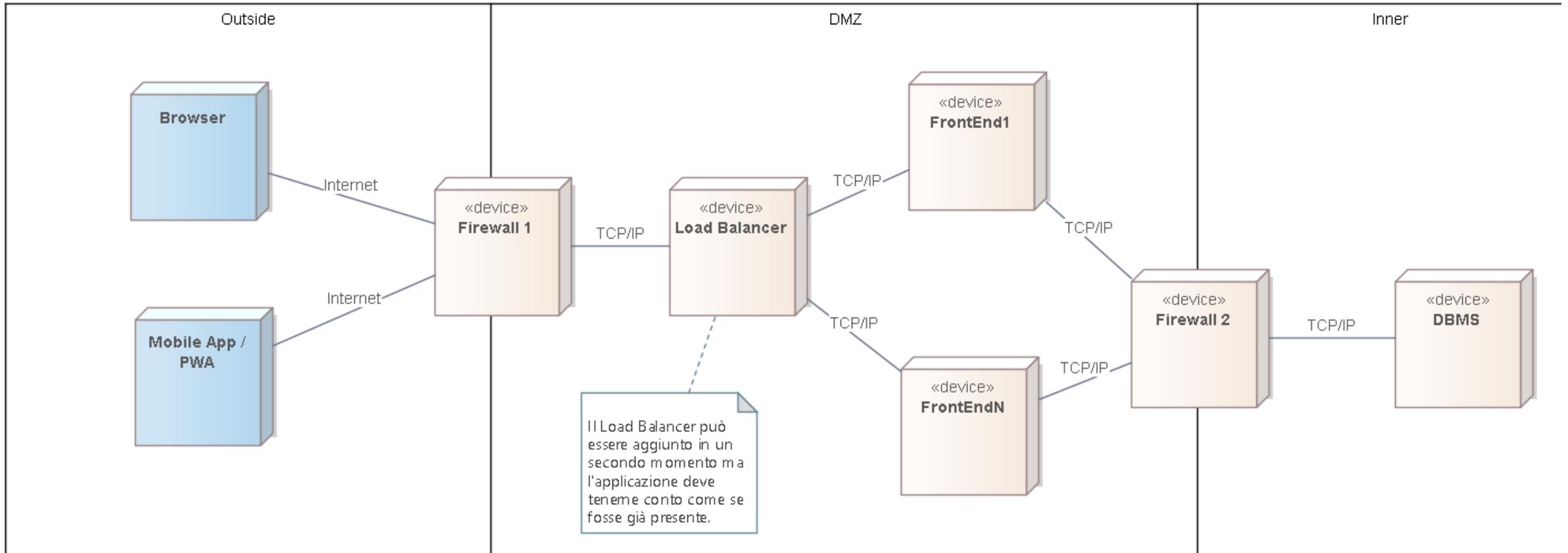
Vantaggi del monolite

- Semplice da sviluppare: gli attuali strumenti di sviluppo e IDE nascono per lo sviluppo di applicazioni monolitiche
- Semplice da implementare: è sufficiente distribuire il file WAR (o la gerarchia di directory) sul runtime appropriato
- Semplice da scalare: è possibile scalare l'applicazione con un servizio di bilanciamento del carico

Svantaggi del monolite/1

- Container (Web o application) sovraccaricato: più grande è l'applicazione, più tempo è necessario per l'avvio. Comporta un impatto sulla produttività degli sviluppatori a causa del tempo sprecato in attesa che il sistema si avvii. Idem per la distribuzione.
- La distribuzione continua è difficile: una grossa applicazione monolitica è anche un ostacolo alle frequenti installazioni. Per aggiornare un componente bisogna ridistribuire l'intera applicazione. Ciò interromperà le attività in background, indipendentemente dal fatto che siano interessate dalla modifica o che possano o meno causare problemi. C'è anche la possibilità che i componenti che non sono stati aggiornati non si avvino correttamente. Di conseguenza, aumenta il rischio associato alla ridistribuzione, scoraggiando gli aggiornamenti frequenti. Questo è un problema soprattutto per gli sviluppatori di interfacce utente, dato che di solito hanno bisogno di ridistribuire frequentemente.



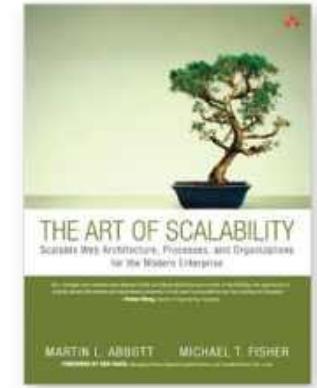
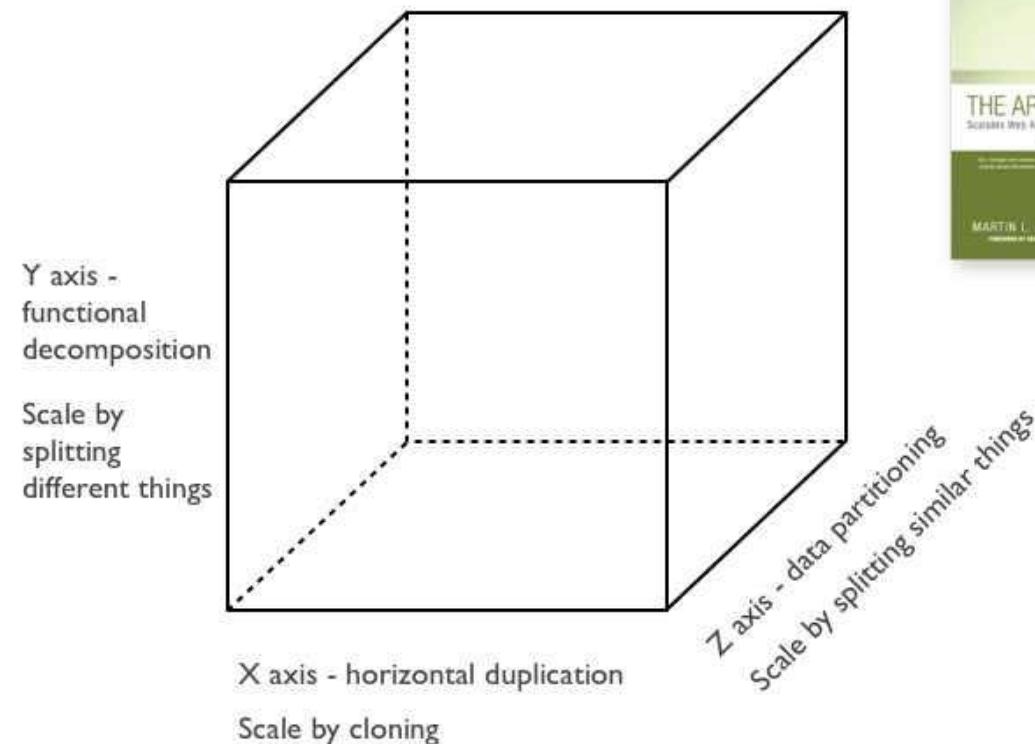


Scalabilità

Cubo di scalabilità

- Le dimensioni secondo cui scalare un'applicazione sono tre e vanno a costituire il **Cubo di Scalabilità o Scale Cube**

3 dimensions to scaling



Architettura a Microservizi

- **Contesto**
 - Si sta sviluppando un'applicazione enterprise sul lato server. Deve supportare una varietà di client diversi tra cui browser desktop, browser mobili e applicazioni mobili native. L'applicazione potrebbe anche esporre un'API per terze parti da consumare. Potrebbe anche integrarsi con altre applicazioni tramite i servizi Web o un broker di messaggi. L'applicazione gestisce le richieste (richieste e messaggi HTTP) eseguendo la logica di business; accedere a un database; scambiare messaggi con altri sistemi; e restituire una risposta HTML / JSON / XML.
 - Esistono componenti logici corrispondenti a diverse aree funzionali dell'applicazione.
-
- **Problema**
 - Qual è l'architettura di distribuzione dell'applicazione?

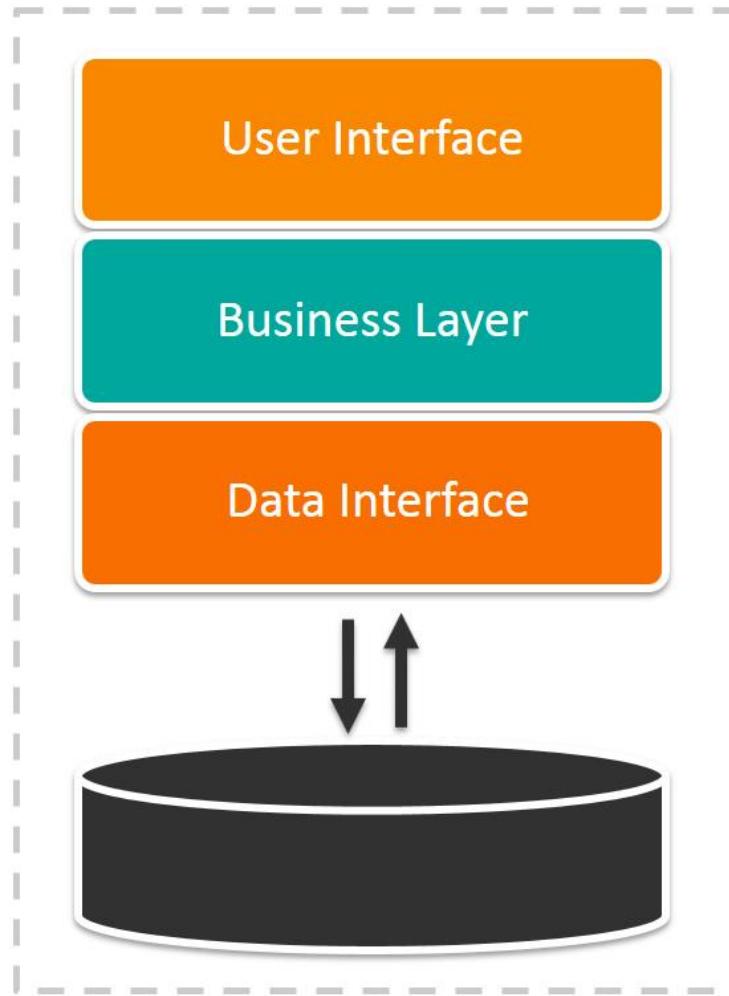
Architettura a Microservizi

- **Problema**
- Qual è l'architettura di distribuzione dell'applicazione?
- **Driver**
 - Team di sviluppatori che lavora sull'applicazione
 - I nuovi membri del team devono diventare rapidamente produttivi
 - L'applicazione deve essere facile da comprendere e modificare
 - Si vuole praticare il continuous deployment dell'applicazione
 - È necessario avere più copie dell'applicazione su più macchine per soddisfare i requisiti di scalabilità e disponibilità
 - Si vuole trarre il massimo vantaggio dalle tecnologie emergenti (framework, linguaggi di programmazione, ecc.) -> evoluzione rapida dello stack tecnologico

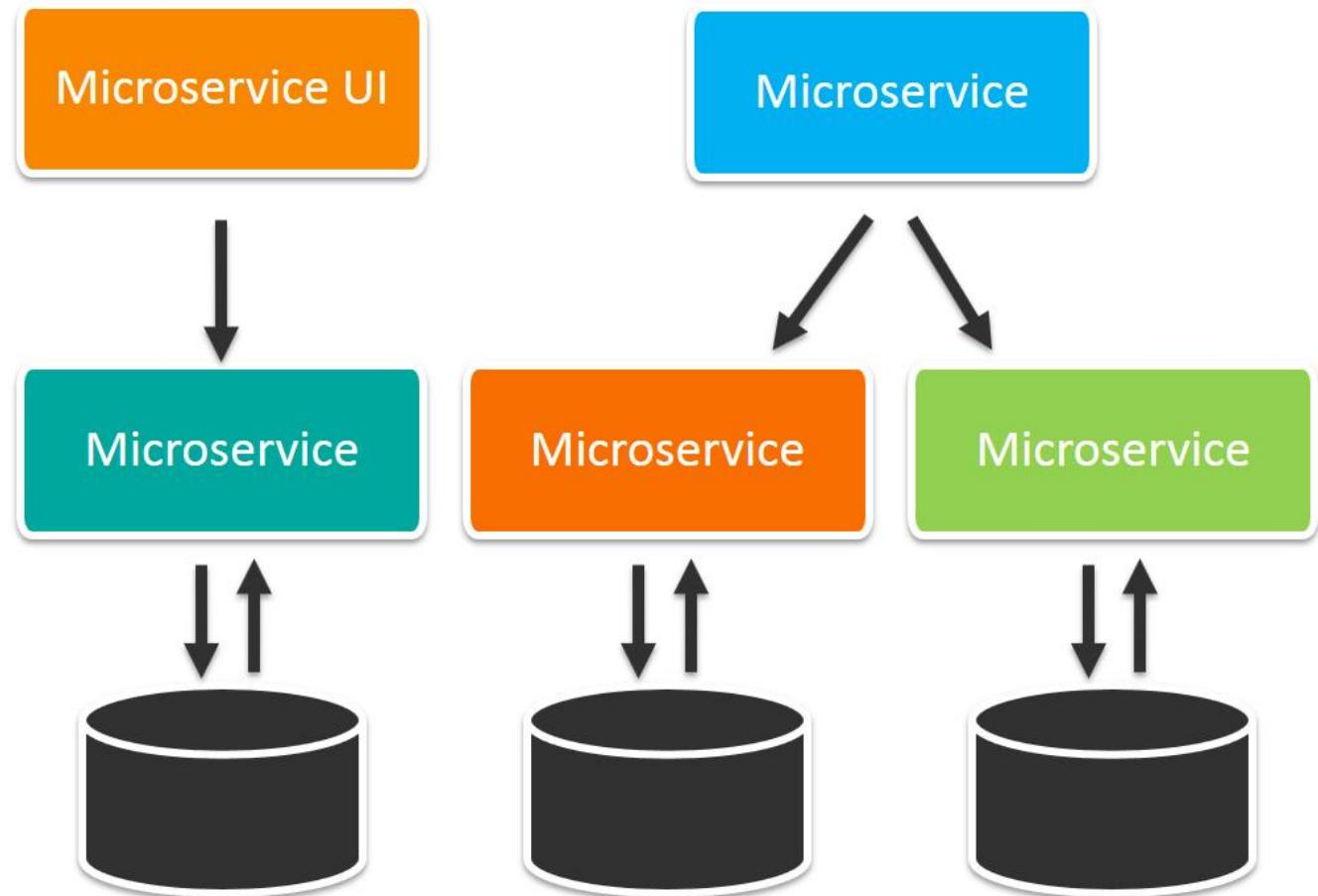
Architettura a Microservizi

- **Soluzione**
- Definire un'architettura che struttura l'applicazione come un insieme di servizi collaborativi liberamente accoppiati.
- Questo approccio corrisponde all'asse Y del **Cubo di Scalabilità**.
- Ogni servizio implementa una serie di funzioni strettamente correlate. Ad esempio, un'applicazione può consistere in servizi come il servizio di gestione degli ordini, il servizio di gestione dei clienti, ecc.
- I servizi comunicano utilizzando protocolli sincroni come HTTP / REST o protocolli asincroni come AMQP. I servizi possono essere sviluppati e distribuiti indipendentemente l'uno dall'altro.
- Ogni servizio ha il proprio database per essere disaccoppiato da altri servizi. La coerenza dei dati tra i servizi viene mantenuta utilizzando il pattern **Saga**

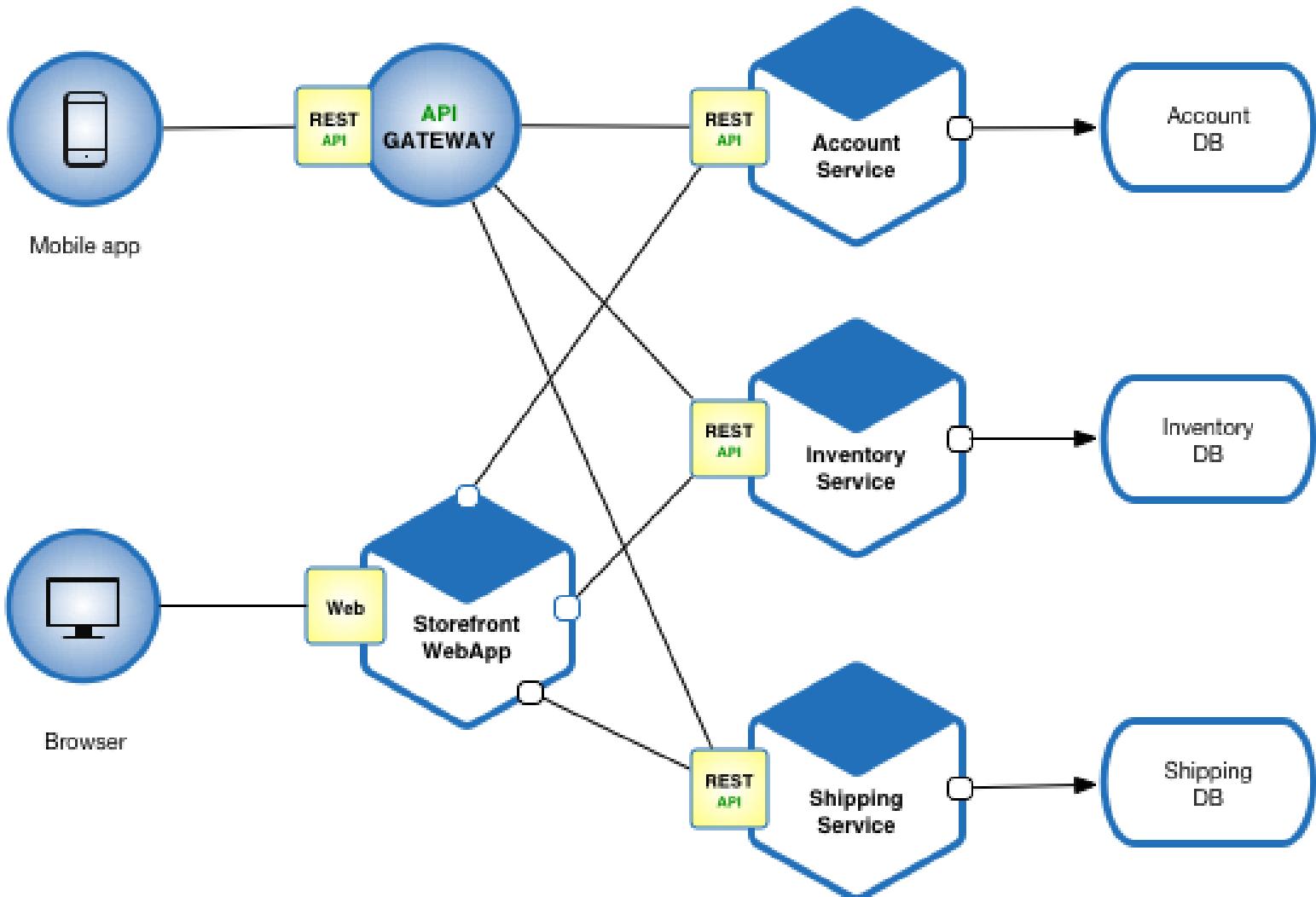
Monolithic Architecture



Microservices Architecture



Esempio di Microservizi



Vantaggi dei Microservizi/1

- Consentono la consegna e l'implementazione continue di applicazioni grandi e complesse.
- Migliore testabilità: i servizi sono più piccoli e più veloci da testare
- Migliore deployability: i servizi possono essere implementati e distribuiti in modo indipendente
- Consente di ripartire lo sforzo di sviluppo su più team autonomi. Ogni squadra è proprietaria ed è responsabile di uno o più singoli servizi. Ogni squadra può sviluppare, distribuire e scalare i propri servizi indipendentemente da tutti gli altri team.
- Ogni microservizio è relativamente piccolo e più facile da comprendere per uno sviluppatore

Vantaggi dei Microservizi/2

- L'IDE è più veloce e rende gli sviluppatori più produttivi
- L'applicazione si avvia più velocemente, il che rende gli sviluppatori più produttivi e accelera rilasci e run
- Migliore isolamento dei guasti. Ad esempio, se c'è una perdita di memoria in un servizio, solo quel servizio sarà interessato. Gli altri servizi continueranno a gestire le richieste. Al contrario, un componente anomalo di un'architettura monolitica può far cadere l'intero sistema.
- Elimina qualsiasi impegno a lungo termine su un preciso stack tecnologico. Quando si sviluppa un nuovo servizio si può scegliere un differente stack tecnologico. Allo stesso modo, quando si apportano modifiche importanti a un servizio esistente, è possibile riscriverlo utilizzando un altro stack tecnologico.

Svantaggi dei Microservizi

- Gli sviluppatori devono affrontare la complessità di un sistema distribuito.
- Gli strumenti di sviluppo / IDE sono orientati alla creazione di applicazioni monolitiche e non forniscono un supporto esplicito per lo sviluppo di applicazioni distribuite.
- Il test è più difficile
- Gli sviluppatori devono implementare il meccanismo di comunicazione tra i servizi.

Quando usare i microservizi?/1

- Decidere quando ha senso utilizzarlo.
- In primis, spesso, quando si sviluppa la prima versione di un'applicazione, non si hanno i problemi che questo approccio risolve.
- Inoltre, l'utilizzo di un'architettura elaborata e distribuita rallenterà lo sviluppo. Questo può essere un grosso problema per le start-up o per progetti critici la cui più grande sfida è spesso il modo di evolvere rapidamente il modello di business / l'applicazione.
- L'utilizzo delle divisioni dell'asse Y del Cubo di Scalabilità potrebbe rendere molto più difficile l'iterazione veloce.
- ***È quindi un problema decidere in fase di avvio se usare o meno i microservizi***

Quando usare i microservizi?/2

- Se si parte con un monolite, la questione diviene come ridimensionare/ridisegnare l'applicazione per portarla su microservizi.
- In tal caso è necessario utilizzare la scomposizione funzionale: le dipendenze intricate potrebbero rendere difficile la decomposizione dell'applicazione monolitica in un insieme di servizi.

Come decomporre un monolite?/1

- Decomporre per funzionalità aziendale (***business capability***) e definire servizi corrispondenti alle funzionalità aziendali.
- Decomporre in modo ***domain-driven***.
- Decomporre in base al verbo o al caso d'uso e definire i servizi responsabili di determinate azioni. Per esempio: un servizio di spedizione responsabile della spedizione degli ordini completi.
- Decomporre in base a nomi o risorse definendo un servizio responsabile di tutte le operazioni su entità / risorse di un determinato tipo. Per esempio: un servizio gestione clienti responsabile della gestione dell'archivio clienti.

Come decomporre un monolite?/2

- Teoricamente ogni servizio dovrebbe avere solo un piccolo insieme di responsabilità. Spesso si parla della progettazione di classi utilizzando il Principio di Responsabilità Singola (SRP). Ha senso applicare l'SRP anche alla progettazione dei servizi.
- Un'analogia che aiuta nel comprendere la logica della progettazione per microservizi è la progettazione delle utility Unix. Unix fornisce un gran numero di utilità come grep, cat e find. Ogni utility fa esattamente una cosa, spesso eccezionalmente bene, e può essere combinata con altre utility usando uno script di shell per eseguire compiti complessi.

Come mantenere la coerenza dei dati?

- Per garantire un accoppiamento lasco, ogni servizio ha il proprio database.
- Mantenere la coerenza dei dati tra i servizi è una sfida perché transazioni distribuite a due fasi non sono banali per molte applicazioni. Un'applicazione a microservizi dovrebbe utilizzare il pattern Saga. Un servizio pubblica un evento quando i suoi dati cambiano. Altri servizi consumano quell'evento e aggiornano i loro dati.
- Esistono diversi modi per aggiornare in modo affidabile i dati e pubblicare eventi tra cui i pattern Event Sourcing e Transaction Log Tailing.

Come implementare le query?

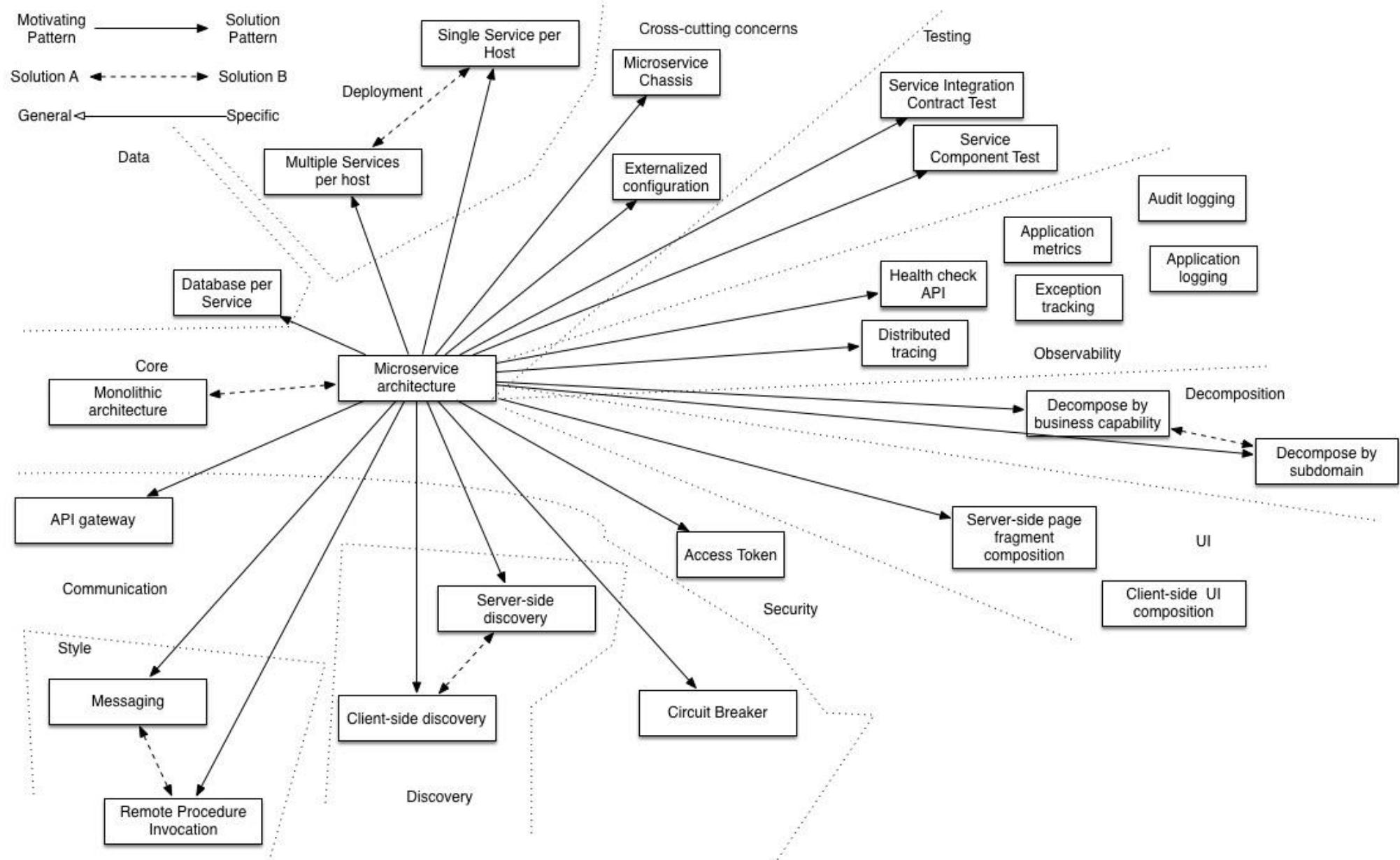
- Un'altra questione è l'implementazione di query che devono recuperare i dati di proprietà di più servizi.
- I pattern di **API Composition** e di **Command Query Responsibility Segregation (CQRS)** sono due tecniche molto utilizzate.

Principi di progettazione

Tecniche di decomposizione

- <https://microservices.io/patterns/decomposition/decompose-by-business-capability.html>
- <https://microservices.io/patterns/decomposition/decompose-by-subdomain.html>

Pattern



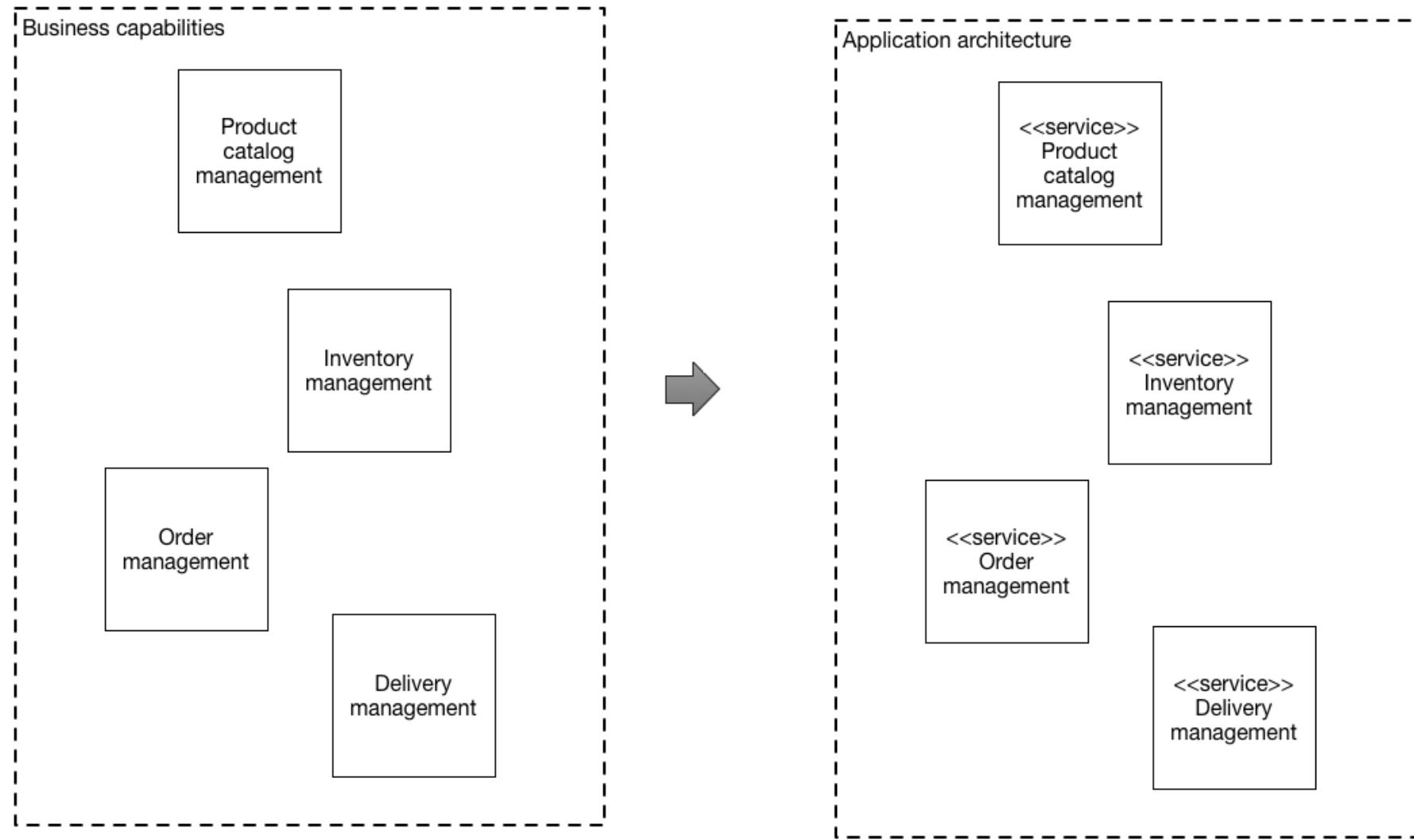
Decomposizione per Business Capability/1

- **Problema**
- Come decomporre un'applicazione in microservizi?
- **Driver**
- L'architettura deve essere stabile
- I servizi devono essere coerenti. Un servizio dovrebbe implementare un piccolo insieme di funzioni fortemente correlate.
- I servizi devono essere conformi al **Common Closure Principle** (ciò che cambia insieme deve essere raggruppato insieme)
- I servizi devono essere liberamente accoppiati: ogni servizio è un'API che ne incapsula l'implementazione.
L'implementazione deve poter essere modificata senza influire sui client
- Ogni servizio deve essere testabile
- Ogni servizio deve essere abbastanza piccolo da essere sviluppato da un team di "due pizze", cioè da una squadra di 6-10 persone
- Ogni squadra che possiede uno o più servizi deve essere autonoma.
- Ogni team deve essere in grado di sviluppare e distribuire i propri servizi con la minima collaborazione con altri team.

Decomposizione per Business Capability/2

- **Soluzione**
- Definire servizi corrispondenti alle capability aziendali. Una capacità aziendale è un concetto derivato dalla modellazione dell'architettura aziendale. È qualcosa che un'azienda fa per generare valore. Una funzionalità aziendale spesso corrisponde a un oggetto aziendale, ad esempio:
 - L'ufficio gestione degli ordini è responsabile per gli ordini
 - L'ufficio clienti è responsabile per la gestione dei clienti
- Le capacità aziendali sono spesso organizzate in una gerarchia multilivello. Ad esempio, un'applicazione aziendale potrebbe avere categorie di livello superiore come lo sviluppo di prodotti / servizi, la consegna di prodotti / servizi, la generazione di richieste, ecc.

Decomposizione per Business Capability/3



Vantaggi della decomposizione per capability

- Questo modello ha i seguenti vantaggi:
 - Architettura stabile poiché le capability aziendali sono relativamente stabili
 - I team di sviluppo sono interfunzionali, autonomi e organizzati intorno a fornire valore aziendale piuttosto che funzioni tecniche
 - I servizi sono coesi e liberamente accoppiati
- **Problemi**
- Ci sono i seguenti problemi da affrontare:
 - Come identificare le capacità aziendali? Identificare le capacità aziendali e quindi i servizi richiede una comprensione del business. Le capacità aziendali di un'organizzazione vengono identificate analizzando lo scopo, la struttura, i processi aziendali e le aree di competenza dell'organizzazione.

Svantaggi della decomposizione per capability

- I vari contesti sono spesso identificati utilizzando un processo iterativo.
- Buoni punti di partenza per l'identificazione delle capacità di business sono:
 - struttura organizzativa: diversi gruppi all'interno di un'organizzazione potrebbero corrispondere a capacità aziendali o gruppi di capacità aziendali.
 - modello di dominio di alto livello: le funzionalità aziendali spesso corrispondono agli oggetti del dominio

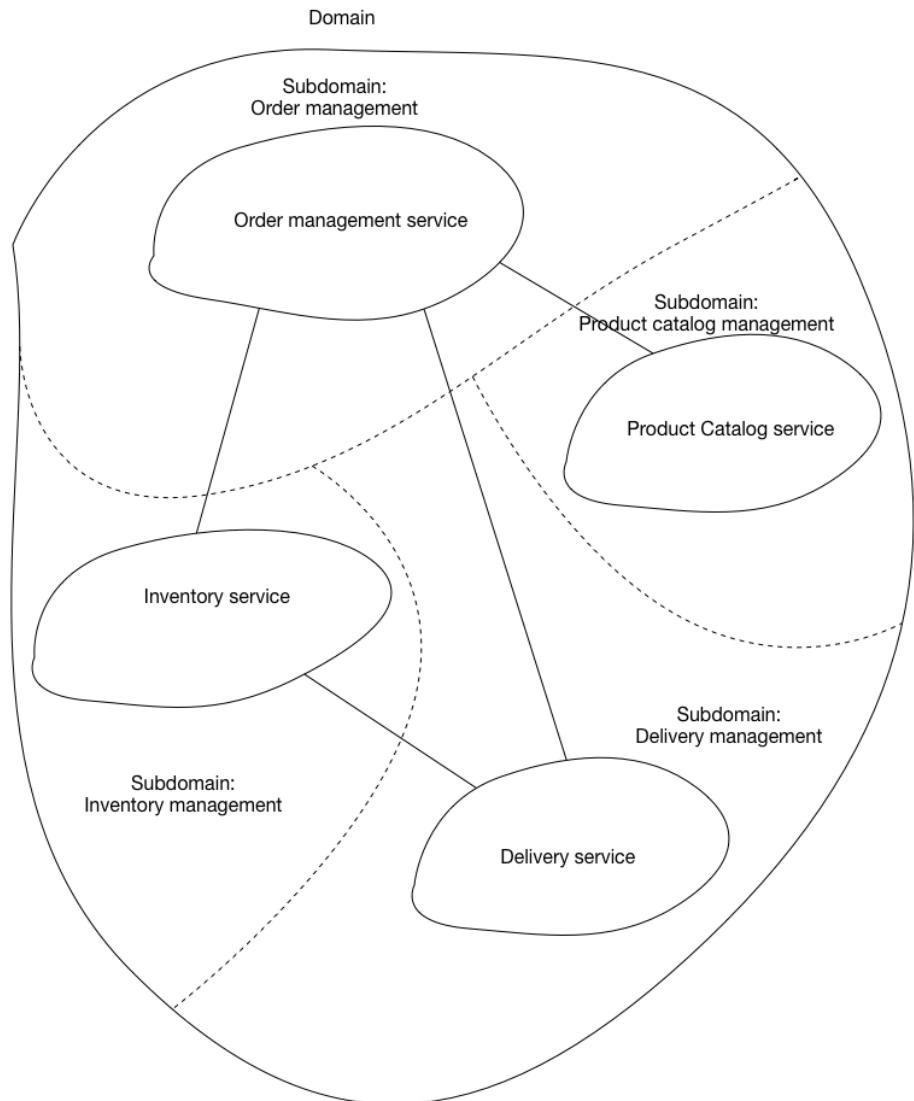
Decomposizione per Subdomain Context/1

- **Problema**
- Come decomporre un'applicazione in microservizi?
- **Driver**
- L'architettura deve essere stabile
- I servizi devono essere coerenti. Un servizio dovrebbe implementare un piccolo insieme di funzioni strettamente correlate.
- I servizi devono essere conformi al **Common Closure Principle** (ciò che cambia insieme deve essere raggruppato insieme)
- I servizi devono essere liberamente accoppiati: ogni servizio è un'API che ne incapsula l'implementazione.
L'implementazione deve poter essere modificata senza influire sui client
- Ogni servizio deve essere testabile
- Ogni servizio deve essere abbastanza piccolo da essere sviluppato da un team di "due pizze", cioè da una squadra di 6-10 persone
- Ogni squadra che possiede uno o più servizi deve essere autonoma.
- Ogni team deve essere in grado di sviluppare e distribuire i propri servizi con la minima collaborazione con altri team.

Decomposizione per Subdomain Context/2

- **Soluzione**
- Definire i servizi corrispondenti ai sottodomini ricavati con il Domain-Driven Design (DDD). DDD fa riferimento allo spazio dei problemi dell'applicazione - l'azienda - come dominio. Un dominio è costituito da più sottodomini. Ogni sottodominio corrisponde a una parte diversa dell'attività.
- I sottodomini possono essere classificati come segue:
 - **Core:** chiave di differenziazione per il business e la parte più preziosa dell'applicazione
 - **Supporto:** relativo a ciò che l'azienda fa ma non la differenzia. Questi possono essere implementati internamente o esternalizzati.
 - **Generico:** non specifico per il business e idealmente implementato con software off the shelf

Decomposizione per Subdomain Context/3



Per maggiori dettagli sul DDD si veda:

<http://dddcommunity.org/>

Vantaggi della decomposizione DDD/4

- Architettura stabile poiché i sottodomini sono relativamente stabili
- I team di sviluppo sono interfunzionali, autonomi e focalizzati per fornire valore aziendale piuttosto che funzioni tecniche
- I servizi sono coesi e liberamente accoppiati

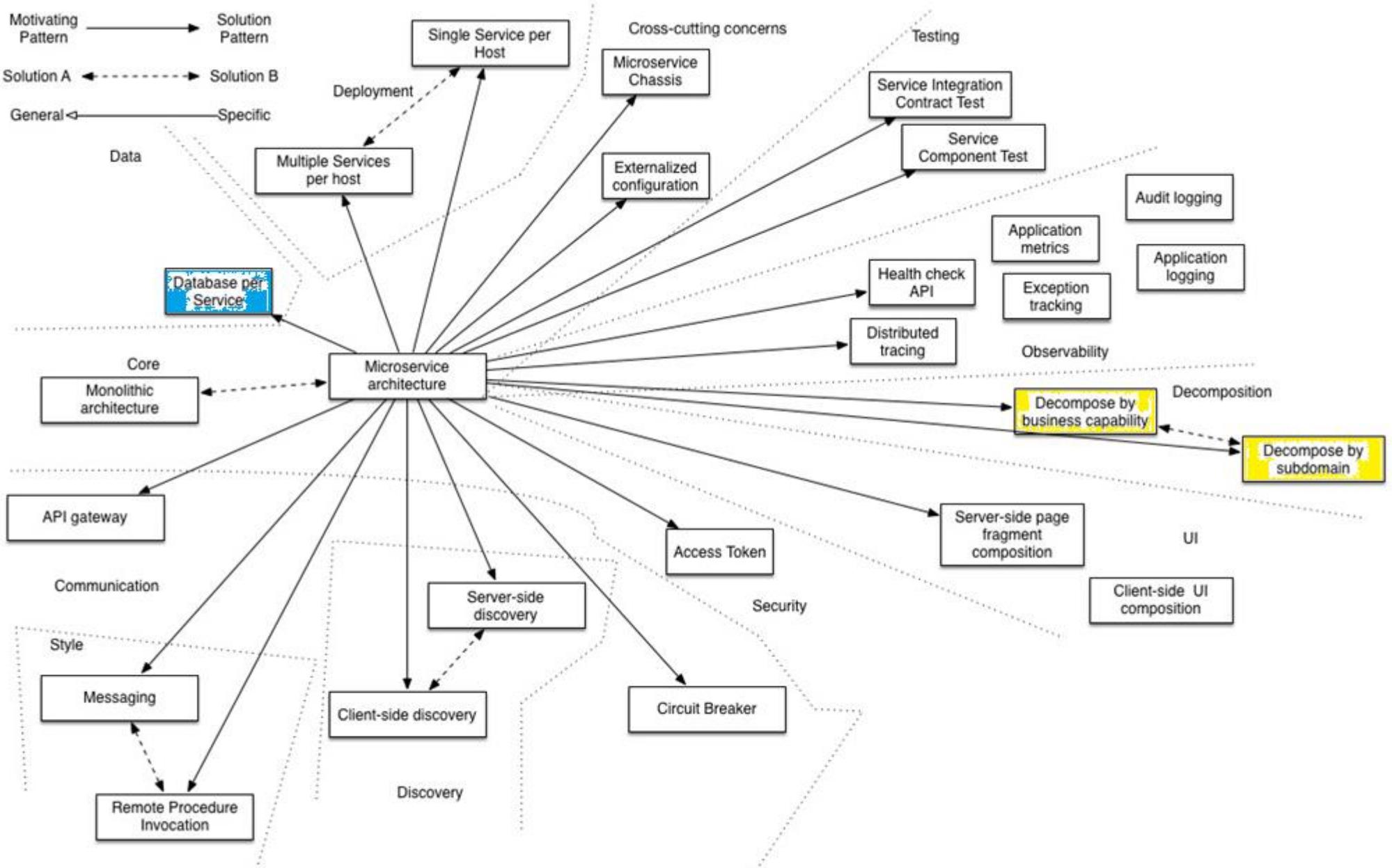
Svantaggi della decomposizione DDD/5

- Come identificare i sottodomini? Identificare sottodomini e quindi i servizi richiede una comprensione del business. Come le capacità aziendali, i sottodomini vengono identificati analizzando il business e la sua struttura organizzativa e identificando le diverse aree di competenza. I sottodomini sono meglio identificati utilizzando un processo iterativo.
- Buonipunti di partenza per l'identificazione dei sottodomini sono:
 - **la struttura organizzativa:** diversi gruppi all'interno di un'organizzazione potrebbero corrispondere a sottodomini
 - **il modello di dominio di alto livello:** i sottodomini hanno spesso un oggetto dominio chiave

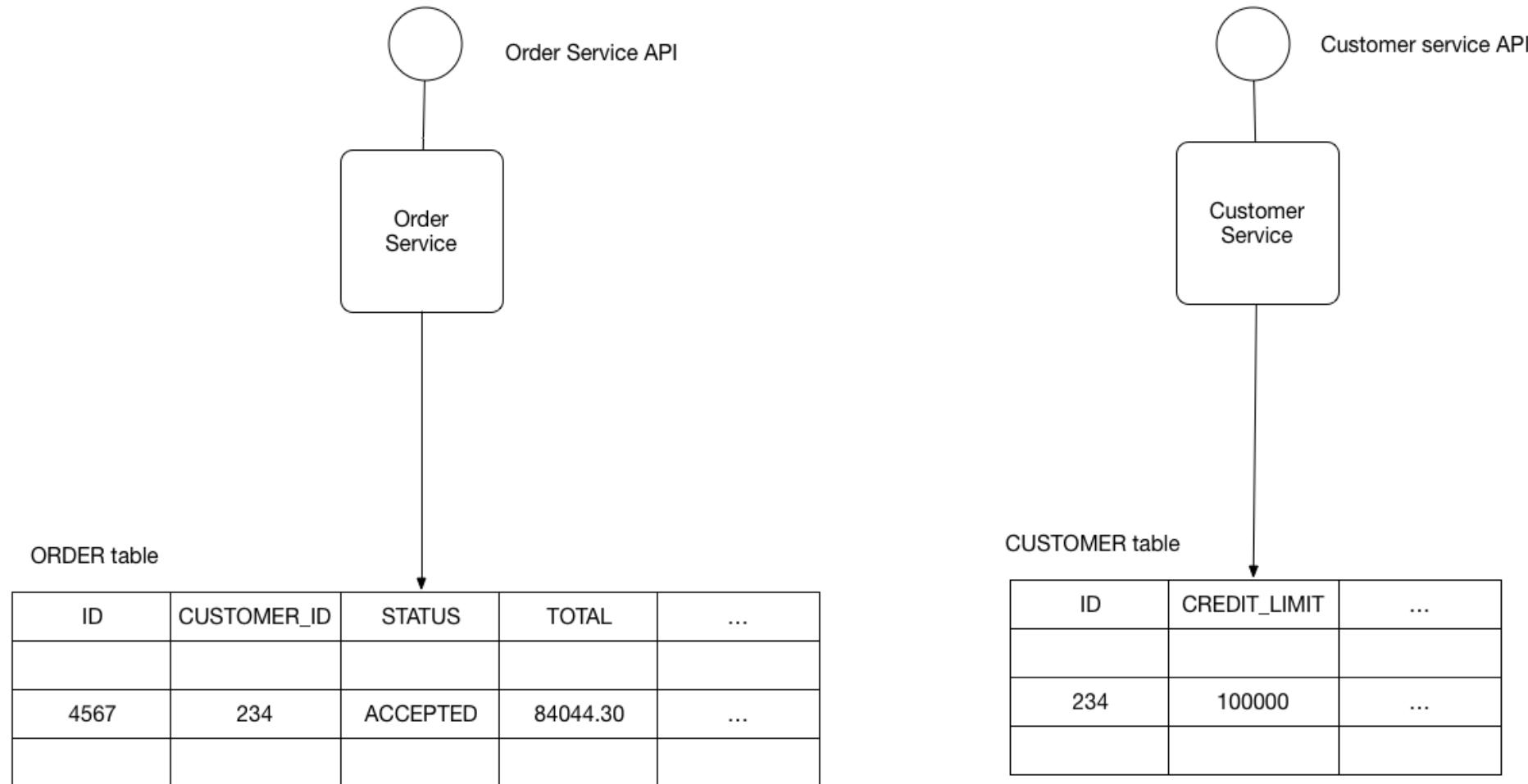
In poche parole ...

- Decomporre per business capability significa **dividere in una serie di passi che rappresentano come il business lavora.**
- Decomporre per subdomain context significa **dividere in base ai flussi informativi.**

| Pattern



Pattern: Database per service



Pattern: Database per service

- **Problema**
- Qual è l'architettura del database in un'applicazione a microservizi?
- **Anti-pattern**
- **Shared database**

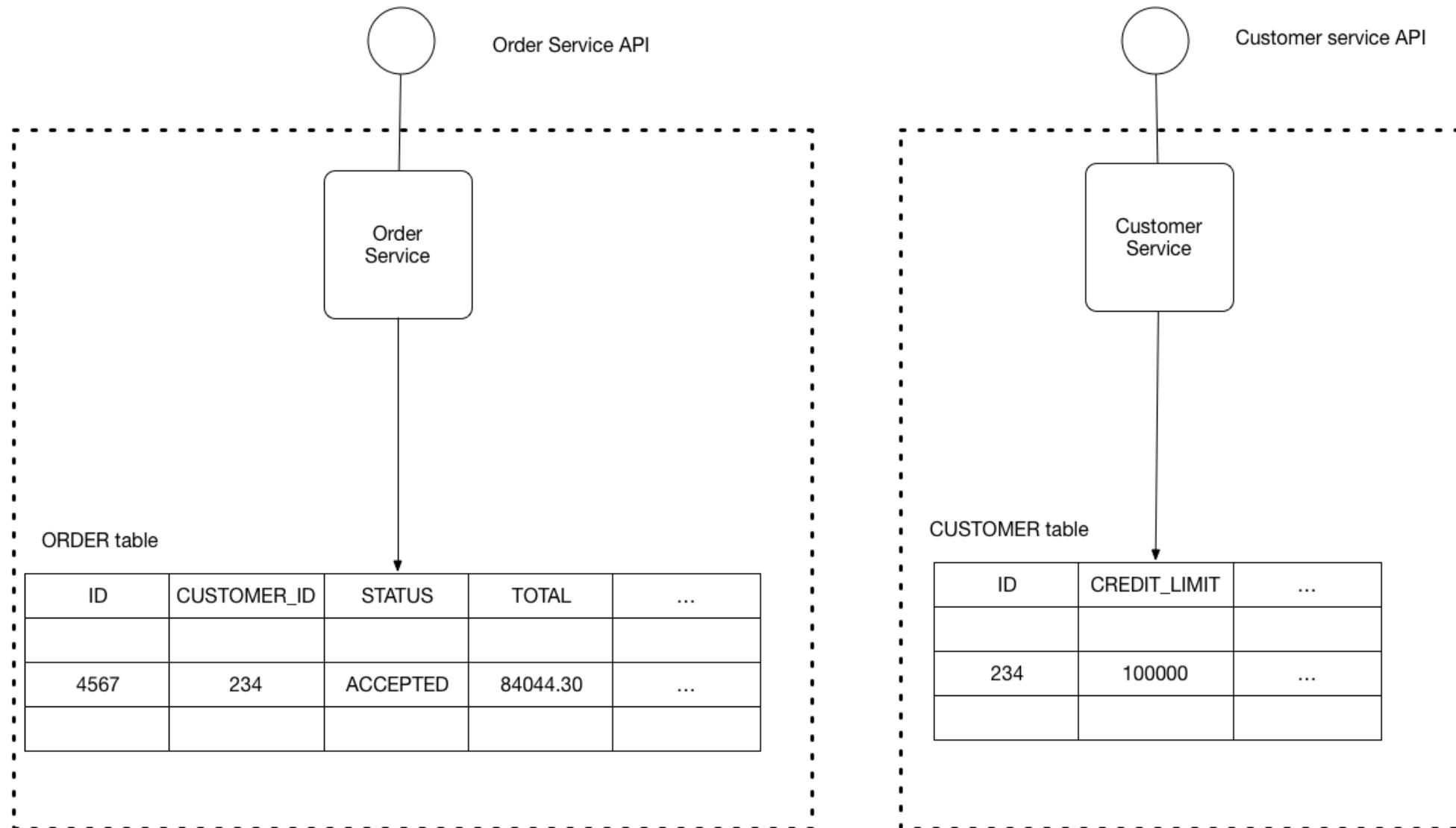
Pattern: Database per service

- **Driver**
- I servizi devono essere liberamente accoppiati in modo che possano essere sviluppati, distribuiti e ridimensionati in modo indipendente
- Alcune transazioni commerciali devono applicare invarianti che si estendono su più servizi. Ad esempio, il caso d'uso Conferma Ordine deve verificare che un nuovo ordine non superi il limite di credito del cliente. Altre transazioni commerciali devono aggiornare i dati di proprietà di più servizi.
- Alcune transazioni commerciali devono interrogare i dati di proprietà di più servizi. Ad esempio, l'utilizzo Visualizza credito disponibile deve interrogare il cliente per trovare il limite di credito e gli ordini per calcolare l'importo totale degli ordini aperti.
- Alcune query devono unire i dati che appartengono a più servizi. Ad esempio, per trovare clienti in una particolare regione e i loro ordini recenti è necessario unire clienti e ordini.
- A volte i DB devono essere replicati e condivisi per poter essere scalati. (Cubo di Scalabilità)
- Servizi diversi hanno requisiti di archiviazione dei dati diversi. Per alcuni servizi, un database relazionale è la scelta migliore. Altri servizi potrebbero necessitare di un database NoSQL come MongoDB, che sia valido per la memorizzazione di dati complessi e non strutturati, o Neo4J, progettato per archiviare e interrogare in modo efficiente i dati di un grafo.

Pattern: Database per service

- **Soluzione**
- Mantenere privati i dati permanenti di ciascun microservizio su quel servizio e accedervi solo tramite la sua API.
- Le transazioni di un servizio riguardano solo il suo database.

Pattern: Database per service



Pattern: Database per service

- Il database del servizio fa effettivamente parte dell'implementazione di quel servizio. Non è possibile accedervi direttamente da altri servizi.
- Esistono diversi modi per mantenere privati i dati permanenti di un servizio. Non è necessario eseguire il provisioning di un server database per ciascun servizio. Ad esempio, se si utilizza un database relazionale, le opzioni sono:
 - Private-tables-per-service: ogni servizio possiede un set di tabelle a cui può accedere solo quel servizio
 - Schema per servizio: ogni servizio ha uno schema di database privato per quel servizio
 - Database-server-per-service: ogni servizio ha il proprio server di database.
- Private-tables-per-service e schema-per-service hanno il sovraccarico più basso. L'utilizzo di uno schema per servizio è attraente dal momento che rende più chiara la proprietà. Alcuni servizi ad alto throughput potrebbero necessitare del proprio server di database.
- È una buona idea creare barriere che facciano rispettare questa modularità. Ad esempio, è possibile assegnare un ID utente di database diverso a ciascun servizio e utilizzare un meccanismo di controllo dell'accesso al database a privilegi. Senza una sorta di barriera per rafforzare l'incapsulamento, gli sviluppatori saranno sempre tentati di bypassare l'API di un servizio e accedere direttamente ai dati.

Pattern: Database per service

- Vantaggi:
- Aiuta a garantire che i servizi siano accoppiati liberamente. Le modifiche al database di un servizio non hanno alcun impatto su altri servizi.
- Ogni servizio può utilizzare il tipo di database più adatto alle sue esigenze. Ad esempio, un servizio che esegue ricerche di testo potrebbe utilizzare ElasticSearch. Un servizio che manipola un grafo social potrebbe utilizzare Neo4j.

Pattern: Database per service

- Teorema CAP:
- In informatica teorica, il teorema CAP, noto anche come teorema di Brewer, afferma che è impossibile per un sistema informatico distribuito fornire simultaneamente tutte e tre le seguenti garanzie:
 - Coerenza (tutti i nodi vedano gli stessi dati nello stesso momento)
 - Disponibilità (la garanzia che ogni richiesta riceva una risposta su ciò che è riuscito o fallito)
 - Tolleranza di partizione (il sistema continua a funzionare nonostante arbitrarie perdite di messaggi)
- Secondo il teorema, un sistema distribuito è in grado di soddisfare al massimo due di queste garanzie allo stesso tempo, ma non tutte e tre.

Pattern: Database per service

- Svantaggi:
- L'implementazione di transazioni commerciali che si estendono su più servizi non è semplice. Le transazioni distribuite andrebbero evitate per il teorema CAP. Inoltre, molti database moderni (NoSQL) non le supportano. La soluzione migliore è usare il pattern Saga. I servizi pubblicano eventi quando aggiornano i dati. Altri servizi si iscrivono agli eventi e aggiornano i loro dati in risposta.
- L'implementazione di query che uniscono dati che ora si trovano in più database è complessa.
- Complessità nella gestione di più database SQL e NoSQL

Pattern: Database per service

- Soluzioni:
- Composizione delle API: l'applicazione esegue il join anziché il database. Ad esempio, un servizio (o il gateway API) potrebbe recuperare un cliente e i suoi ordini recuperando prima il cliente dal servizio clienti e quindi interrogando il servizio ordini per restituire gli ordini più recenti del cliente.
- Command Query Responsibility Segregation (CQRS) - mantiene una o più viste materializzate che contengono dati da più servizi. Le visualizzazioni sono gestite da servizi che si iscrivono agli eventi che ogni servizio pubblica quando aggiorna i suoi dati. Ad esempio, un negozio online potrebbe implementare una query che trova i clienti in una particolare regione e i loro ordini recenti mantenendo una vista che unisce clienti e ordini. La visualizzazione viene aggiornata da un servizio che si abbona agli eventi di cliente e ordini.

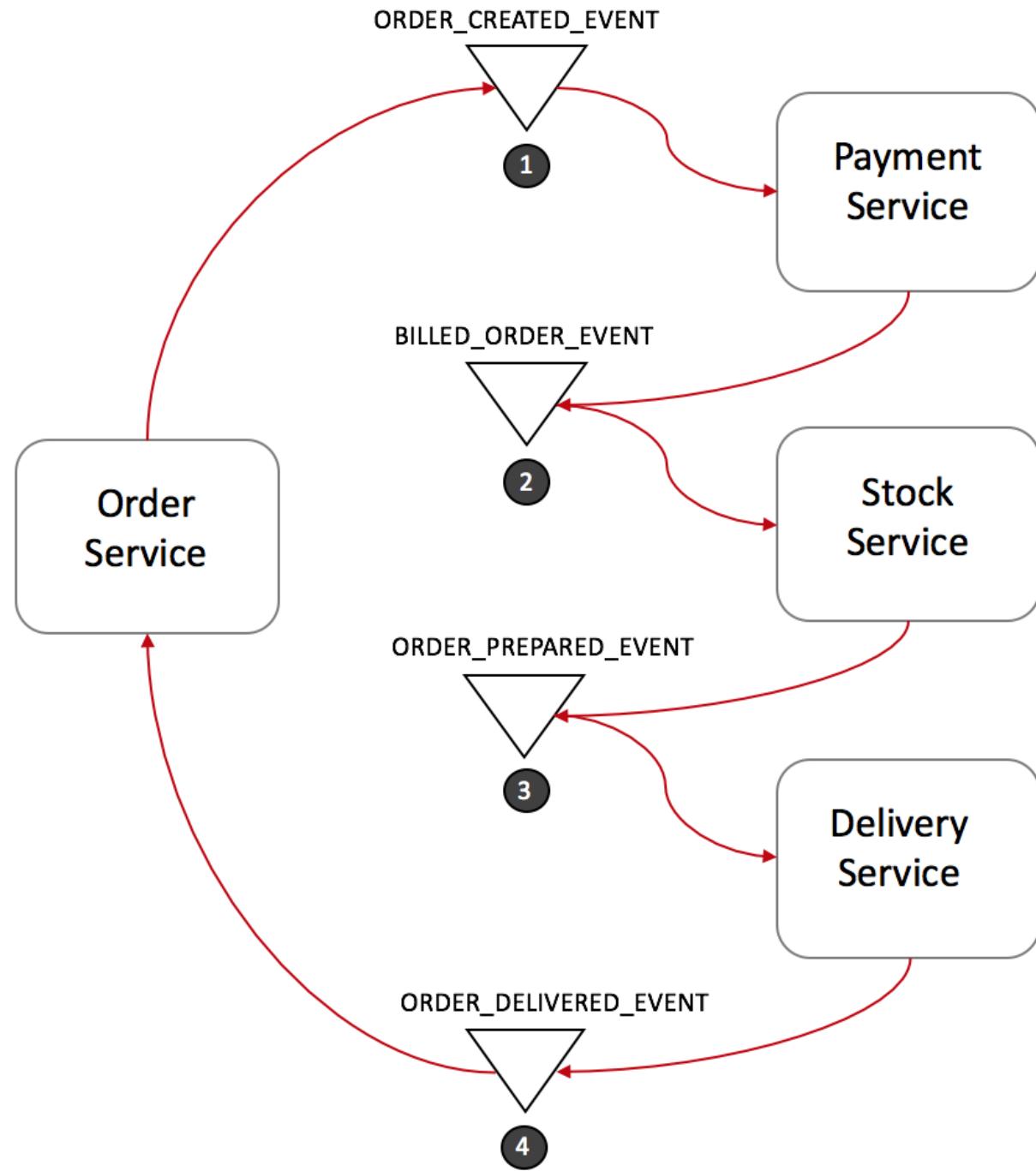
Pattern: Saga

- Usando il pattern Database per Service ogni servizio ha il proprio database.
- Alcune transazioni commerciali, tuttavia, comprendono più servizi, pertanto è necessario un meccanismo per garantire la coerenza dei dati tra i servizi. Ad esempio, immaginiamo che stai costruendo un negozio di e-commerce in cui i clienti hanno un limite di credito. L'applicazione deve garantire che un nuovo ordine non superi il limite di credito del cliente. Poiché gli ordini e i clienti si trovano in diversi database, l'applicazione non può semplicemente utilizzare una transazione ACIDa locale.
- **Problema**
- Come mantenere la coerenza dei dati tra i servizi?
- **Driver**
- 2PC non è un'opzione (Two-Phases Commit protocol)

Pattern: Saga

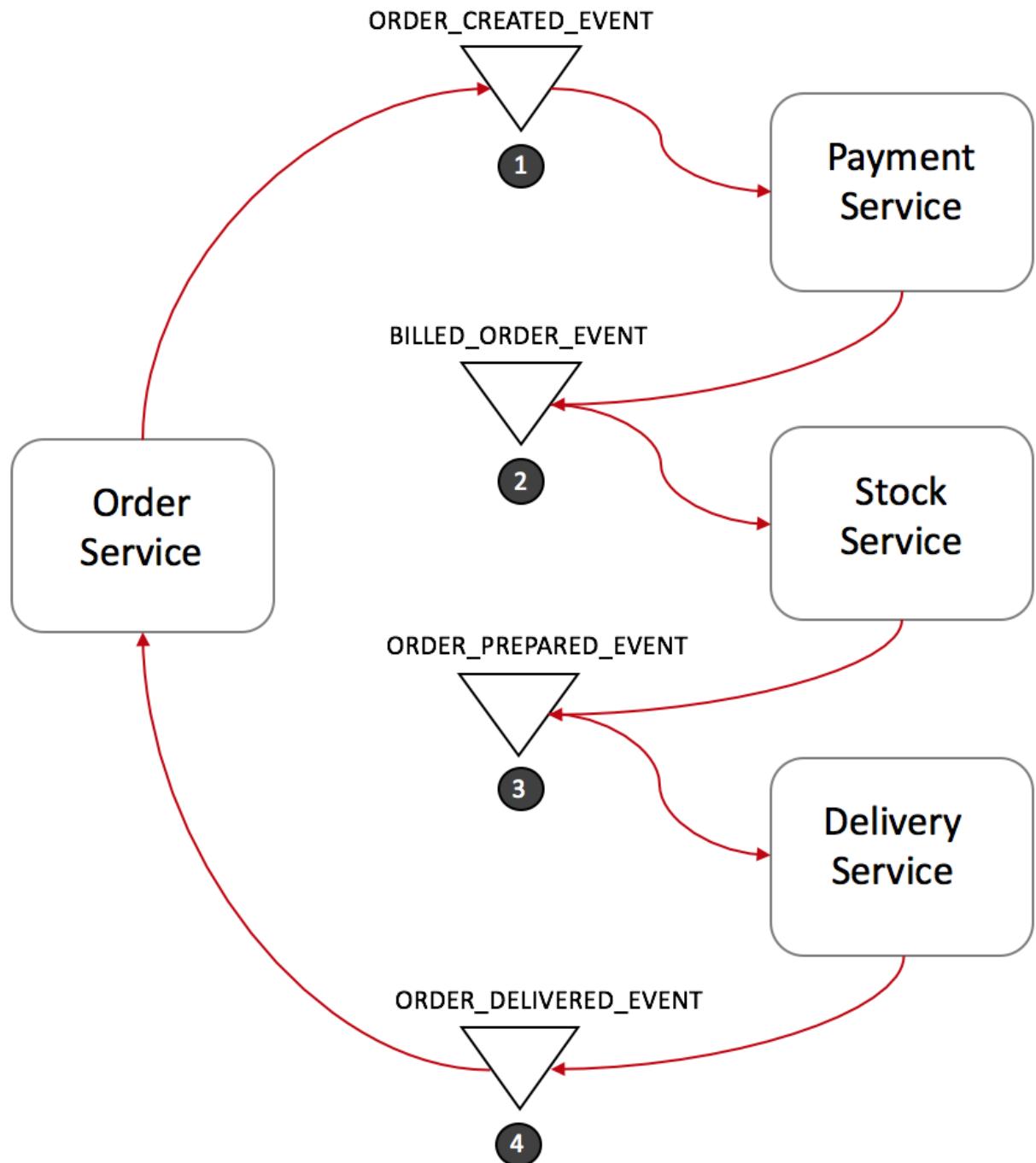
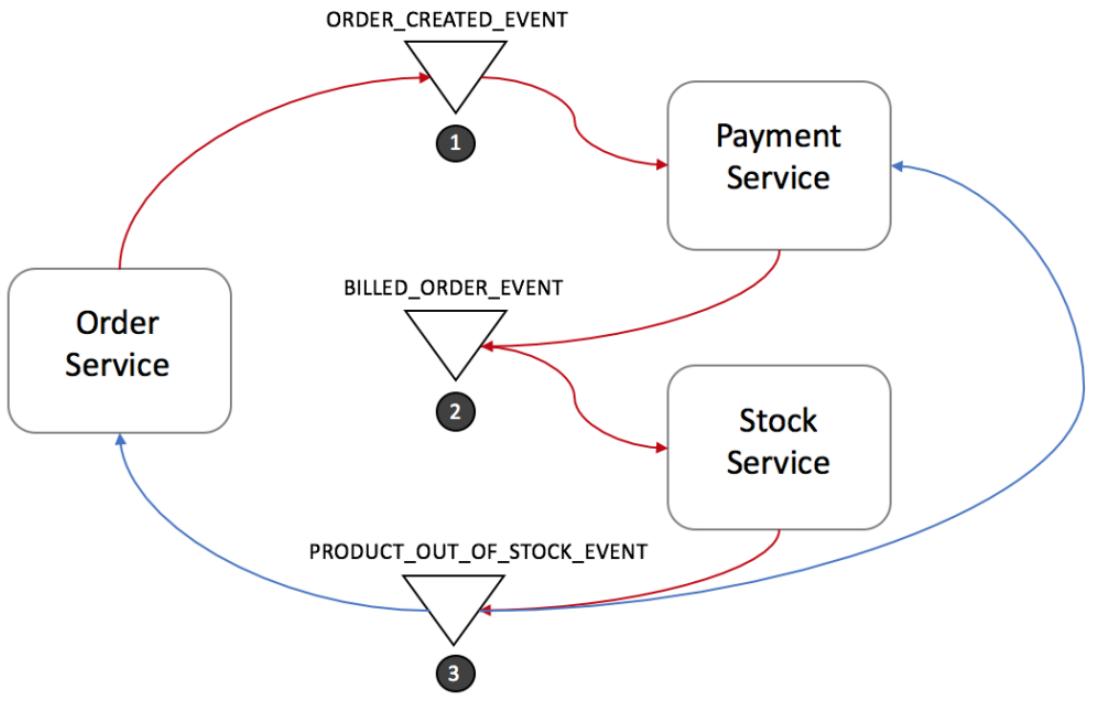
- **Soluzione**
- Implementa ogni transazione aziendale che si estende su più servizi come una saga. Una saga è una sequenza di transazioni locali. Ogni transazione locale aggiorna il database e pubblica un messaggio o un evento per attivare la successiva transazione locale nella saga. Se una transazione locale non riesce perché viola una regola aziendale, la saga esegue una serie di transazioni compensative che annullano le modifiche apportate dalle transazioni locali precedenti.

Pattern: Saga



Pattern: Saga

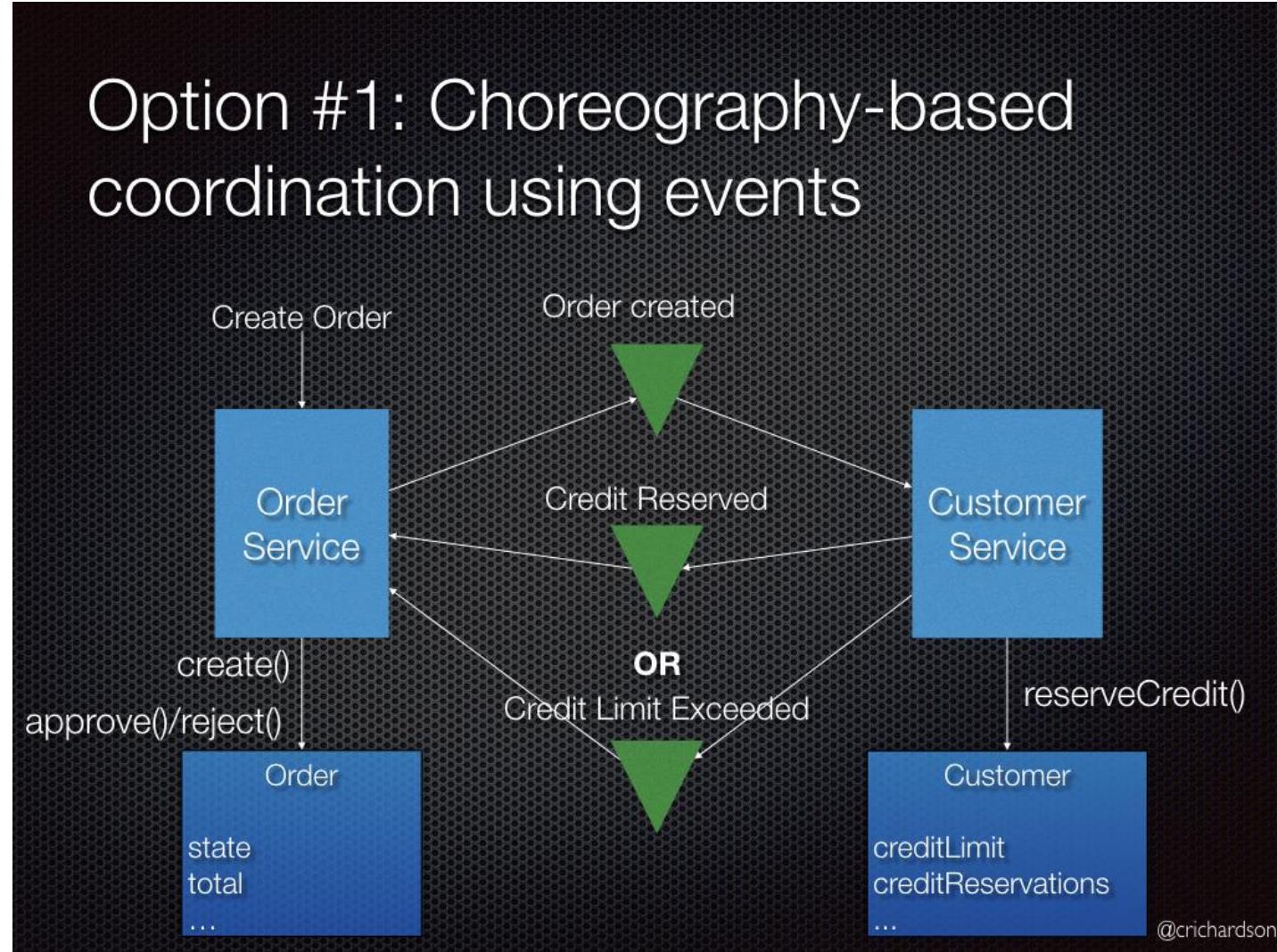
Rollback



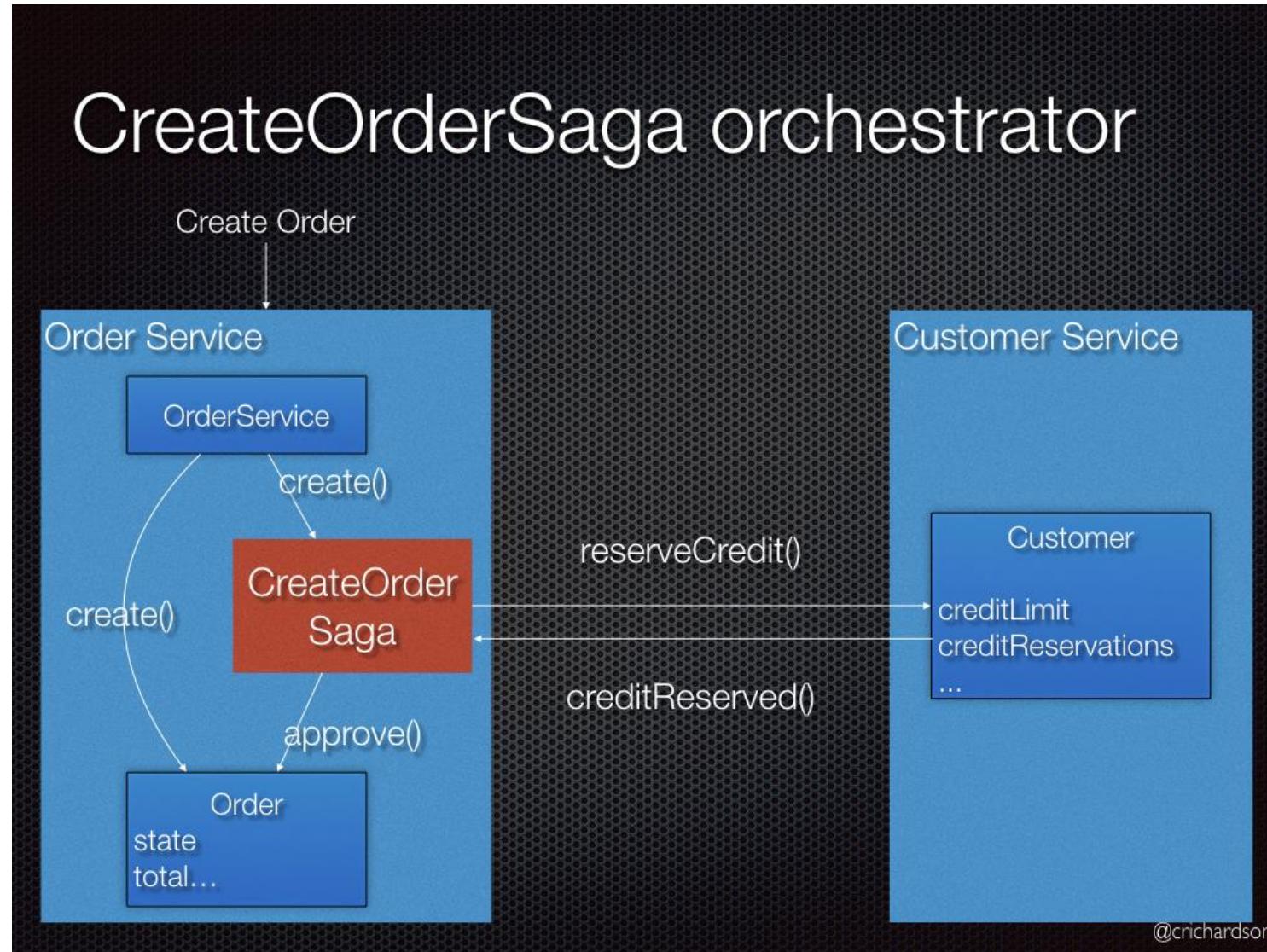
Pattern: Saga

- Ci sono due modi per coordinare le saghe:
 - Coreografia: ogni transazione locale pubblica eventi di dominio che attivano transazioni locali in altri servizi
 - Orchestrazione: un orchestratore (oggetto) indica ai partecipanti le transazioni locali da eseguire

Pattern: Saga



Pattern: Saga



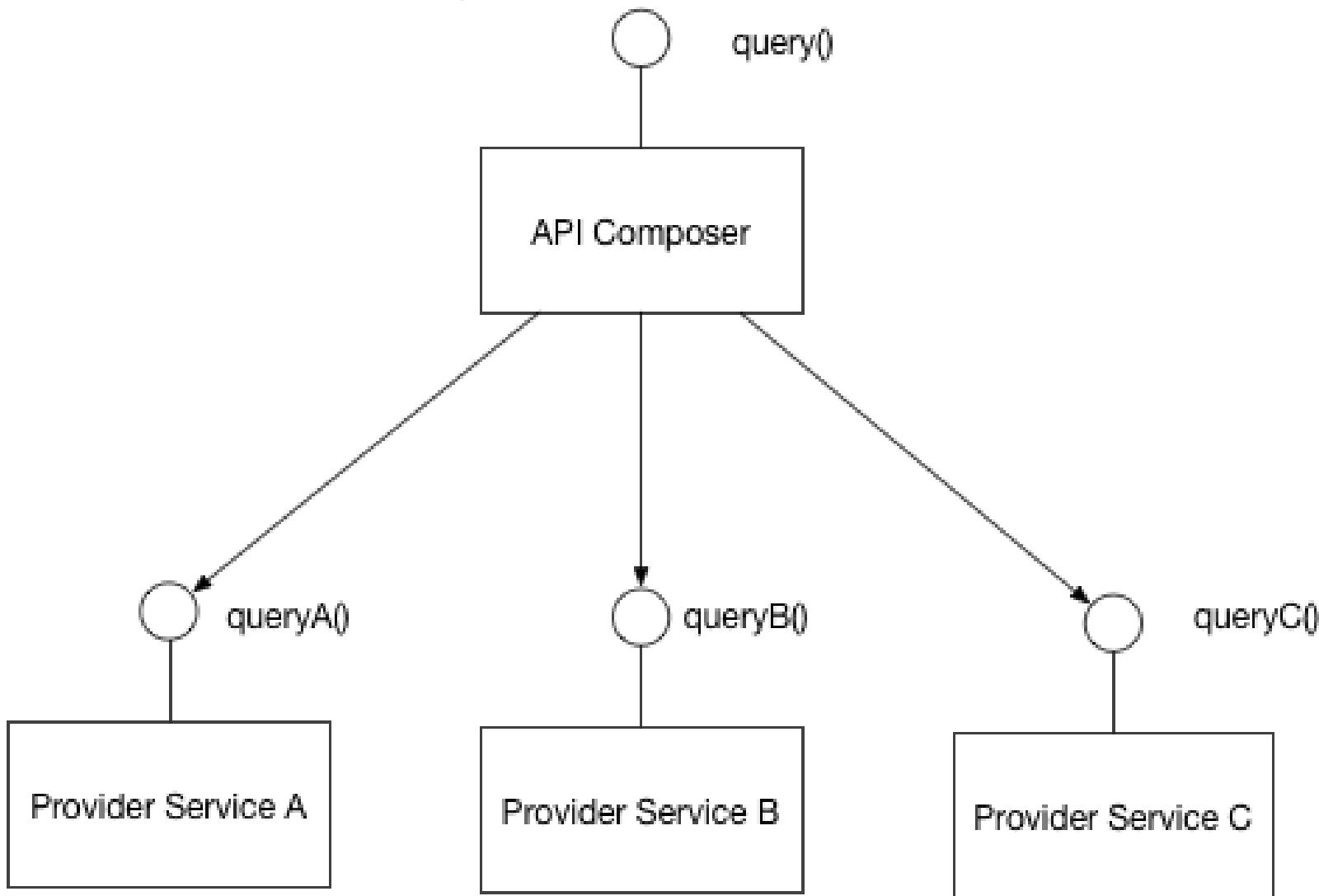
Pattern: Saga

- **Vantaggi:**
- Consente a un'applicazione di mantenere la coerenza dei dati tra più servizi senza utilizzare transazioni distribuite
- **Svantaggi:**
- Il modello di programmazione è più complesso. Ad esempio, uno sviluppatore deve progettare transazioni compensative che annullino esplicitamente le modifiche apportate in precedenza in una saga.
- **Altre questioni** da non trascurare:
 - Per essere affidabile, un servizio deve aggiornare atomicamente il proprio database e pubblicare un messaggio / evento. Non può utilizzare il meccanismo tradizionale di una transazione distribuita che abbraccia il database e il broker dei messaggi. Invece, deve utilizzare uno dei pattern seguenti.
 - Sourcing di eventi
 - Eventi applicativi
 - Una saga basata su coreografia può pubblicare eventi utilizzando Aggregati e Eventi di dominio

Pattern: API Composition

- È stato applicato il pattern di architettura a microservizi e il pattern Database per service. Di conseguenza, non è più semplice implementare query che uniscono dati da più servizi.
- **Problema**
- Come implementare le query in un'architettura a microservizi?
- **Soluzione**
- Implementare una query definendo una Composer API, che richiama i servizi che possiedono i dati ed esegue un join in memoria dei risultati.

Pattern: API Composition



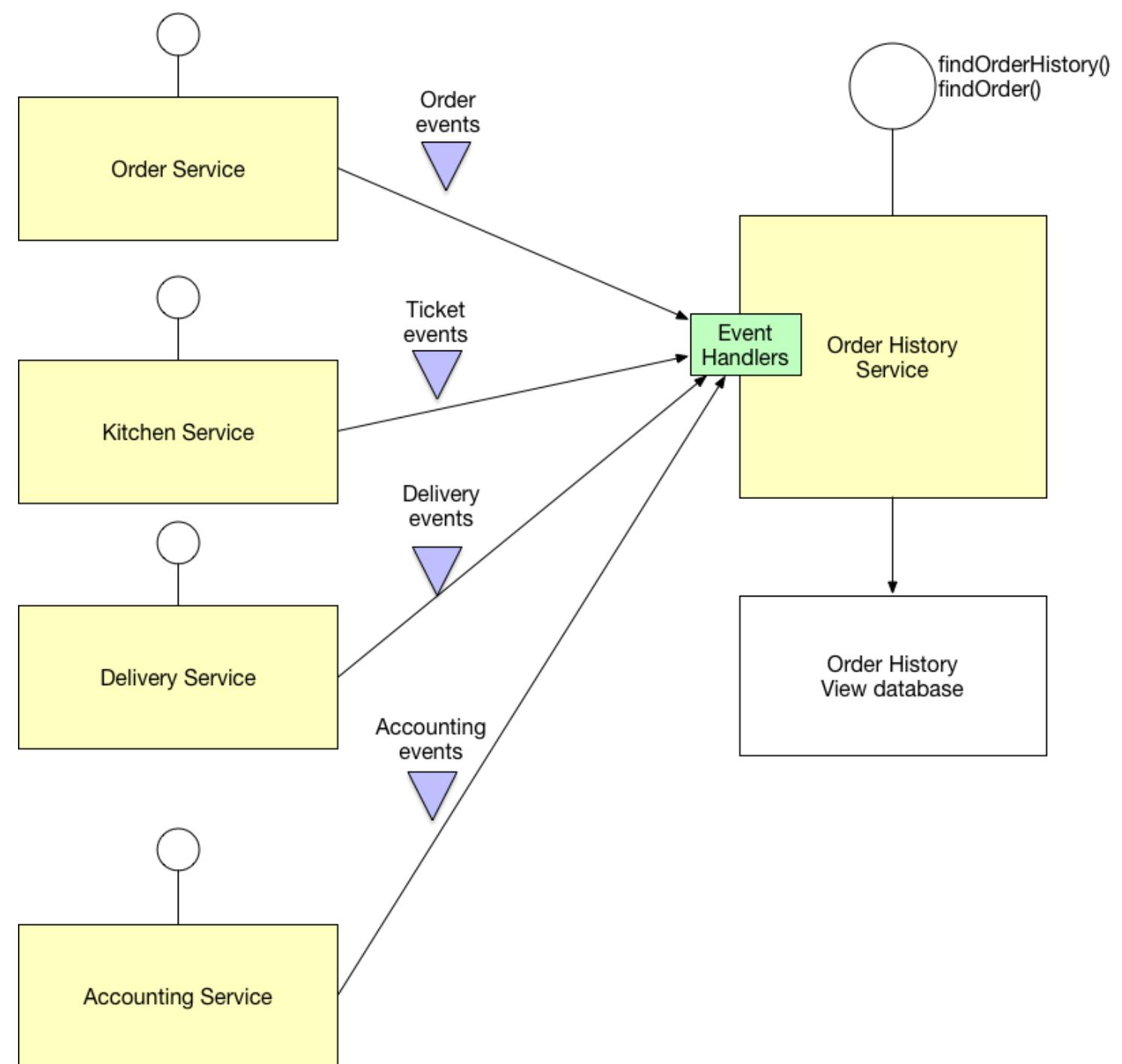
Pattern: API Composition

- **Vantaggi:**
- È un modo semplice per interrogare i dati in un'architettura a microservizi
- **Svantaggi:**
- Alcune query comporterebbero join in memoria inefficienti per set di dati di grandi dimensioni.

Pattern: Command Query Responsibility Segregation (CQRS)

- È stato applicato il pattern di architettura a microservizi e il pattern Database per service. Di conseguenza, non è più semplice implementare query che uniscono dati da più servizi.
 - Inoltre, se è stato applicato il pattern di sourcing di eventi, i dati non sono più facilmente interrogabili.
-
- **Problema**
 - Come implementare una query che recupera i dati da più servizi in un'architettura a microservizi?
-
- **Soluzione**
 - Definire un database di visualizzazione, che è una replica di sola lettura progettata per supportare tale query. L'applicazione mantiene la replica fino ai dati sottoscrivendo gli eventi di dominio pubblicati dal servizio proprietario dei dati.

Pattern: Command Query Responsibility Segregation (CQRS)



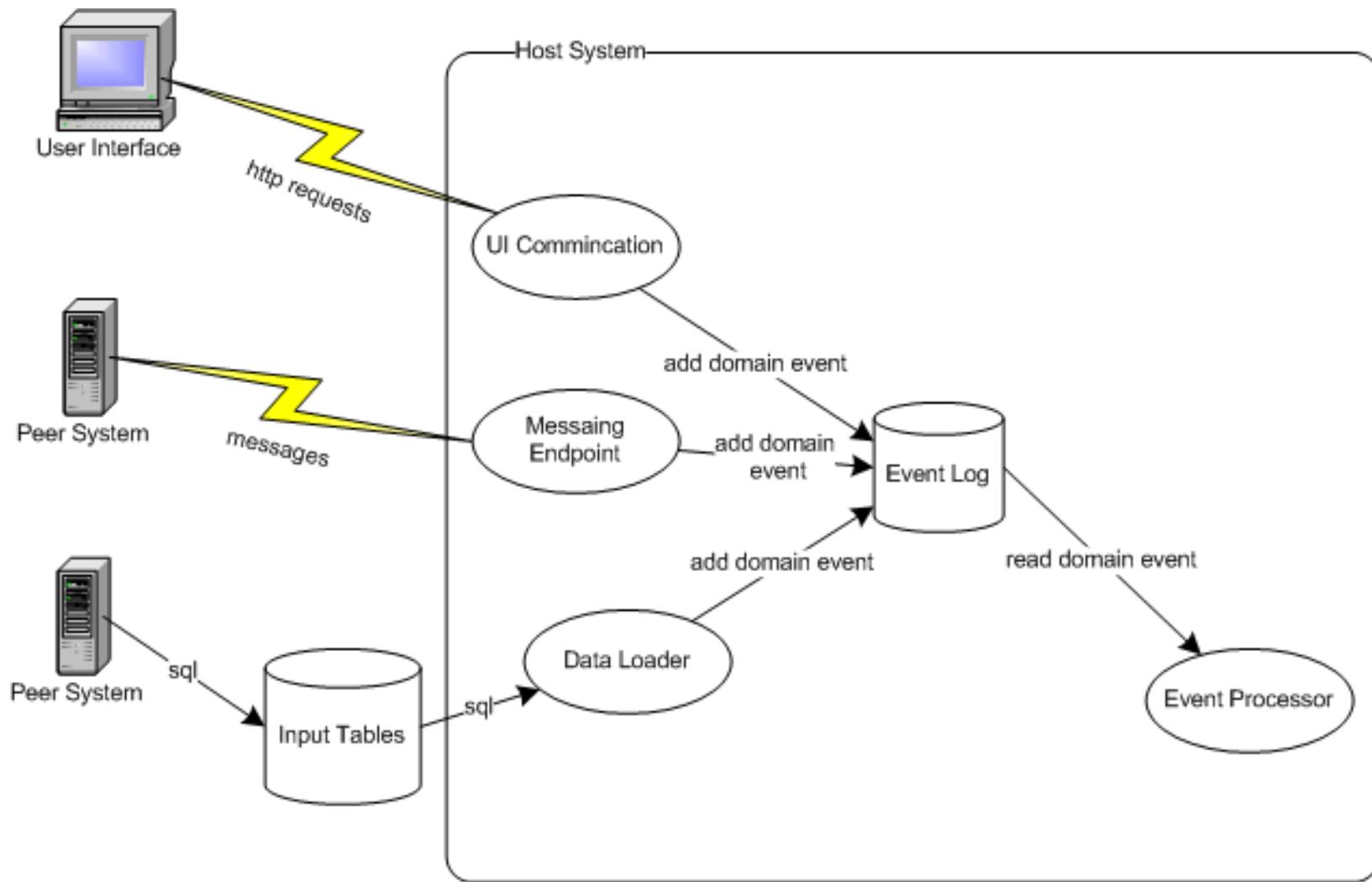
Pattern: Command Query Responsibility Segregation (CQRS)

- **Vantaggi**
 - Supporta più viste denormalizzate che sono scalabili e performanti
 - Migliore separazione delle richieste = comandi e query più semplici
 - Indispensabile in un'architettura con pattern di Event Sourcing
- **Svantaggi**
 - Aumento della complessità
 - Potenziale duplicazione del codice
 - Ritardo di replica / coerenza delle viste

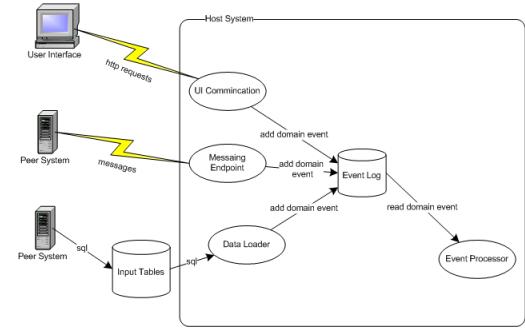
Pattern: Domain event

- Nasce in un Domain-Driven Design (DDD).
- Un servizio spesso ha bisogno di pubblicare eventi quando aggiorna i suoi dati. Questi eventi potrebbero essere necessari, ad esempio, per aggiornare una vista CQRS. In alternativa, il servizio potrebbe partecipare a una saga basata sulla coreografia, che utilizza eventi per il coordinamento.
- **Problema**
- In che modo un servizio pubblica un evento quando aggiorna i suoi dati?
- **Soluzione**
- Organizzare la logica di business di un servizio come una raccolta di aggregati DDD che emettono eventi di dominio quando vengono creati o aggiornati. Il servizio pubblica questi eventi di dominio in modo che possano essere utilizzati da altri servizi.

Pattern: Domain event

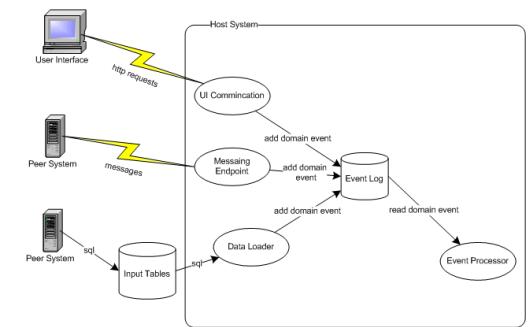


Pattern: Domain event



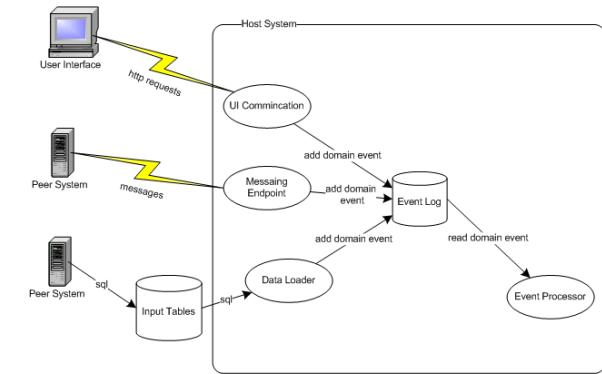
- Si ha un sistema con input da un'interfaccia utente, un sistema di messaggistica e alcune manipolazioni dirette delle tabelle del database. Per risolvere questi eventi con Domain Event, abbiamo componenti che interagiscono con ciascuno di questi flussi di input e convertiamo gli input in un flusso di Domain Event che sono memorizzati in un log persistente. Un processore di eventi legge quindi gli eventi dal log e li elabora attivando la nostra applicazione per fare tutto ciò che si suppone di fare.
- In questo stream, il primo livello di input del sistema non intraprende alcuna azione sullo stimolo se non per creare e registrare un evento. Il secondo livello può quindi ignorare l'effettiva fonte di input, reagisce semplicemente all'evento e lo elabora.
- Spesso ha senso separare i log se gli eventi hanno requisiti di risposta diversi. Un'interfaccia utente in genere richiede un tempo di risposta molto più rapido rispetto a molti sistemi di messaggistica remota, quindi ha senso inserire il traffico dell'interfaccia utente in un registro diverso e utilizzare un processore separato.

Pattern: Domain event



- Il diagramma implica l'interazione di pipe e filtri asincroni, ma questa non è una parte essenziale dell'approccio. In effetti, un approccio comune, in particolare con gli stimoli dell'interfaccia utente, consiste nel fare in modo che il gestore dell'interfaccia utente invochi il processore di eventi direttamente in un'interazione sincrona.
- Ogni evento di dominio acquisisce informazioni dallo stimolo esterno. Poiché questo è registrato e vogliamo utilizzare il log come traccia di controllo, è importante che i dati di quest'ultimo siano immutabili. Dal momento in cui si è creato l'oggetto evento non deve essere possibile modificarlo. Tuttavia c'è anche un altro tipo di dati sull'evento che registra ciò che un sistema ha fatto con esso - i dati di elaborazione. I dati su un evento di dominio sono dati immutabili che catturano l'evento e dati di elaborazione mutabili che registrano ciò che il sistema fa in risposta ad esso. I dati di origine su un addebito su carta di credito includevano l'ammontare della tariffa, chi era il venditore, ecc. I dati di elaborazione potrebbero includere la dichiarazione in cui è apparso. Se la piattaforma ha un supporto particolare per oggetti immutabili, può valere la pena dividere l'evento in due oggetti per trarne vantaggio.
- Sebbene i dati degli eventi non possano mai cambiare, è possibile che il sistema debba gestire una modifica, in genere perché l'evento originale non era corretto. Puoi gestirlo ricevendo questa modifica come un Evento Retroattivo separato. Il processore quindi gestisce l'evento retroattivo per correggere le conseguenze dell'evento errato precedente. Spesso questa elaborazione può essere eseguita a un livello molto generico.
- Una terza, ma solo occasionale, categoria di dati degli eventi è memorizzata nella cache dei dati da derivazioni da altri dati sul flusso di eventi. In questi casi, il processore di eventi riepiloga le informazioni degli eventi passati durante l'elaborazione dell'evento corrente e aggiunge i dati riepilogati all'evento corrente per accelerare l'elaborazione futura. Come ogni cache, è importante segnalare il fatto che questi dati sono liberi di essere rimossi e ricalcolati in caso di modifiche.

Pattern: Domain event



- Diversi eventi si verificano per diversi motivi, quindi è comune utilizzare diversi tipi di eventi. L'event processor utilizzerà quindi il tipo di evento come parte del suo meccanismo di invio. Il tipo di evento può essere rappresentato utilizzando sottotipi di eventi o un oggetto di tipo evento separato o una combinazione di entrambi.
- È abbastanza comune che diversi tipi di eventi portino dati diversi, quindi l'utilizzo di sottotipi di eventi per i tipi di evento si adatta bene a questo. Il problema con i sottotipi è che questo porta a una proliferazione di tipi, che è particolarmente frustrante se gran parte dei dati è la stessa. Un approccio ibrido utilizza sottotipi per gestire dati diversi e oggetti di tipo evento per segnalare il dispacciamento.
- Gli eventi riguardano qualcosa che accade in un determinato momento, quindi è naturale che gli eventi contengano informazioni temporali. Quando si esegue questa operazione è importante considerare due Time Point che potrebbero essere memorizzati con l'evento: il momento in cui si è verificato l'evento e il momento in cui l'evento è stato annotato. Questi punti di tempo corrispondono alle nozioni di tempo reale e timestamp.
- È possibile che si possa aver bisogno di più punti di registrazione per memorizzare quando l'evento è stato annotato da sistemi diversi.

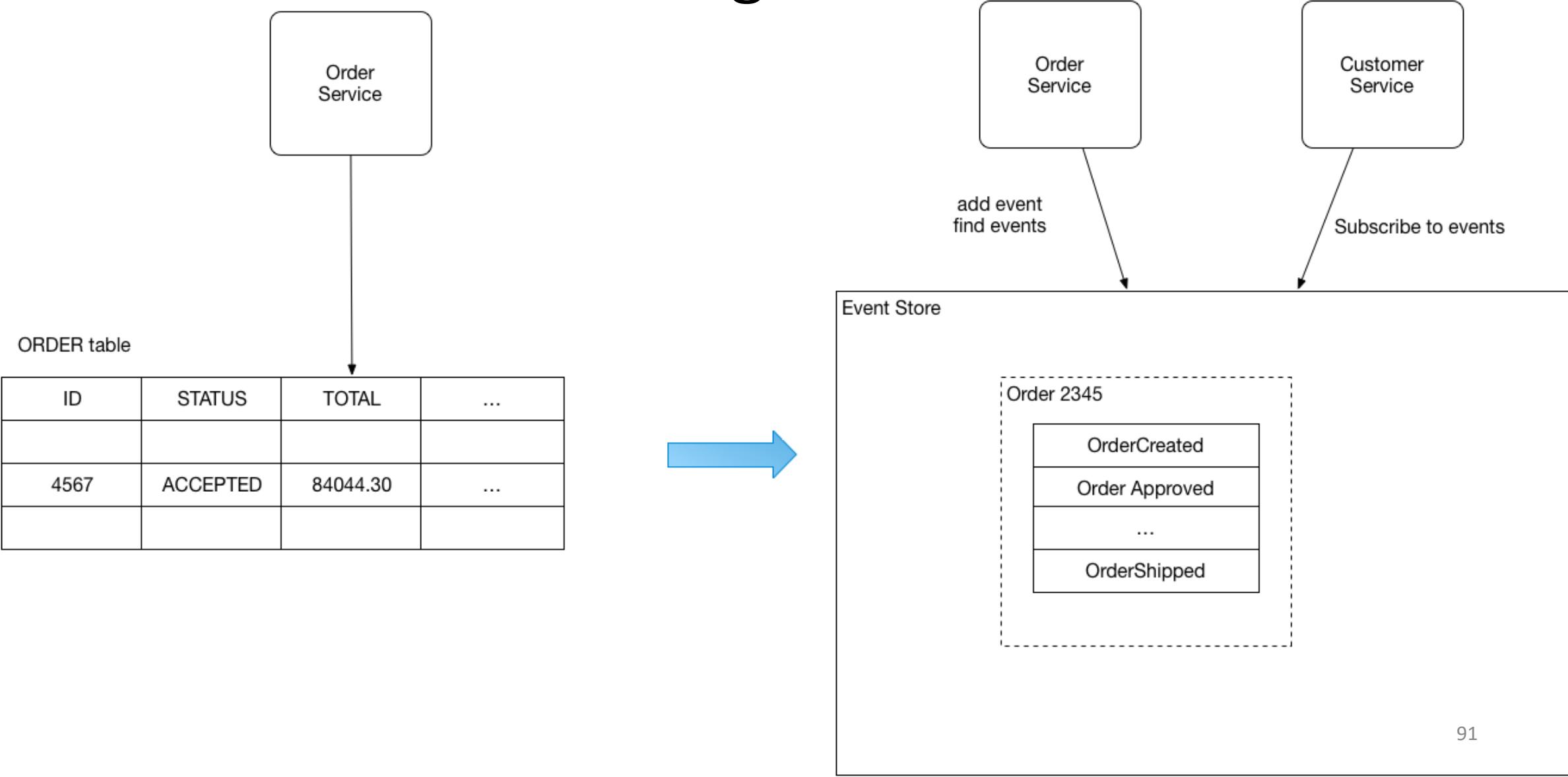
Pattern: Event sourcing

- In generale, un servizio deve aggiornare atomicamente il database e pubblicare messaggi / eventi. Ad esempio, usa il modello Saga. Per essere affidabile, ogni fase di una saga deve aggiornare atomicamente il database e pubblicare messaggi / eventi. In alternativa, potrebbe utilizzare il pattern di eventi Dominio, forse per implementare CQRS. In entrambi i casi, non è possibile utilizzare una transazione distribuita che si estende sul database e il broker dei messaggi per aggiornare il database e pubblicare messaggi / eventi.
- **Problema**
- Come aggiornare in modo affidabile / atomico il database e pubblicare messaggi / eventi.
- **Driver**
- 2PC non è possibile

Pattern: Event sourcing

- **Soluzione**
- Una buona soluzione a questo problema è utilizzare l'event sourcing. L'individuazione degli eventi persiste nello stato di un'entità aziendale come un Ordine o un Cliente come una sequenza di eventi che ne cambiano lo stato. Ogni volta che lo stato di un'entità aziendale cambia, un nuovo evento viene aggiunto all'elenco degli eventi. Poiché il salvataggio di un evento è una singola operazione, è intrinsecamente atomico. L'applicazione ricostruisce lo stato corrente di un'entità ripetendo gli eventi.
- Le applicazioni mantengono gli eventi in un archivio eventi, che è un database di eventi. Il negozio ha un'API per aggiungere e recuperare gli eventi di un'entità. Anche l'archivio eventi si comporta come un broker di messaggi. Fornisce un'API che consente ai servizi di iscriversi agli eventi. Quando un servizio salva un evento nell'archivio eventi, viene consegnato a tutti gli utenti interessati.
- Alcune entità, come un cliente, possono avere un numero elevato di eventi. Per ottimizzare il caricamento, un'applicazione può salvare periodicamente un'istantanea dello stato corrente di un'entità. Per ricostruire lo stato corrente, l'applicazione trova lo snapshot più recente e gli eventi che si sono verificati da quell'istantanea. Di conseguenza, ci sono meno eventi da riprodurre.

Pattern: Event sourcing



Pattern: Event sourcing

- **Vantaggi**
- Risolve uno dei problemi chiave nell'implementazione di un'architettura basata sugli eventi e rende possibile pubblicare in modo affidabile eventi ogni volta che si verificano cambiamenti di stato.
- Poiché persiste gli eventi piuttosto che gli oggetti del dominio, evita in gran parte il gap object-relational.
- Fornisce un registro di controllo, affidabile al 100%, delle modifiche apportate a un'entità aziendale
- Permette di implementare query temporali che determinano lo stato di un'entità in qualsiasi momento.
- La logica di business basata sugli approvvigionamenti di eventi è costituita da entità aziendali liberamente accoppiate che scambiano eventi. Ciò semplifica notevolmente la migrazione da un'applicazione monolitica a un'architettura a microservizi.
- **Svantaggi**
- È uno stile di programmazione diverso e sconosciuto e quindi c'è una curva di apprendimento.
- L'archivio eventi è difficile da interrogare poiché richiede query tipiche per ricostruire lo stato delle entità aziendali. È probabile che sia complesso e inefficiente. Di conseguenza, l'applicazione deve utilizzare Command Query Responsibility Segregation (CQRS) per implementare le query. Ciò a sua volta significa che le applicazioni devono gestire dati coerenti ma instabili in degli intervalli.

Eventuate

- Una piattaforma per Event Sourcing:
- <http://eventuate.io/>

Pattern: API Gateway / Backend for Front-End

- Immaginiamo di realizzare un negozio online che utilizza il modello di architettura a microservizi e che si sta implementando la pagina dei dettagli del prodotto. È necessario sviluppare più versioni dell'interfaccia utente dei dettagli del prodotto:
 - Interfaccia utente basata su HTML5 / JavaScript per browser desktop e mobili: l'HTML è generato da un'applicazione Web lato server
 - Client nativi per Android e iPhone: questi client interagiscono con il server tramite API REST
- Inoltre, il negozio online deve esporre i dettagli del prodotto tramite un'API REST per l'utilizzo da parte di applicazioni di terze parti.
- L'interfaccia utente di dettagli di un prodotto può visualizzare molte informazioni su un prodotto. Ad esempio, la pagina dei dettagli di Amazon.com visualizza:
 - Informazioni di base sul libro come titolo, autore, prezzo, ecc.
 - La cronologia degli acquisti per il libro
 - Disponibilità
 - Opzioni di acquisto
 - Altri oggetti che vengono spesso acquistati con questo libro
 - Altri oggetti acquistati dai clienti che hanno acquistato questo libro
 - Recensioni dei clienti
 - Classifica dei venditori
 - ...

Pattern: API Gateway / Backend for Front-End

- Poiché lo store online utilizza il modello di architettura a microservizi, i dati dei dettagli del prodotto sono distribuiti su più servizi. Per esempio,
 - Servizio informazioni sul prodotto: informazioni di base sul prodotto come titolo, autore
 - Servizio prezzi - prezzo del prodotto
 - Servizio ordini - cronologia acquisti per prodotto
 - Servizio di inventario - disponibilità del prodotto
 - Servizio di revisione - recensioni dei clienti ...
- Di conseguenza, il codice che visualizza i dettagli del prodotto deve in grado di recuperare le informazioni da tutti questi servizi.

Pattern: API Gateway / Backend for Front-End

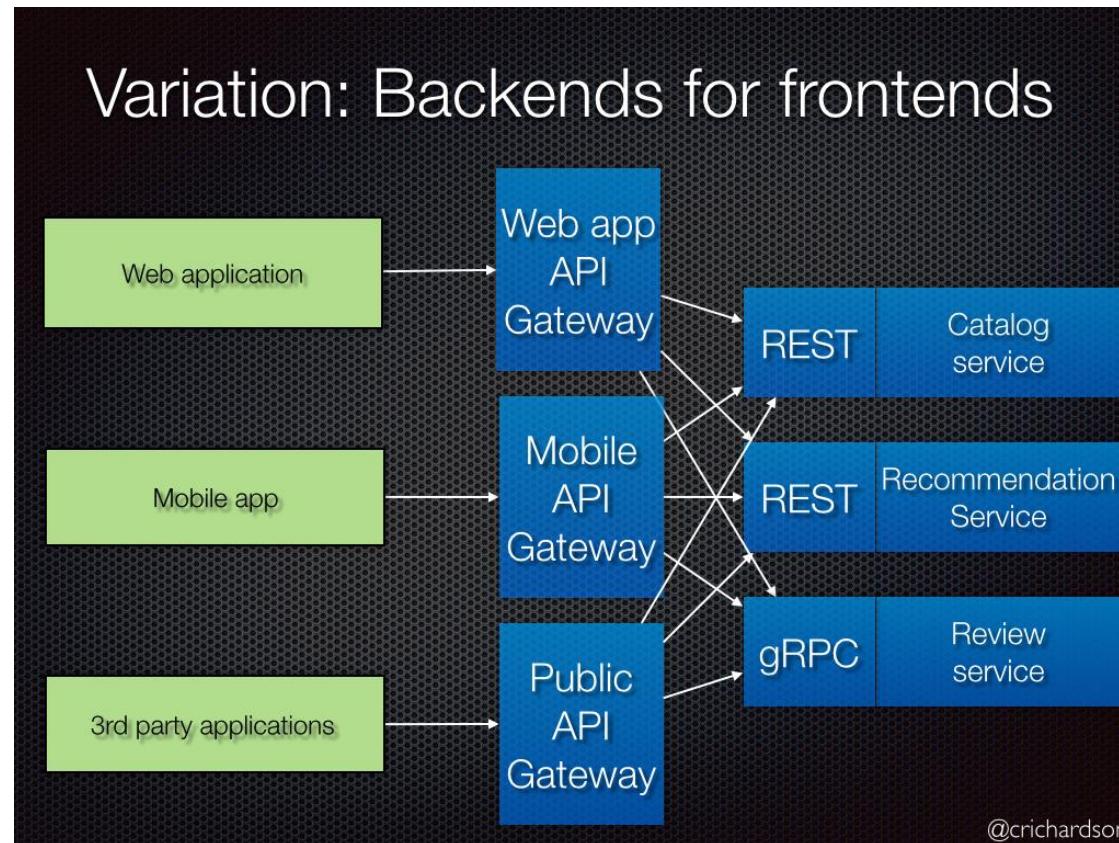
- **Problema**
 - In che modo i client di un'applicazione basata su microservizi accedono ai singoli servizi?
- **Driver**
 - La granularità delle API fornite dai microservizi è spesso diversa da quella di cui ha bisogno un cliente. I microservizi in genere forniscono API a grana fine, il che significa che i clienti devono interagire con più servizi. Ad esempio, come descritto sopra, un cliente che necessita dei dettagli per un prodotto ha bisogno di recuperare i dati da numerosi servizi.
 - Diversi clienti hanno bisogno di dati diversi. Ad esempio, la versione del browser desktop di un desktop della pagina dei dettagli del prodotto è in genere più elaborata della versione mobile.
 - Le prestazioni della rete sono diverse per diversi tipi di client. Ad esempio, una rete mobile è in genere molto più lenta e presenta una latenza molto più elevata rispetto a una rete non mobile. E, naturalmente, qualsiasi WAN è molto più lenta di una LAN. Ciò significa che un client mobile nativo utilizza una rete con caratteristiche prestazionali molto diverse rispetto a una LAN utilizzata da un'applicazione Web lato server. L'applicazione Web sul lato server può effettuare più richieste di back-end dei servizi senza influire sull'esperienza utente, laddove, come un client mobile, è possibile effettuare solo alcune.
 - Il numero di istanze di servizio e le loro posizioni (host + porta) cambiano dinamicamente
 - La suddivisione in servizi può cambiare nel tempo e deve essere nascosta ai clienti
 - I servizi potrebbero utilizzare una serie diversificata di protocolli, alcuni dei quali potrebbero non essere adatti al web

Pattern: API Gateway / Backend for Front-End

- **Soluzione**
- Implementare un gateway API che è il singolo punto di accesso per tutti i client. Il gateway API gestisce le richieste in due modi.
 - Alcune richieste vengono semplicemente inoltrate / inoltrate al servizio appropriato.
 - Altre richieste vengono coordinate su più servizi.
- Anziché fornire un'API adatta a tutti, il gateway API può esporre un'API diversa per ogni client. Ad esempio, il gateway API Netflix esegue un codice specifico per tipo di client che fornisce ad ogni client un'API più adatta alle sue esigenze.
- Il gateway API potrebbe anche implementare la sicurezza, ad es. verificare che il cliente sia autorizzato ad eseguire la richiesta

Pattern: API Gateway / Backend for Front-End

- Una variante di questo modello è il modello Backend per front-end. Definisce un gateway API separato per ogni tipo di client.



Pattern: API Gateway / Backend for Front-End

- **Vantaggi**
- Isola i client da come l'applicazione è partizionata in microservizi
- Isola i clienti dal problema di determinare le posizioni delle istanze di servizio
- Fornisce l'API ottimale per ogni cliente
- Riduce il numero di richieste / roundtrip. Ad esempio, il gateway API consente ai client di recuperare i dati da più servizi con un singolo round trip. Meno richieste significa anche meno spese generali e migliora l'esperienza dell'utente. Un gateway API è essenziale per le applicazioni mobili.
- Semplifica il client spostando la logica per la chiamata di più servizi dal client al gateway API
- Traduce da un protocollo API standard web-friendly "standard" a qualsiasi protocollo utilizzato internamente

Pattern: API Gateway / Backend for Front-End

- **Svantaggi**

- Maggiore complessità: il gateway API è un'altra parte mobile che deve essere sviluppata, implementata e gestita
- Aumento del tempo di risposta a causa dell'ulteriore hop di rete attraverso il gateway API - tuttavia, per la maggior parte delle applicazioni il costo di un roundtrip extra è insignificante.

- **Punti aperti**

- Come implementare il gateway API? Un approccio event-driven / reattivo è il migliore se deve scalare per gestire carichi elevati. Sulla JVM, le librerie basate su NIO come Netty, Spring Reactor, ecc. hanno senso. NodeJS è un'altra opzione.

Pattern: Microservice chassis

- Quando si avvia lo sviluppo di un'applicazione, si impiega spesso una notevole quantità di tempo per mettere in atto i meccanismi per gestire i problemi trasversali. Esempi di problemi trasversali includono:
 - Configurazione esterna: include credenziali e posizioni di rete di servizi esterni come database e broker di messaggi
 - Registrazione: configurazione di un framework di registrazione come log4j o logback
 - Controlli di integrità: un URL che un servizio di monitoraggio può eseguire "ping" per determinare lo stato dell'applicazione
 - Metriche: misurazioni che forniscono informazioni su ciò che l'applicazione sta facendo e su come sta andando
 - Tracciabilità distribuita: servizi di strumenti con codice che assegna a ogni richiesta esterna un identificativo univoco che viene passato tra i servizi.
- Oltre a questi problemi generici trasversali, esistono anche altre questione trasversali specifiche per le tecnologie utilizzate da un'applicazione. Le applicazioni che utilizzano servizi di infrastruttura come i database o i broker di messaggi richiedono una configurazione standard per farlo. Ad esempio, le applicazioni che utilizzano un database relazionale devono essere configurate con un pool di connessioni. Anche le applicazioni Web che elaborano richieste HTTP necessitano di una configurazione standard.
- È comune passare uno o due giorni, a volte anche di più, impostando questi meccanismi. Se si pensa di passare mesi o anni a sviluppare un'applicazione monolitica, l'investimento iniziale nella gestione dei problemi trasversali è insignificante. La situazione è molto diversa, tuttavia, se si sta sviluppando un'applicazione che ha un'architettura a microservizi. Ci sono decine o centinaia di servizi. Si creeranno spesso nuovi servizi, ognuno dei quali richiederà solo giorni o settimane per svilupparsi. Non ci si può permettere di passare qualche giorno a configurare i meccanismi per gestire i problemi trasversali. Ciò che è ancora peggio è che in un'architettura A microservizi ci sono ulteriori problemi trasversali che bisogna affrontare, inclusi la registrazione e l'individuazione del servizio e gli interruttori automatici per la gestione affidabile di guasti parziali.

Pattern: Microservice chassis

- **Driver**
- La creazione di un nuovo microservizio dovrebbe essere veloce e facile
- Quando si crea un microservizio, è necessario gestire i problemi trasversali come la configurazione, la registrazione, i controlli di integrità, le metriche, la registrazione e la scoperta dei servizi, gli interruttori di circuito. Esistono anche preoccupazioni trasversali specifiche per le tecnologie utilizzate dai microservizi.

Pattern: Microservice chassis

- **Soluzione**
- Costruire i microservizi utilizzando un telaio (chassis) di microservizi, che gestisce i problemi trasversali
- Esempi di framework per chassis a microservizi:
 - Java
 - Spring Boot e Spring Cloud
 - Dropwizard
 - Go
 - Gizmo
 - Micro
 - Go kit

Pattern: Circuit Breaker

- Si ha un'architettura a Microservizi.
- I servizi a volte collaborano durante la gestione delle richieste. Quando un servizio invoca in modo sincrono un altro, esiste sempre la possibilità che l'altro servizio non sia disponibile o che mostri una latenza così elevata da risultare sostanzialmente inutilizzabile. Risorse preziose come i thread potrebbero essere consumate nel chiamante in attesa che l'altro servizio risponda. Ciò potrebbe portare all'esaurimento delle risorse, che renderebbe il servizio chiamante incapace di gestire altre richieste. Il fallimento di un servizio può potenzialmente collegarsi ad altri servizi in tutta l'applicazione.
- **Problema**
- Come evitare che un errore di rete o di servizio si propaghi a cascata in altri servizi?
- **Soluzione**
- Un client di servizio dovrebbe richiamare un servizio remoto tramite un proxy che funziona in modo simile a un interruttore automatico. Quando il numero di guasti consecutivi supera una soglia, l'interruttore di circuito scatta e per la durata di un periodo di timeout tutti i tentativi di richiamare il servizio remoto si interrompono immediatamente. Al termine del timeout, l'interruttore automatico consente il passaggio di un numero limitato di richieste di test. Se tali richieste hanno successo, l'interruttore ripristina il normale funzionamento. Altrimenti, se si verifica un errore, il periodo di timeout ricomincia.

Pattern: Circuit Breaker

- **Vantaggi**
- I servizi gestiscono il malfunzionamento dei servizi che a loro volta invocano
- **Svantaggi**
- Il valore del timeout è difficile da quantificare senza evitare falsi positivi o introdurre latenze eccessive

Pattern: Server-side service discovery

- I servizi in genere devono chiamarsi l'un l'altro. In un'applicazione monolitica, i servizi si richiamano l'un l'altro attraverso metodi o chiamate remote a procedura.
- In una sistema distribuito tradizionale, i servizi vengono eseguiti in posizioni fisse e ben note (host e porte) e possono quindi chiamarsi facilmente utilizzando HTTP / REST o alcuni meccanismi RPC. Tuttavia, una moderna applicazione basata su microservizi viene generalmente eseguita in ambienti virtualizzati o containerizzati in cui il numero di istanze di un servizio e le relative posizioni cambiano in modo dinamico.
- Di conseguenza, è necessario implementare un meccanismo che consenta ai client del servizio di effettuare richieste a un insieme di istanze di servizio che cambia dinamicamente.

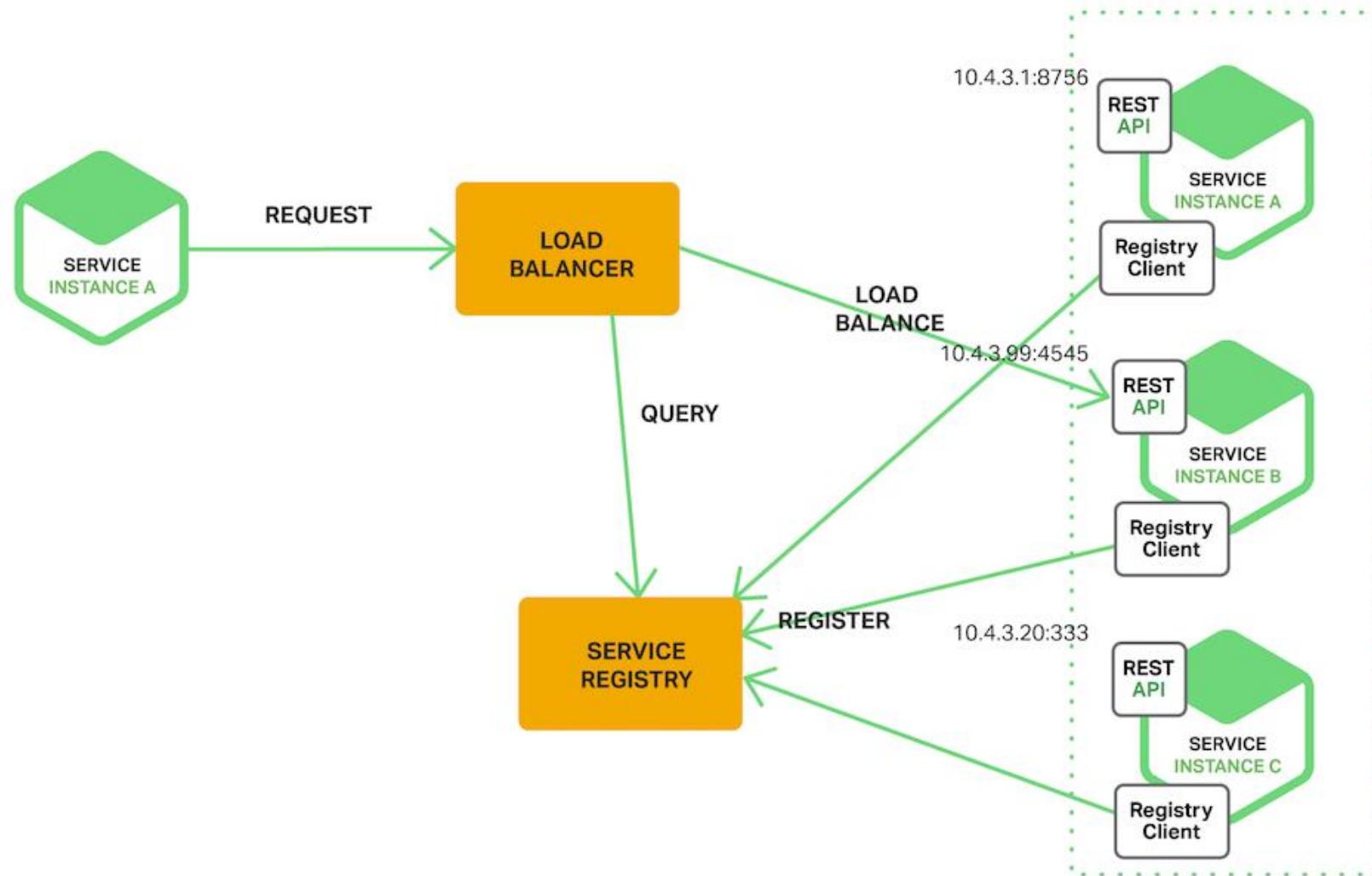
Pattern: Server-side service discovery

- I servizi in genere devono chiamarsi l'un l'altro. In un'applicazione monolitica, i servizi si richiamano l'un l'altro attraverso metodi o chiamate remote a procedura.
- In una sistema distribuito tradizionale, i servizi vengono eseguiti in posizioni fisse e ben note (host e porte) e possono quindi chiamarsi facilmente utilizzando HTTP / REST o alcuni meccanismi RPC. Tuttavia, una moderna applicazione basata su microservizi viene generalmente eseguita in ambienti virtualizzati o containerizzati in cui il numero di istanze di un servizio e le relative posizioni cambiano in modo dinamico.
- Di conseguenza, è necessario implementare un meccanismo che consenta ai client del servizio di effettuare richieste a un insieme di istanze di servizio che cambia dinamicamente.

Pattern: Server-side service discovery

- **Problema**
- In che modo il client di un servizio, il gateway API o un altro servizio, rileva la posizione di un'istanza del servizio?
- **Driver**
- Ogni istanza di un servizio espone un'API remota come HTTP / REST o Thrift ecc. in una particolare posizione (host e porta)
- Il numero di istanze di servizi e le loro posizioni cambia dinamicamente.
- A macchine e contenitori virtuali vengono in genere assegnati indirizzi IP dinamici.
- Il numero di istanze di servizi potrebbe variare in modo dinamico. Ad esempio, un gruppo di scalabilità automatica regola il numero di istanze in base al carico.
- **Soluzione**
- Quando si effettua una richiesta a un servizio, il client effettua una richiesta tramite un router (a.k.a load balancer) che viene eseguito in una posizione ben nota. Il router interroga un registro di servizio, che potrebbe essere incorporato nel router, e inoltra la richiesta a un'istanza di servizio disponibile.

Pattern: Server-side service discovery



Pattern: Server-side service discovery

- Un Elastic Load Balancer (ELB) di AWS è un esempio di router di rilevamento lato server. Un client effettua richieste HTTP (o apre connessioni TCP) all'ELB, che carica il traffico tra una serie di istanze EC2. Un ELB può bilanciare il carico del traffico esterno da Internet o, se installato in un VPC, bilanciare il carico del traffico interno. Un ELB funziona anche come registro di servizio. Le istanze EC2 vengono registrate con l'ELB esplicitamente tramite una chiamata API o automaticamente come parte di un gruppo con ridimensionamento automatico.
- Alcune soluzioni di clustering come Kubernetes e Marathon eseguono un proxy su ciascun host che funziona come router di individuazione sul lato server. Per accedere a un servizio, un client si connette al proxy locale utilizzando la porta assegnata a quel servizio. Il proxy inoltra quindi la richiesta a un'istanza di servizio in esecuzione da qualche parte nel cluster.

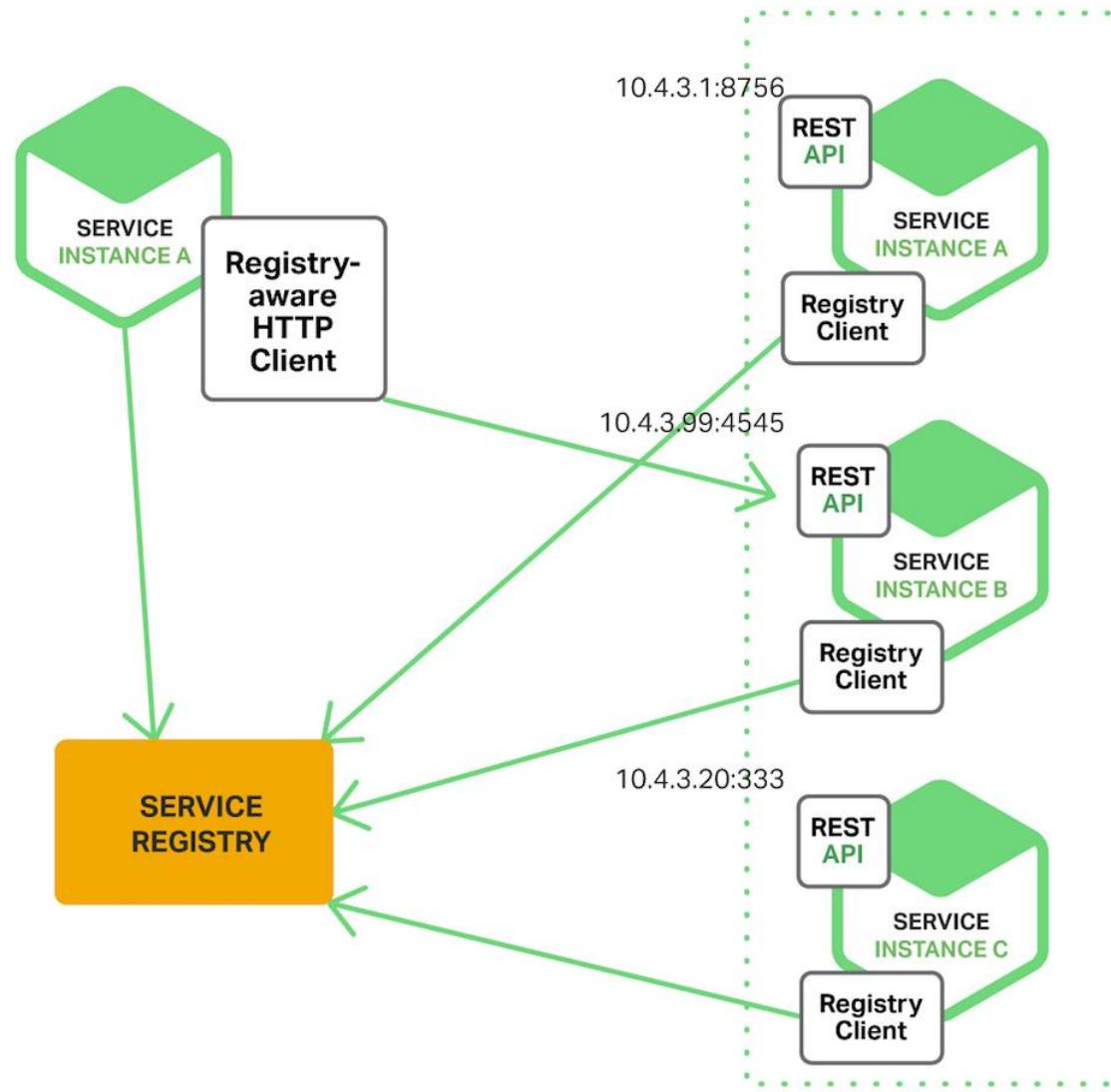
Pattern: Server-side service discovery

- **Vantaggi**
 - Rispetto al rilevamento lato client, il codice client è più semplice poiché non deve occuparsi della scoperta. Invece, un client fa semplicemente una richiesta al router
 - Alcuni ambienti cloud forniscono questa funzionalità, ad es. Bilanciamento del carico elastico AWS
- **Svantaggi**
 - A meno che non faccia parte dell'ambiente cloud, il router deve essere un altro componente di sistema che deve essere installato e configurato. Dovrà anche essere replicato per disponibilità e capacità.
 - Il router deve supportare i protocolli di comunicazione necessari (ad es. HTTP, gRPC, Thrift, ecc.) A meno che non sia un router basato su TCP
 - Sono necessari più hop di rete rispetto all'utilizzo di Client Side Discovery

Pattern: Client-side service discovery

- **Problema**
- In che modo il client di un servizio, il gateway API o un altro servizio, rileva la posizione di un'istanza del servizio?
- **Driver**
- Ogni istanza di un servizio espone un'API remota come HTTP / REST o Thrift ecc. in una particolare posizione (host e porta)
- Il numero di istanze di servizi e le loro posizioni cambia dinamicamente.
- A macchine e contenitori virtuali vengono in genere assegnati indirizzi IP dinamici.
- Il numero di istanze di servizi potrebbe variare in modo dinamico. Ad esempio, un gruppo di scalabilità automatica regola il numero di istanze in base al carico.
- **Soluzione**
- Quando si effettua una richiesta a un servizio, il client ottiene il percorso di un'istanza del servizio interrogando un Registro servizi, che conosce le posizioni di tutte le istanze del servizio.

Pattern: Client-side service discovery



Pattern: Client-side service discovery

- **Vantaggi**
 - Meno parti mobili e hop di rete rispetto a Server Side Discovery
- **Svantaggi**
 - Questo modello associa il client al registro del servizio
 - È necessario implementare la logica di individuazione del servizio lato client per ogni linguaggio / framework di programmazione utilizzato dall'applicazione, ad esempio Java / Scala, JavaScript / NodeJS. Ad esempio, Netflix Prana fornisce un approccio basato su proxy HTTP per l'individuazione dei servizi per i client non JVM.

Il caso Netflix

- <https://medium.com/netflix-techblog/optimizing-the-netflix-api-5c9ac715cf19>

Confronto con la progettazione di applicazioni monolitiche

Esempi e confronti sui microservizi

- <https://www.guru99.com/microservices-tutorial.html>
- <https://www.devteam.space/blog/microservice-architecture-examples-and-diagram/>

Il test dei microservizi

- <https://labs.spotify.com/2018/01/11/testing-of-microservices/>
- <https://medium.freecodecamp.org/these-are-the-most-effective-microservice-testing-strategies-according-to-the-experts-6fb584f2edde>
- <https://www.testingexcellence.com/testing-microservices-beginners-guide/>

Sviluppare microservizi con Spring Boot e Spring Cloud

Microservizi e Spring/1

- <http://www.springboottutorial.com/creating-microservices-with-spring-boot-part-1-getting-started>
- <http://www.springboottutorial.com/creating-microservices-with-spring-boot-part-2-forex-microservice>
- <http://www.springboottutorial.com/creating-microservices-with-spring-boot-part-3-currency-conversion-microservice>
- <http://www.springboottutorial.com/microservices-with-spring-boot-part-4-ribbon-for-load-balancing>
- <http://www.springboottutorial.com/microservices-with-spring-boot-part-5-eureka-naming-server>

Microservizi e Spring/2

- <https://medium.com/omarelgabrys-blog/microservices-with-spring-boot-intro-to-microservices-part-1-c0d24cd422c3>
- <https://spring.io/blog/2015/07/14/microservices-with-spring>

App per dispositivi mobile e
microservizi

Mobile apps e microservizi

- <https://hackernoon.com/microservices-architecture-for-mobile-application-development-part-i-20b4f4089a24>
- <https://hackernoon.com/microservices-architecture-for-mobile-application-development-part-ii-1a68db3aa438>

Esempi di implementazione di microservizi in Java