Luca Cabibbo

Architettura dei Sistemi Software

Microservizi e gestione dei dati

dispensa asw560

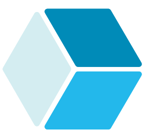
ottobre 2024

Code is easy; state is hard.

Edson Yanaga

- Fonti

- Ford, N. and Richards, M. **Fundamentals of Software Architecture**: An Engineering Approach. O'Reilly, 2020.
- Ford, N., Richards, M., Sadalage, P. and Dehghani, Z. **Software Architecture: The Hard Parts**: Modern Trade-Off Analyses for Distributed Architectures. O'Reilly, 2021.
- Richardson, C. **Microservices Patterns**: With examples in Java. Manning, 2019.
- Vernon, V. **Implementing Domain-Driven Design**. Addison-Wesley, 2013.

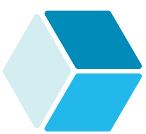


- Obiettivi e argomenti

- Obiettivi
  - presentare alcuni pattern per la gestione dei dati nei sistemi distribuiti, che possono essere applicati nell'architettura a microservizi
- Argomenti
  - introduzione
  - microservizi e basi di dati
  - API Composition
  - CQRS (Command-Query Responsibility Segregation)
  - Event Sourcing
  - Saga
  - discussione

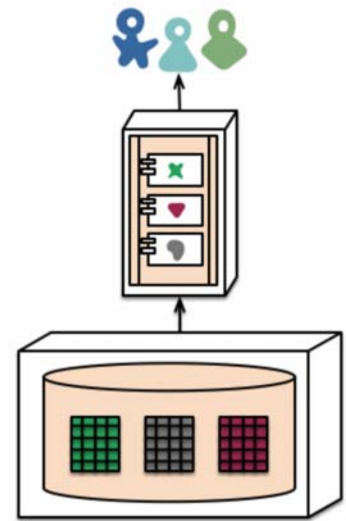
* Introduzione

- Questa dispensa discute l'organizzazione e la gestione dei dati nell'architettura a microservizi – più precisamente, la gestione dei dati persistenti e delle basi di dati
  - evidentemente, la gestione dei dati non deve pregiudicare la possibilità di raggiungere gli obiettivi di interesse dell'architettura a microservizi – in particolare, prestazioni, scalabilità, disponibilità e modificabilità di una singola applicazione
  - inoltre, la modalità di gestione dei dati non dovrebbe contravvenire i principi di progettazione dell'architettura a microservizi



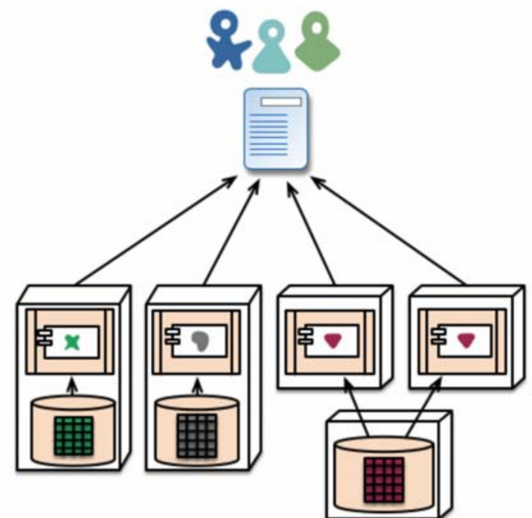
* Microservizi e basi di dati

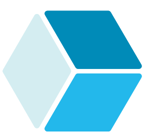
- Come organizzare i dati e le basi di dati nell'architettura a microservizi?
  - una possibilità è che tutti i microservizi accedano a una base di dati condivisa monolitica (pattern Shared Database [POSA])
  - questa soluzione è però in genere sconsigliata
    - perché induce un accoppiamento alto tra microservizi – i microservizi sono accoppiati, tramite la base di dati, agli altri microservizi e, in qualche modo, alle loro strutture di dati interne
    - inoltre, una base di dati monolitica non scala facilmente
  - piuttosto, una soluzione di solito preferibile consiste nell'operare anche una decomposizione della base di dati



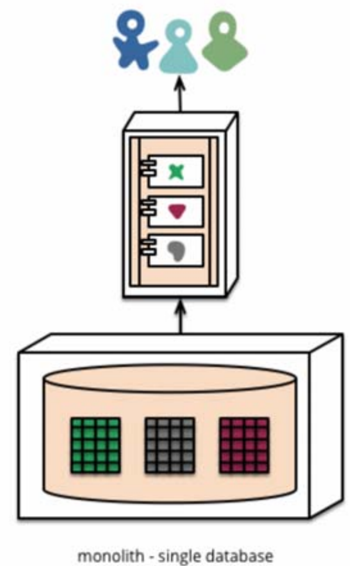
Microservizi e basi di dati

- I microservizi derivano in genere da una decomposizione di un'applicazione inizialmente monolitica – che opera su una base di dati inizialmente monolitica
  - la decomposizione in microservizi va applicata sia sulle funzionalità dell'applicazione che sui dati su cui essi operano
  - in genere è preferibile che ciascun microservizio gestisca i propri dati persistenti in una propria base di dati "privata" – che può essere realizzata con la tecnologia più opportuna per quel microservizio
  - questo sostiene accoppiamento basso e autonomia dei microservizi
  - inoltre può consentire migliori opportunità per la scalabilità e la disponibilità



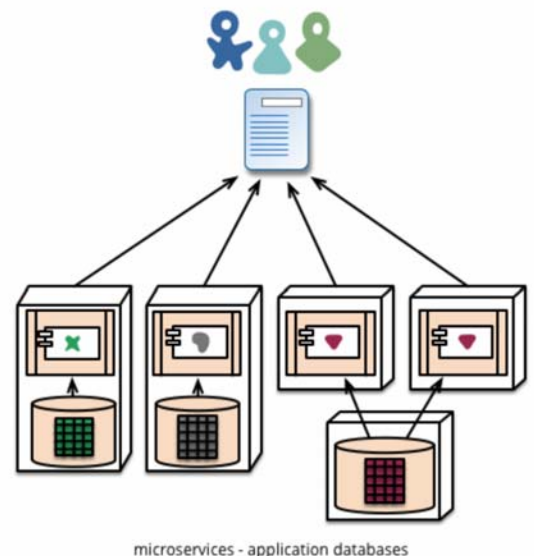


Soluzioni a confronto

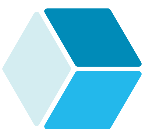


- Base di dati monolitica condivisa
  - tutti i microservizi accedono a un'unica base di dati condivisa – non viene effettuata nessuna decomposizione della base di dati
  - vantaggi
    - semplice da realizzare
    - consistenza dei dati basata su transazioni
  - inconvenienti
    - problemi di accoppiamento e di autonomia – può essere difficile l'aggiornamento dei microservizi
    - problemi di scalabilità e di affidabilità
    - i diversi microservizi non sono autonomi nella possibilità di ottimizzare l'accesso ai dati

Soluzioni a confronto



- Una base di dati per microservizio
  - ciascun microservizio accede a una propria base di dati "privata"
  - vantaggi
    - offre migliori opportunità per accoppiamento, scalabilità e possibilità di ottimizzazione
  - inconvenienti (problemi che solleva)
    - alcuni dati devono essere condivisi tra più microservizi
    - richiede una gestione opportuna dell'accesso (interrogazioni e aggiornamenti) a dati gestiti da diversi microservizi
    - è in genere possibile solo una consistenza debole dei dati (eventual consistency) – perché l'uso di transazioni distribuite sincrone non è compatibile con la scalabilità

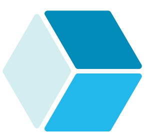


Decomposizione dei dati

- Nella decomposizione di un'applicazione inizialmente monolitica – con una base di dati inizialmente monolitica – anche la base di dati deve essere decomposta in modo opportuno
  - questa decomposizione non deve pregiudicare la possibilità di raggiungere gli obiettivi dell'architettura a microservizi – in particolare, prestazioni, scalabilità, disponibilità e modificabilità
  - inoltre, la modalità di gestione dei dati non dovrebbe contravvenire i principi di progettazione dell'architettura a microservizi
  - ad es., idealmente, ciascun microservizio dovrebbe accedere e modificare solo dati locali – per evitare problemi di disponibilità e di scalabilità

Decomposizione dei dati

- Come operare la decomposizione dei dati nell'architettura a microservizi?
  - in pratica, una scelta comune è che ciascun microservizio sia responsabile della gestione di uno o più tipi di aggregati (pattern Aggregates [DDD])

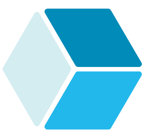


Decomposizione dei dati

- Pattern *Entities* [DDD], *Value Objects* [DDD] e *Aggregates* [DDD]
  - un'entità rappresenta un concetto primario del dominio, i cui oggetti sono caratterizzati da una continuità durante il loro ciclo di vita e da un'identità immutabile – il valore degli altri attributi può invece variare
    - ad es., **Restaurant**, **Customer** e **Order**
  - un oggetto valore è una classe di oggetti secondaria del dominio, in genere senza un'esistenza e un'identità autonoma, utile per esprimere ulteriori informazioni di oggetti entità
    - ad es., **MenuItem** e **OrderLine**
  - un aggregato è un gruppo di oggetti di dominio correlati (entità e oggetti valore) che vanno trattati come un'unità (di accesso e di manipolazione) – un albero o sottografo di oggetti, con radice in un'entità
    - ad es., un **Order** con le sue **OrderLine**

Decomposizione dei dati

- Come operare la decomposizione dei dati nell'architettura a microservizi?
  - un'altra scelta progettuale comune è l'utilizzo di tecnologie non relazionali (NoSQL) per la gestione dei dati
    - possono sostenere scalabilità e disponibilità meglio delle tecnologie relazionali
    - sono in genere compatibili con un'organizzazione dei dati basata su aggregati e l'accesso ai dati basato su aggregati
    - supportano l'accesso efficiente a singole istanze di aggregati
      - ma in genere non supportano l'esecuzione di interrogazioni per correlare più aggregati
    - supportano l'aggiornamento transazionale di singole istanze di aggregati
      - ma non supportano l'aggiornamento transazionale di più istanze di aggregati

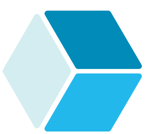


Decomposizione dei dati

- Come operare la decomposizione dei dati nell'architettura a microservizi?
  - inoltre, non è in genere una buona idea che i dati vengano "partizionati" tra i diversi microservizi – senza ridondanze né sovrapposizioni tra i dati gestiti da ciascun microservizio
    - in questo caso, infatti, quando un microservizio A ha bisogno accedere a dati "privati" di un altro microservizio B, dovrebbe farlo effettuando un'invocazione remota a B – ma questo può causare problemi di scalabilità e di disponibilità

Decomposizione dei dati

- Come operare la decomposizione dei dati nell'architettura a microservizi?
  - dunque, in genere i dati vengono "decomposti" tra i diversi microservizi (e non semplicemente "partizionati") – con possibili ridondanze e sovrapposizioni tra i dati gestiti da ciascun microservizio (e tra i relativi aggregati) – ma in modo tale che ciascun microservizio acceda e modifichi direttamente solo dati locali
    - ad es., un microservizio potrebbe essere responsabile di gestire alcuni aggregati e supportare i loro aggiornamenti – mentre un altro microservizio potrebbe essere invece responsabile di supportare delle interrogazioni che combinano i dati di aggregati gestiti da altri microservizi
  - come gestire questo tipo di decomposizione, supportando interrogazioni e aggiornamenti in modo flessibile ed efficiente, e fornendo anche un livello adeguato di consistenza dei dati?

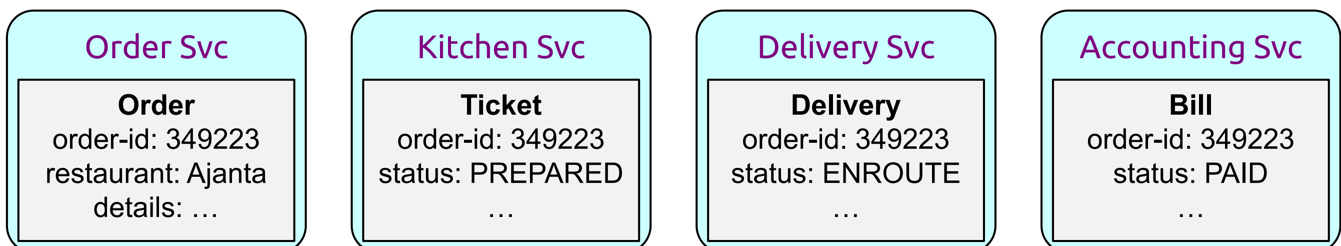


Pattern per la gestione dei dati

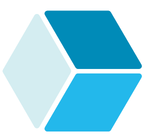
- Come gestire questo tipo di decomposizione, supportando interrogazioni e aggiornamenti in modo flessibile ed efficiente, e fornendo anche un livello adeguato di consistenza dei dati?
  - ci possono aiutare alcuni pattern per la gestione dei dati nei sistemi distribuiti (definiti prima dei microservizi), tra cui
    - API Composition
    - Command-Query Responsibility Segregation (CQRS)
    - Event Sourcing (ES)
    - Saga
  - nel seguito, supponiamo che ciascun microservizio abbia la responsabilità di gestire i dati di uno o più tipi di aggregati (con possibili sovrapposizioni parziali tra di essi)
    - obiettivo è decomporre i dati dell'intera applicazione – con riferimento a un insieme di aggiornamenti e di interrogazioni di interesse

* API Composition

- Consideriamo prima la gestione delle interrogazioni
  - la nostra ipotesi è che ciascun microservizio sia responsabile di gestire i dati di uno o più tipi di aggregati, di interesse per quello specifico microservizio

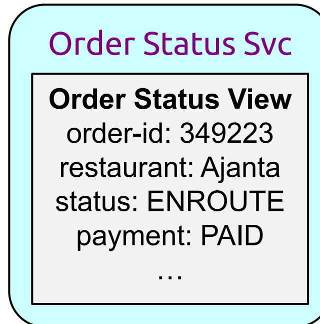


  - alcune interrogazioni potranno essere soddisfatte da un singolo microservizio
    - ad es., trovare un ordine con il dettaglio dei prodotti ordinati
  - altre interrogazioni però potrebbero essere interessate a dati gestiti da più microservizi
    - ad es., trovare un ordine con tutte le sue informazioni



API Composition

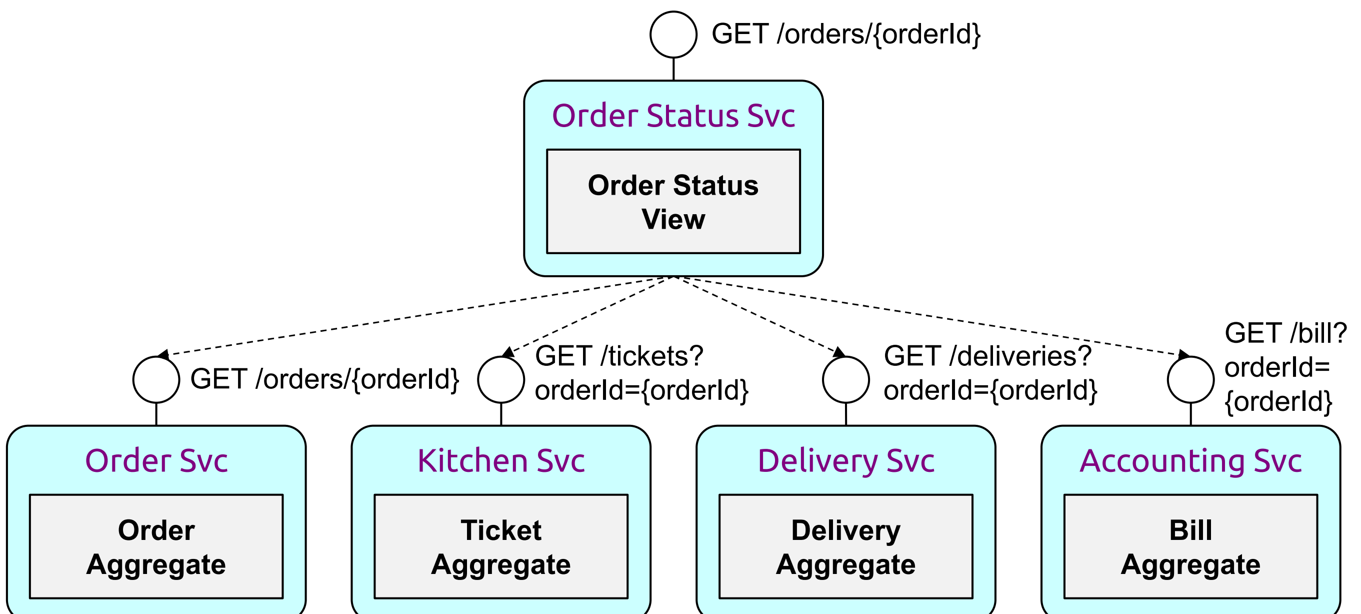
- Supponiamo di voler realizzare un microservizio in grado di restituire tutte le informazioni sullo stato di un ordine
  - ad es., informazioni di base sull'ordine, nonché informazioni sullo stato dell'ordine, della sua consegna e del suo pagamento

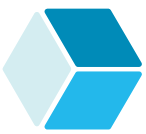


  - in un'applicazione monolitica, con una base di dati relazionale monolitica, questa interrogazione potrebbe essere realizzata da una singola interrogazione SQL, che fa il join di diverse tabelle
    - ma come realizzarla in un'applicazione a microservizi?

API Composition

- Una possibile soluzione è data dal pattern *API Composition*
  - implementa un'interrogazione che richiede dati da più microservizi come una vista non materializzata, che realizza interrogando separatamente ciascun microservizio di interesse mediante la sua API e combinandone i risultati



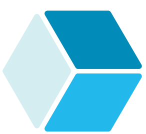


Conseguenze

- Vantaggi
  - semplice e intuitivo
- Inconvenienti
  - penalizzazione nelle prestazioni
    - dovuta all'invocazione di molti servizi e all'esecuzione di molte interrogazioni
  - rischio di una disponibilità ridotta
    - se la composizione avviene in modo sincrono
  - non garantisce una consistenza forte dei dati
    - i dati restituiti da un'interrogazione potrebbero essere inconsistenti – ad es., perché i dati di interesse potrebbero variare durante l'esecuzione delle diverse parti dell'interrogazione

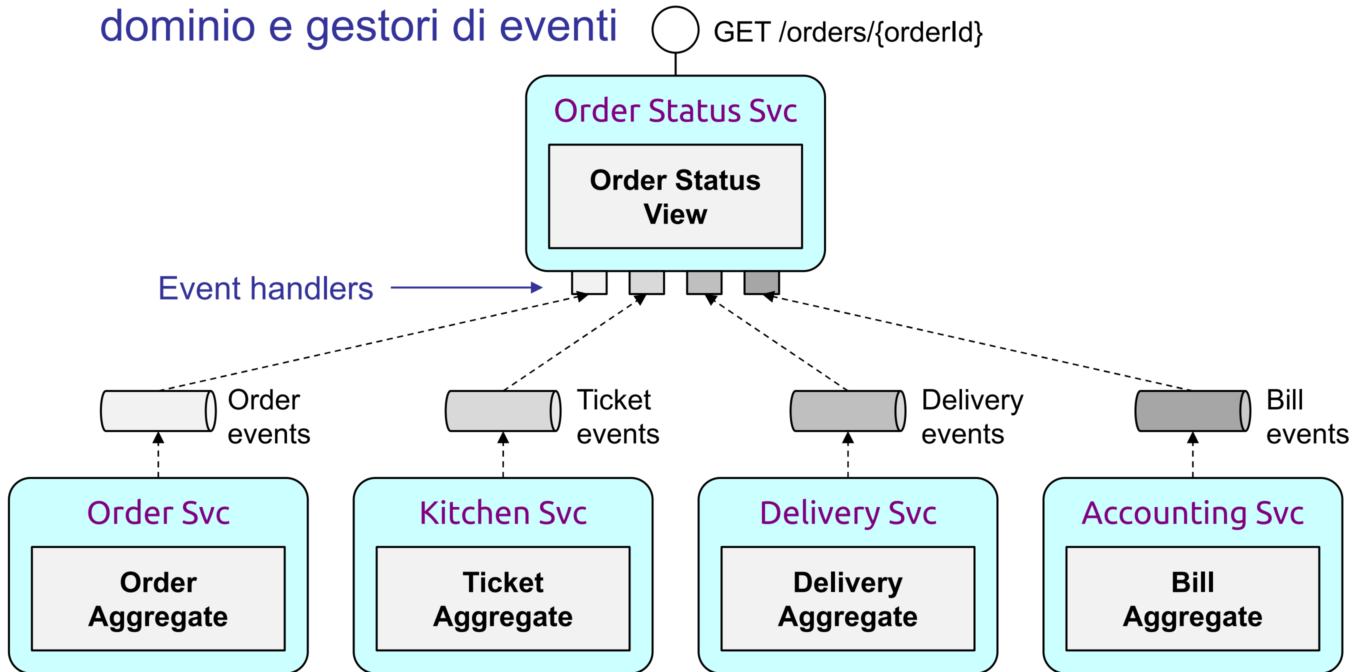
* CQRS (Command-Query Responsibility Segregation)

- Nel pattern API Composition, un'interrogazione viene gestita come una vista non materializzata, e calcolata su richiesta
  - una possibile alternativa consiste nel gestire l'interrogazione di interesse come una vista materializzata, ovvero pre-calcolata e sempre aggiornata
  - intuitivamente, l'idea è di ovviare (almeno in parte) agli inconvenienti del pattern API Composition, gestendo i dati dei diversi microservizi in modo parzialmente replicato e dunque ridondante
  - bisogna capire come gestire questa vista materializzata, se possibile in modo asincrono



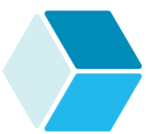
CQRS

- Una soluzione a questo problema è data dal pattern *CQRS*
  - implementa un'interrogazione che richiede dati da più microservizi come una vista materializzata, che realizza replicando i dati dai microservizi di interesse mediante eventi di dominio e gestori di eventi



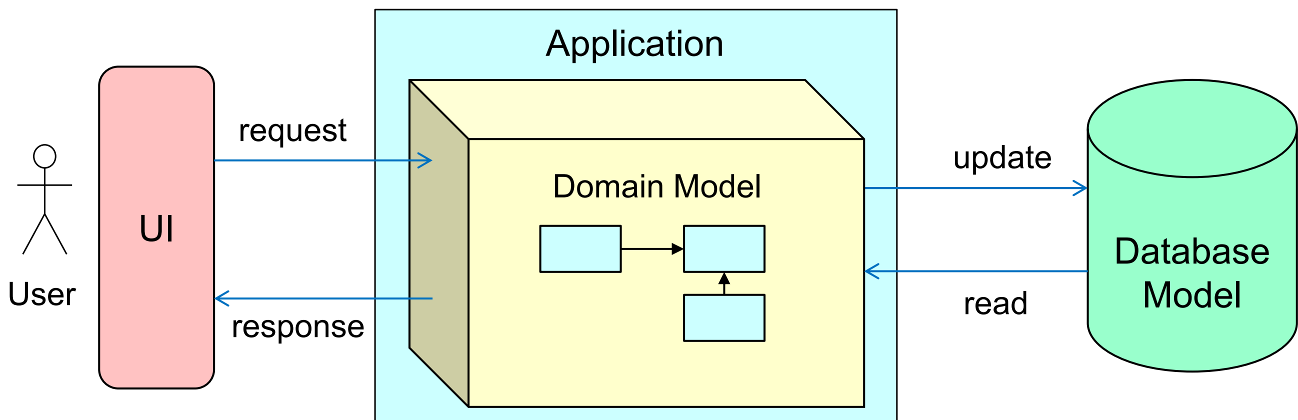
Verso il pattern CQRS

- La formulazione originale del pattern CQRS (che è stato definito ben prima dei microservizi) propone di sostituire una singola base di dati usata sia per gli aggiornamenti che per le interrogazioni con due (o più) basi di dati separate per gli aggiornamenti e le interrogazioni – in modo da poterle gestire e ottimizzare separatamente
  - esaminiamo ora la trasformazione proposta da CQRS



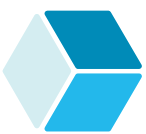
Approccio tradizionale (non CQRS)

- Consideriamo un'applicazione o componente software organizzato nel seguente modo (non è basato su CQRS)
  - un client (un utente o un'altra applicazione o componente) fa una richiesta al componente
  - il componente elabora la richiesta, leggendo e aggiornando (se serve) i dati nella base di dati – e poi restituisce una risposta al suo client



Approccio tradizionale (non CQRS)

- Questa organizzazione è basata su
  - un singolo modello per la gestione dei dati dell'applicazione o componente, che implementa tutte le operazioni di interesse (sia di aggiornamento che di interrogazione)
  - una singola base di dati, per gestire la persistenza dei dati di questo singolo modello
- Per alcuni componenti o applicazioni questa organizzazione è inadeguata – ad esempio
  - se bisogna gestire aggiornamenti complessi oppure interrogazioni complesse
  - se sono necessarie viste multiple sui dati – ad es., se gli utenti di un sistema di e-commerce vogliono consultare i loro ordini, ma la direzione vuole analizzare gli ordini, ad es., per prodotto
  - se c'è uno sbilanciamento tra il carico degli aggiornamenti e delle interrogazioni, che si vogliono ottimizzare separatamente

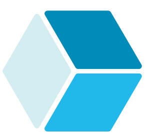


Il principio CQS

- Il pattern architetturale CQRS deriva da *CQS* (*Command-Query Separation*) – è un principio di progettazione orientata agli oggetti che suggerisce che ciascuna operazione sia un comando oppure un'interrogazione – ma non entrambe le cose
  - un *comando* (*command*) è un'operazione che modifica i dati, provocando degli effetti collaterali, ma senza restituire un valore
  - un'*interrogazione* (*query*) è invece un'operazione che si limita a leggere i dati, senza modificarli, e restituisce un valore
  - un sistema o un progetto in cui ci sono solo comandi e interrogazioni può presentare dei vantaggi
    - ad es., può essere più facile da comprendere
    - ad es., se in una classe i metodi sono solo comandi o solo interrogazioni
  - per questo, CQS sconsiglia operazioni che siano contemporaneamente sia comandi che interrogazioni

CQRS

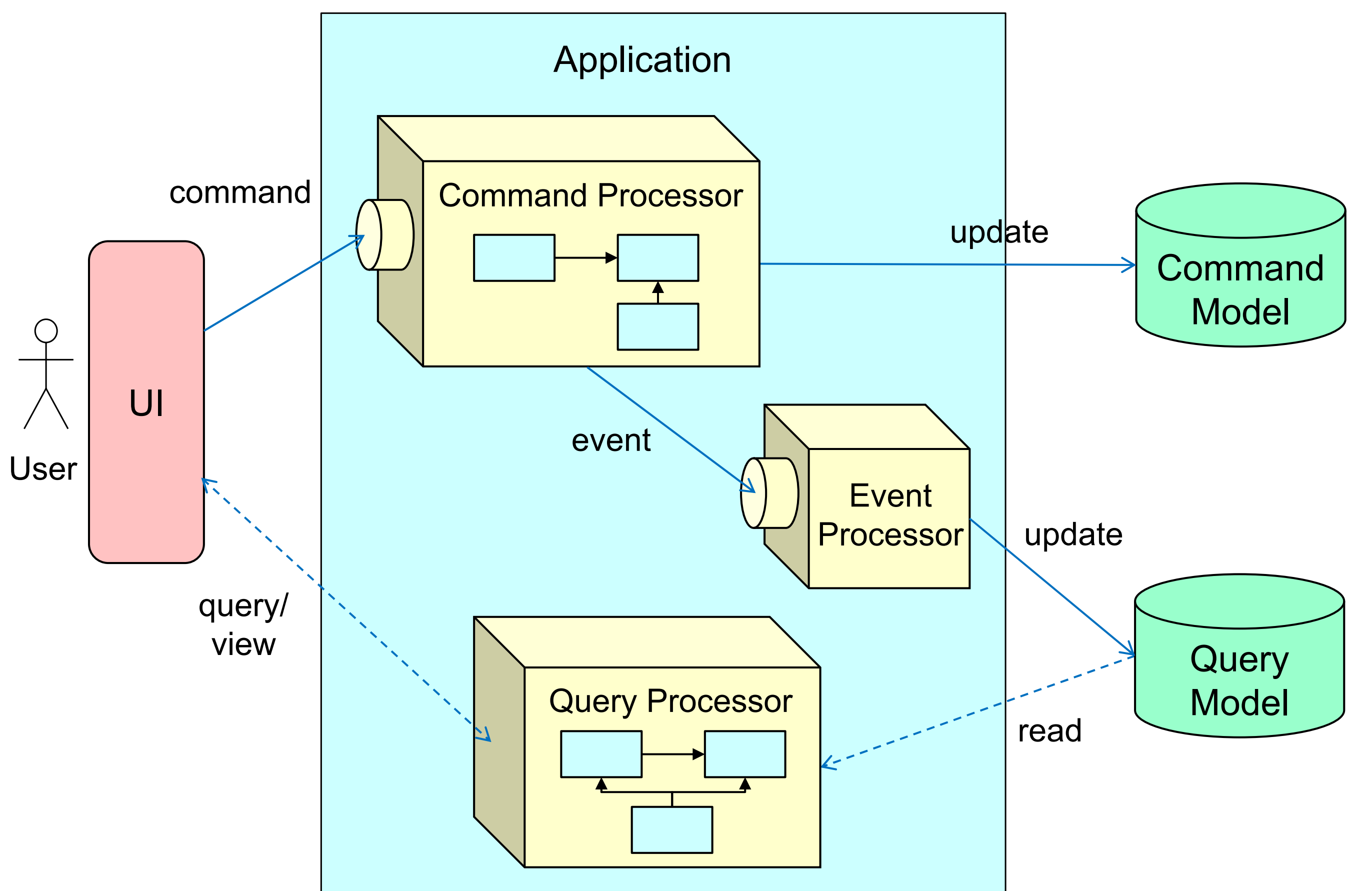
- Il pattern architetturale *Command-Query Responsibility Segregation* (comunemente abbreviato in *CQRS*) è una versione architetturale e più drastica del principio CQS
  - CQRS suggerisce una separazione forte tra l'elaborazione dei comandi e quella delle interrogazioni, con
    - un elemento responsabile dell'elaborazione dei comandi (*command processor*) e un elemento separato responsabile dell'elaborazione delle interrogazioni (*query processor*)
    - un modello e una base di dati per la gestione dei comandi (*command model* o *write model*) e un modello e una base di dati separata per le interrogazioni (*query model* o *read model*)
    - un elemento che si occupa di propagare gli aggiornamenti causati dai comandi anche alla base di dati per le interrogazioni (*event processor*)

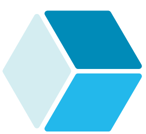


CQRS

- CQRS prevede di solito l'utilizzo di basi di dati separate per i comandi e per le interrogazioni
  - la base di dati per i comandi (*command model* o *write model*) è di solito normalizzata e senza ridondanze – per ottimizzare l'esecuzione dei comandi
  - la base di dati per le interrogazioni (*query model* o *read model*) è invece di solito denormalizzata e con ridondanze – per ottimizzare delle specifiche interrogazioni e viste
    - la base di dati per le interrogazioni definisce una sorta di vista materializzata sulla base di dati per i comandi
    - la base di dati per le interrogazioni può essere in genere (ri)generata a partire dalla sola base di dati per i comandi

CQRS



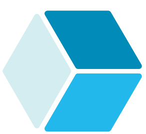


CQRS

- Dinamica (comandi)
  - un client (un utente o un'altra applicazione o componente) fa una richiesta al componente sotto forma di comando – il comando viene messo in una coda ed elaborato in modo asincrono
  - il command processor elabora il comando, aggiornando la base di dati per i comandi (in genere in modo sincrono e transazionale) – dopo di che, invia un evento di notifica per ciascuno degli aggiornamenti che ha effettuato
  - l'event processor, quando riceve la notifica di un evento di aggiornamento, in modo asincrono, aggiorna di conseguenza la base di dati per le interrogazioni
    - questi aggiornamenti avvengono di solito in modo asincrono e <u>non</u> transazionale rispetto all'aggiornamento della base di dati per i comandi – attenzione alla consistenza dei dati!

CQRS

- Dinamica (interrogazioni)
  - il client può anche fare la richiesta di un'interrogazione
  - il query processor esegue l'interrogazione (di solito in modo sincrono) accedendo solo alla base di dati per le interrogazioni (ma non a quella per i comandi)

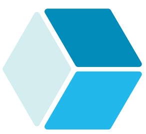


CQRS

- Alcune varianti della struttura proposta
  - la base di dati per le interrogazioni è una replica separata (e in sola lettura) della base di dati per i comandi – in questo caso lo scopo è solo quello di separare il carico di lavoro delle due basi di dati
  - anziché un singolo command processor (responsabile di molti comandi) vengono usati più command processor – ma di solito viene comunque usata una singola base di dati per i comandi
  - anziché un singolo query processor (responsabile di tutte le interrogazioni e viste) vengono usati più query processor, di solito ciascuno con il suo event processor – è anche possibile avere una base di dati distinta per ciascun query processor
  - talvolta, invece, viene usata una singola base di dati, utilizzata sia per i comandi che per le interrogazioni

CQRS e microservizi

- Il pattern CQRS può essere applicato nell'architettura a microservizi
  - un microservizio può essere responsabile di gestire
    - il command model per uno o più tipi di aggregati
    - il query model per uno o più tipi di aggregati
    - in genere, un microservizio gestisce command model e query model per tipi di aggregati <u>differenti</u>
  - quando un microservizio esegue un comando (in genere per aggiornare un singolo aggregato), allora notifica i relativi eventi di aggiornamento a tutti i microservizi interessati a quel tipo di aggregato
    - questi microservizi aggiornano di conseguenza i propri query model
  - in questo modo, è possibile mantenere allineati dati di interesse relativi a più aggregati in microservizi diversi

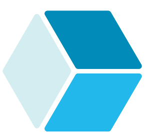


Conseguenze

- Vantaggi
  - consente un’ottimizzazione separata per i comandi e per le interrogazioni
  - separazione degli interessi e riduzione della complessità – la logica dei comandi è separata e indipendente da quella delle interrogazioni – anche le interrogazioni sono separate dai comandi
  - interrogazioni o viste diverse possono essere gestite da elementi distinti, anche in basi di dati differenti – ciascuna interrogazioni o vista può essere ottimizzata separatamente
  - flessibilità – è semplice aggiungere nuove interrogazioni e nuove viste sui dati
  - abilita l’esecuzione efficiente di interrogazioni nell’architettura a microservizi

Conseguenze

- Inconvenienti
  - non garantisce una consistenza forte dei dati
    - è possibile l’inconsistenza dei dati tra la base di dati per i comandi e quella per le interrogazioni – a causa della comunicazione asincrona, è possibile che un’interrogazione restituisca dati non aggiornati rispetto a quelli presenti nella base di dati per i comandi
  - questo pattern costituisce uno scostamento dall’approccio tradizionale di modellazione dei dati e del dominio – per alcuni team può essere difficile da implementare
  - lo sforzo richiesto dalla separazione tra comandi e interrogazioni potrebbe non essere giustificato nei sistemi più semplici

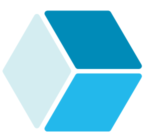


* Event Sourcing

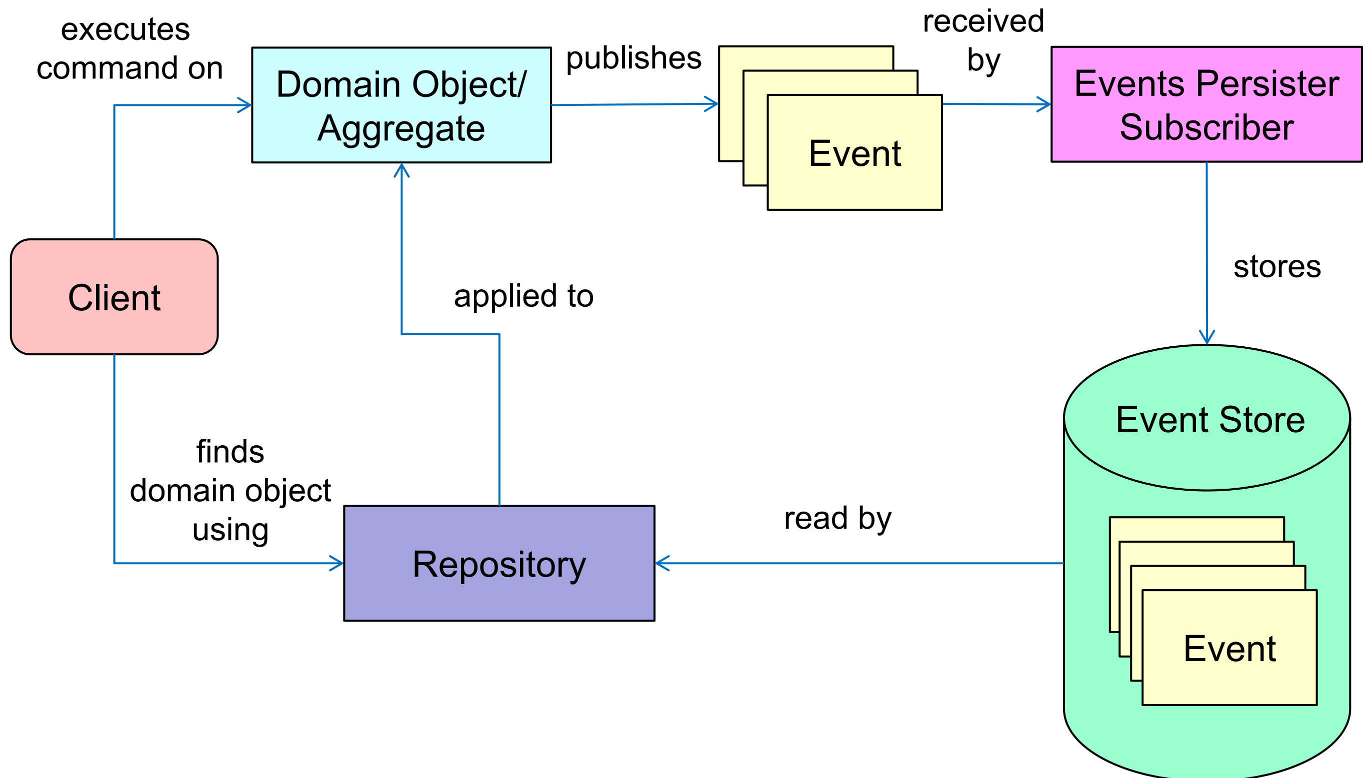
- Molte applicazioni o componenti software memorizzano solo lo stato corrente dei dati che gestiscono – tuttavia, talvolta è necessario o utile anche tenere traccia di tutti i cambiamenti che sono avvenuti sui dati, che hanno portato al loro stato corrente
  - una possibile soluzione consiste nel modificare il modello dei dati in modo che ciascun oggetto di dominio sia responsabile di conoscere, oltre al proprio stato, anche i suoi cambiamenti e chi li ha effettuati
    - tuttavia, questa soluzione è complicata, perché mischia la gestione dello stato corrente con quella dei cambiamenti
  - una soluzione alternativa consiste invece di gestire e memorizzare separatamente i cambiamenti dei dati (dai dati correnti)

Event Sourcing

- Il pattern architetturale *Event Sourcing* ("specificare/ottenere eventi", talvolta abbreviato in *ES*) suggerisce di gestire e memorizzare separatamente tutti i cambiamenti dei dati nella forma di eventi di dominio (sulla base del pattern *Domain Events* di DDD) in un *event store*
  - quando si verifica un cambiamento in un oggetto di dominio (o di un aggregato), all'event store viene aggiunto un evento relativo al cambiamento – i dati nell'event store vengono solo appesi, e mai modificati né cancellati
  - quando si vuole trovare lo stato corrente di un oggetto di dominio (o di un aggregato), ad es. tramite un repository, quest'ultimo cerca nell'event store tutti gli eventi relativi a quell'oggetto e poi li combina, per calcolare appunto lo stato corrente dell'oggetto

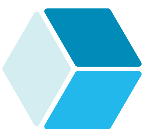


Event Sourcing



Conseguenze

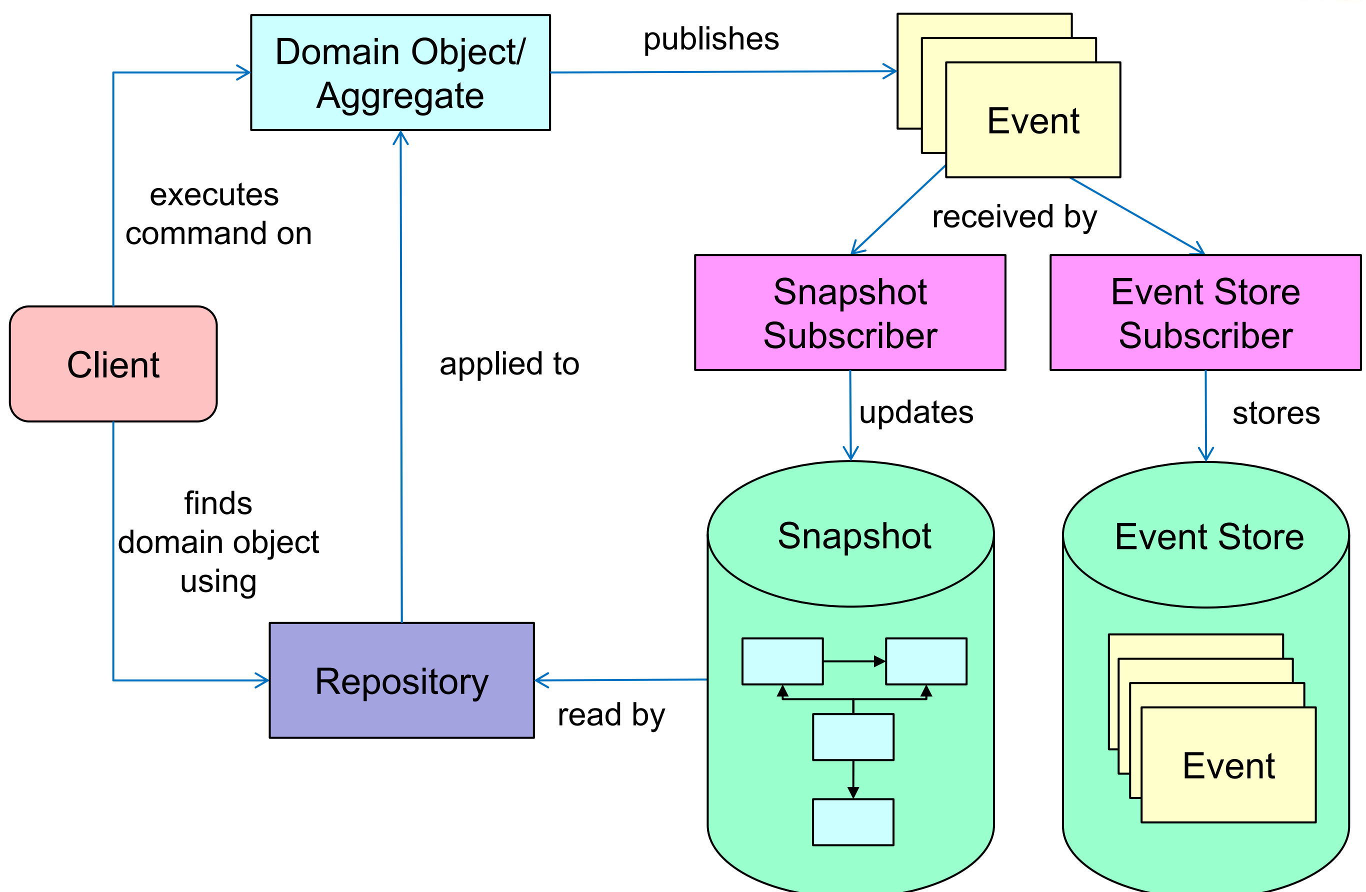
- Vantaggi
  - semplificazione nella gestione dei comandi
  - prestazioni e scalabilità – i comandi vengono eseguiti semplicemente memorizzando (appendendo) gli eventi all'event store
  - riduzione della complessità – lo stato degli oggetti di dominio è modellato separatamente dalla loro storia
  - i dati possono essere rappresentati alla granularità più piccola possibile (senza perdita di informazioni) – è possibile calcolare lo stato corrente degli oggetti di dominio rielaborando tutti gli eventi (nonché ogni stato passato, in un qualunque istante di tempo) – può essere possibile anche calcolare dati che non erano stati inizialmente previsti

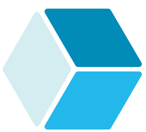


Conseguenze

- Inconvenienti
  - questo pattern costituisce uno scostamento dall'approccio tradizionale di modellazione dei dati e del dominio – per alcuni team può essere difficile da implementare
  - la ricostruzione dello stato degli oggetti di dominio può essere onerosa
    - questo problema può essere risolto da un'applicazione congiunta di Event Sourcing e CQRS

CQRS + Event Sourcing



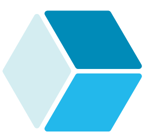


CQRS + Event Sourcing

- I pattern CQRS ed Event Sourcing si possono combinare facilmente
  - l'event store memorizza gli eventi di dominio – e funge da command model
  - inoltre, c'è anche una base di dati per lo stato corrente degli oggetti di dominio – questa base di dati può essere considerata una vista materializzata sull'event store
  - per gestire la base di dati per gli oggetti di dominio (considerata un query model in CQRS) viene introdotto un event processor che riceve gli eventi di dominio e li utilizza per aggiornare questa base di dati

* Saga

- Consideriamo ora la gestione delle transazioni
  - una *transazione* è un'operazione di aggiornamento dei dati
    - in genere, si ritiene che le transazioni debbano avvenire in modo atomico, consistente, isolato e persistente (transazioni "acide")
    - tuttavia, talvolta è accettabile rilassare le proprietà di isolamento e consistenza – questo viene talvolta fatto anche nelle basi di dati relazionali
  - le transazioni che riguardano l'aggiornamento di una singola istanza di aggregato in un singolo microservizio non sono problematiche
  - come gestire però transazioni che riguardano l'aggiornamento di più istanze di aggregati, gestite da microservizi diversi?

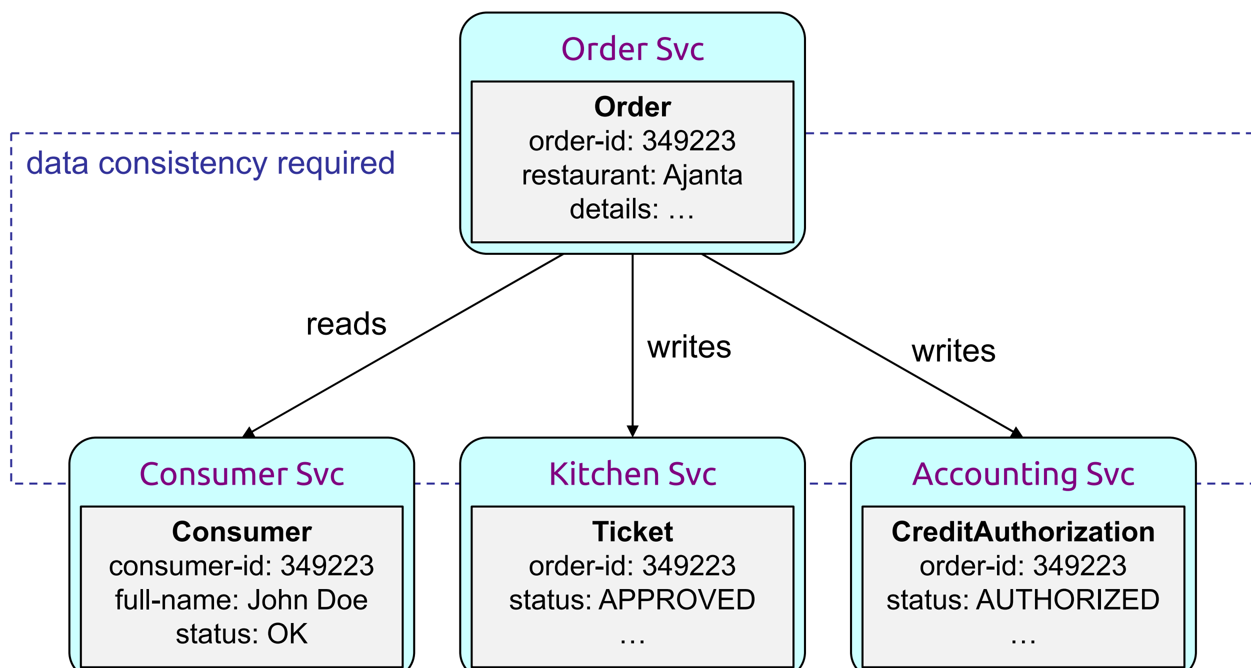


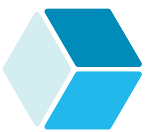
Esempio

- Come transazione di esempio, consideriamo l'operazione createOrder del servizio Order Service – che deve
  - creare un ordine Order – servizio Order Service
  - verificare che il consumatore possa effettuare l'ordine – servizio Consumer Service
  - creare una comanda Ticket per l'ordine – servizio Kitchen Service
  - ottenere l'autorizzazione al pagamento con carta di credito e creare un CreditAuthorization – servizio Accounting Service
  - questa transazione deve essere atomica nel senso che, ad es., se l'autorizzazione al pagamento non viene concessa, allora devono essere annullati sia l'ordine che la comanda

Esempio

- Come transazione di esempio, consideriamo l'operazione createOrder del servizio Order Service



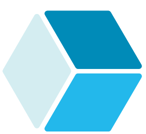


Transazioni distribuite

- Come gestire la transazione distribuita createOrder del servizio Order Service?
  - in un'applicazione monolitica, con una base di dati relazionale monolitica, questa operazione potrebbe essere realizzata come una singola transazione che racchiude diverse istruzioni SQL
  - in un'applicazione a microservizi, invece, potrebbe essere realizzata come una transazione distribuita sincrona, di tipo 2PC, che racchiude diverse transazioni "locali", ognuna eseguita da un singolo microservizio
    - il protocollo *2PC* (*Two-Phase Commit*) è basato su un coordinamento sincrono tra i diversi partecipanti a una transazione distribuita, organizzato appunto in due fasi: richiesta/preparazione e validazione
    - tuttavia, questa soluzione solleva problema di scalabilità e di riduzione della disponibilità – inoltre, potrebbe non essere compatibile con l'uso di basi di dati non relazionali

Transazioni distribuite

- Come gestire la transazione distribuita createOrder del servizio Order Service?
  - fortunatamente, per i sistemi distribuiti esistono altre soluzioni per realizzare transazioni distribuite – in modo da garantire atomicità, persistenza e un certo livello di consistenza dei dati
    - in particolare, le saga sono un meccanismo per mantenere la consistenza dei dati nell'architettura a microservizi senza usare transazioni distribuite sincrone (2PC)

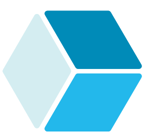


Saga

- Una possibile soluzione per la gestione delle transazioni nell'architettura a microservizi è data dal pattern *Saga*
  - implementa un'operazione che deve aggiornare aggregati in più microservizi come una sequenza di transazioni locali
  - ciascuna di queste transazioni locali aggiorna una singola istanza di aggregato in un singolo microservizio
  - le diverse transazioni locali sono coordinate usando dei messaggi asincroni
    - il completamento di ciascuna transazione locale abilita ("trigger") l'esecuzione della prossima transazione locale
    - se invece una transazione locale fallisce, le precedenti transazioni locali della saga vanno in genere annullate mediante delle opportune transazioni compensatrici

Esempio

- Ecco una saga per l'operazione createOrder
  1. Order Service :: createOrder – crea un ordine Order nello stato APPROVAL_PENDING
  2. Consumer Service :: verifyConsumerDetails – verifica che il consumatore possa effettuare l'ordine
  3. Kitchen Service :: createTicket – verifica i dettagli dell'ordine, e crea una comanda Ticket per l'ordine nello stato CREATE_PENDING
  4. Accounting Service :: authorizeCreditPayment – ottieni l'autorizzazione al pagamento dell'ordine con carta di credito
  5. Kitchen Service :: approveTicket – cambia lo stato del Ticket per l'ordine in AWAITING_ACCEPTANCE
  6. Order Service :: approveOrder – cambia lo stato dell'Order in APPROVED

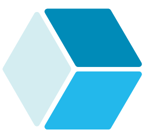


Saga e transazioni locali

- Le saga sono composte da una sequenza di transazioni locali
  - alcune transazioni locali sono in sola lettura (ad es., verifyConsumerDetails)
  - inoltre, alcune transazioni locali non possono fallire (ad es., approveTicket e approveOrder) – nel caso, vanno ripetute
  - l'ultima transazione locale di una saga che può fallire per motivi di business (ad es., authorizeCreditPayment) è chiamata la *transazione pivot*
  - le transazioni locali che effettuano aggiornamenti e che sono seguite da una transazione locale che può fallire richiedono una *transazione compensatrice*
    - ad es., la transazione createOrder richiede una transazione compensatrice rejectOrder, poiché verifyConsumerDetails o authorizeCreditPayment possono fallire
    - in modo analogo, la transazione createTicket richiede una transazione compensatrice rejectTicket

Coordinamento delle saga

- Il coordinamento di una saga riguarda la gestione del sequenziamento tra le transazioni locali e delle relative decisioni – e può avvenire in due modalità (entrambe in modo asincrono)
  - coreografia
    - la logica di coordinamento della saga è distribuita tra i servizi che partecipano alla saga
    - i servizi comunicano mediante la notifica asincrona di eventi
  - orchestrazione
    - la logica di coordinamento della saga è centralizzata in uno dei servizi (chiamato orchestratore)
    - il servizio orchestratore comunica con gli altri servizi partecipanti mediante l'invio asincrono di comandi
    - i servizi partecipanti comunicano l'esito delle proprie transazioni locali mediante la notifica asincrona di eventi
  - i messaggi devono essere scambiati in modo transazionale

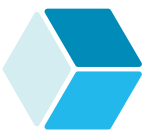


Conseguenze

- Vantaggi
  - atomicità e persistenza – le saga supportano l’aggiornamento atomico e persistente di più aggregati, gestiti da microservizi diversi
  - consistenza (debole) dei dati – le saga supportano un certo livello di consistenza debole dei dati, che è adeguato in molti contesti applicativi
  - scalabilità, disponibilità e modificabilità – l’utilizzo della comunicazione asincrona per il coordinamento tra i microservizi partecipanti sostiene scalabilità, disponibilità e accoppiamento debole tra i microservizi
  - complessità – l’utilizzo dell’orchestrazione (che pure ha degli inconvenienti) sostiene la separazione degli interessi e semplifica la logica di coordinamento delle transazioni

Conseguenze

- Inconvenienti
  - consistenza (forte) dei dati – le saga non garantiscono una consistenza forte dei dati
  - isolamento – non è garantito isolamento nell’esecuzione delle saga – sono di conseguenza possibili alcune anomalie dovute alla mancanza di isolamento
  - questo pattern costituisce uno scostamento dall’approccio tradizionale di modellazione dei dati e delle transazioni – per alcuni team può essere difficile da implementare
  - complessità – la comprensione e il monitoraggio delle saga è difficile (soprattutto in caso di utilizzo di una coreografia)



* Discussione

- Questa dispensa ha discusso la gestione dei dati persistenti (ovvero, delle basi di dati) nell'architettura a microservizi
  - nell'architettura a microservizi, ciascun microservizio è in genere responsabile dei dati persistenti relativi a uno o più tipi di aggregati
    - dunque, la decomposizione di un'applicazione inizialmente monolitica deve essere in genere accompagnata da un'opportuna decomposizione della relativa base di dati inizialmente monolitica
  - questa decomposizione dei dati deve essere accompagnata da un supporto flessibile alle interrogazioni e agli aggiornamenti, garantendo anche un livello adeguato di consistenza dei dati
    - questo obiettivo è sostenuto da alcuni pattern per la gestione dei dati nei sistemi distribuiti, che possono essere applicati anche nell'architettura a microservizi