Riassunto MicroServizi

Gabriele Naretto

2022 Sessione Gennaio-Febbraio

Contents

1 Introduzione

Dagli anni ‘90 il modello multistrato (multi-tier architecture) è stato considerato un pattern architetturale fondamentale per costruire un sistema software. Secondo tale modello le varie funzionalità software sono logicamente separate su più strati che comunicano tra di loro. Ogni strato comunica con gli strati adiacenti in modo diretto richiedendo e offrendo servizi. In effetti in questa architettura il sistema software, sia pure se logicamente suddiviso in strati, risulta essere un unico sistema monolitico. L’avvento e la diffusione del cloud computing, le pratiche di continuous delivery, l’approccio alla gestione della complessità del

software basato sul DDD (Domain-Driven Design), l'organizzazione agile delle aziende in team di sviluppo piccoli e autonomi (3-7 persone) sono il contesto in cui è emerso il modello dell'architettura a Microservizi.

2 Cosa sono i Microservizi ?

Definizione di Martin Fowler: *Lo stile architetturale a Microservizi è un approccio allo sviluppo di una singola applicazione come insieme di piccoli servizi, ciascuno dei quali viene eseguito da un proprio processo e comunica con un meccanismo snello, spesso una HTTP API.*

Definizione di Amazon AWS: *I microservizi sono un approccio per sviluppare e organizzare l'architettura dei software secondo cui quest'ultimi sono composti di servizi indipendenti di piccole dimensioni che comunicano tra loro tramite API ben definite. Questi servizi sono controllati da piccoli team autonomi. Le architetture dei microservizi permettono di scalare e sviluppare le applicazioni in modo più rapido e semplice, permettendo di promuovere l'innovazione e accelerare il time-to-market di nuove funzionalità.*

2.0.1 Autonomia

Ogni Microservizio è un'entità separata che viene generalmente pubblicata su una piattaforma oppure eseguita da uno processo di sistema ad hoc. La comunicazione tra i servizi avviene attraverso la rete al fine di garantire l'indipendenza tra i servizi ed evitare ogni forma di accoppiamento. Ogni Microservizio si propone all'esterno come una black-box, infatti espone solo un Application Programming Interface (API), astraendo rispetto al dettaglio di come le funzionalità sono implementate e dallo specifico linguaggio o tecnologia utilizzati. Ciò mira a far si che il cambiamento di ciascun Microservizio non abbia impatto sugli altri.

2.0.2 Riassuntino

Quindi detto in parole povere i Microservizi sono un insieme di backend ognuno a se, alcuni possono condividere il database ma idealmente ognuno dovrebbe avere il proprio. La cosa importante e che insieme questi Microservizi formino l'ecosistema dell'applicazione. Per fare un esempio prendiamo un applicazione per la gestione di prenotazioni, un Microservizio gestirà l'autenticazione degli utenti, un altro le prenotazioni e un altro ancora aggiornerà e cancellerà informazioni per gli utenti. Questi tre backend saranno sviluppati su tre progetti diversi e dovranno collaborare tra loro per far funzionare al meglio l'applicazione.

2.1 Differenze con SOA

- Le SOA tendono a usare Enterprise Service Bus (ESB) mentre i microservizi usano un sistema di messaggistica meno elaborato e semplice

- Le SOA sono composte da più linee di codice rispetto ai microservizi
- Le SOA pongono maggiore enfasi sulla riusabilità, mentre i microservizi si concentrano su disaccoppiare il più possibile
- Un cambiamento in SOAs richiede un intero cambiamento nell'applicazione monolitica
- Le SOA usano più spesso DB relazionali mentre i microservizi gravitano più verso i DB non relazionali (ricordiamo che non è un obbligo, ogni microservizio può usare un database che più si addice al suo compito)

2.2 Come possiamo ostare un Microservizio ?

- **Virtual Machine**
- **Platform as a Service**: quindi Cloud Computing
- **Containers**: quindi Docker e Kubernetes

3 Pro e Contro

3.1 Pro dei microservizi

- **Dominio incentrato sul Business**
- **Automazione**: Si possono usare più facilmente Tools per testare e sviluppare (per esempio Spring)
- **Velocizzare i tempi di rilascio del software e reagire velocemente alle esigenze del mercato**: In generale, nell'architettura a microservizi, ogni singolo servizio è autonomo rispetto agli altri, di conseguenza può raggiungere l'ambiente di produzione in modo indipendente dagli altri, senza che tale attività abbia effetti drammatici sul resto del sistema
- **Sperimentare più facilmente nuove tecnologie**: Molto spesso, la principale barriera per adottare una nuova tecnologia risiede nel rischio associato all'utilizzo di qualcosa di nuovo e con il quale si ha poca esperienza. Confinando questo rischio a una piccola porzione di un sistema software, che è possibile riscrivere in appena due settimane di lavoro, il rischio risulta molto contenuto e quindi è una sfida da accettare.
- **Migliori performance grazie all'utilizzo di tecnologie ad hoc**
- Resilienza: In un'architettura a microservizi, quando una componente non funziona non è automatico che tutto il sistema software smetta di funzionare. In molti casi è possibile isolare il problema e intervenire mentre il resto del sistema continua a funzionare, cosa non possibile in un'architettura monolitica.

- **Scalabilità**: In generale risulta molto più semplice ed economico scalare un microservizio rispetto a un sistema software monolitico di grandi dimensioni.
- **Facilità di deployment**: Con l'approccio a microservizi ogni singolo servizio può raggiungere l'ambiente di produzione in modo indipendente, sicché se si verifica un problema esso è facilmente isolato e possono essere intraprese azioni di rollback più velocemente.
- **Componibilità**: Tra le opportunità più interessanti dell'architettura a microservizi vi è la possibilità di riusare le funzionalità. Infatti è possibile che una stesso servizio venga utilizzato in modi differenti e per scopi diversi
- **Sostituibilità**: Quando un sistema software è organizzato a microservizi, il costo di sostituire un servizio con un altro più efficiente e migliore è molto contenuto poiché le modifiche consistono solo in un parte del sistema

3.2 Contro dei microservizi

- **Duplicazione dei Dati/Database decentralizzati**: la scomposizione in servizi si porta dietro anche una gestione del dato capace di soddisfare il funzionamento di ogni componente del software. Occorre dunque capire come partizionare un database che prevede accessi plurimi e non può più godere dei vantaggi esclusivi della centralizzazione. Ci sono pertanto pro e contro, ma in generale una maggior complessità da gestire, soprattutto nel caso di database articolati e di grandi dimensioni.
- **Comunicazione tra i singoli servizi**:a livello tecnico, più servizi vengono tra loro disaccoppiati rispetto a un blocco monolitico, più si genera una oggettiva esigenza di comunicazione per far dialogare l'applicazione nel suo insieme. Occorre quindi garantire bassi livelli di latenza e code di messaggi per instradare le comunicazioni con la maggior continuità possibile.
- **Scalabilità complessa**:se l'indipendenza dei servizi consente di scalarli in maniera ottimale, al tempo stesso garantire la disponibilità di un elevato livello di servizi costituisce un altro degli aspetti di complessità che va valutato con estrema attenzione quando si pensa all'applicazione nella sua interezza.
-

Da quanto fin qui trattato, potrebbe sembrare che l'architettura a microservizi risulti essere la manna dal cielo, non è così. Infatti non possiamo dimenticare di ribadire che utilizzare tanti servizi che dialogano tra di loro attraverso la rete, vuol dire di fatto avere a che fare con un sistema distribuito, con tutti i problemi dal caso. Si pensi, ad esempio, a cosa vuol dire autenticare un utente su un'architettura distribuita volendo garantire il single sign-on, oppure a come

gestire la mutua autenticazione tra i servizi che compongono il sistema software. E ancora: cosa vuol dire testare un sistema software di questo tipo? Infine, vengono fuori altre tematiche tipiche dei sistemi distribuiti come la gestione di situazioni in cui è necessario che il sistema software sia in grado di riconfigurarsi al volo quando una nuova istanza di un servizio viene creata e il resto della rete può automaticamente trovare essa e iniziare a comunicare con essa. Questo processo è chiamato service discovery.

4 Architetture dei microservizi

4.1 Comunicazione tra microservizi

Per poter "collaborare" i microservizi hanno bisogno di parlarsi tra loro, vediamo quali modi abbiamo per farlo.

- **Message bus**
- **Chiamate HTTP**: Tendenzialmente in modo Sincrono

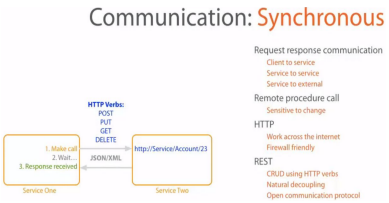

Figure 1: Comunicazione HTTP

- **Message Broke**: Quindi avremo Pubblicazione e iscrizione ai messaggi, Coda dei messaggi e molto altro il tutto (tendenzialmente) in modo Asincrono

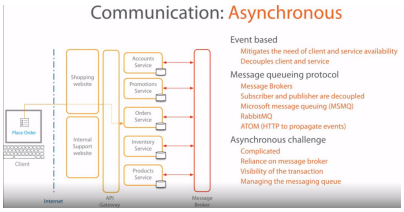

Figure 2: Comunicazione Message Broker

4.1.1 Broker Composition pattern

Pattern che utilizza un message broker per far comunicare tra loro i microservizi, il principale esponente di questa tipologia è il pattern SAGA

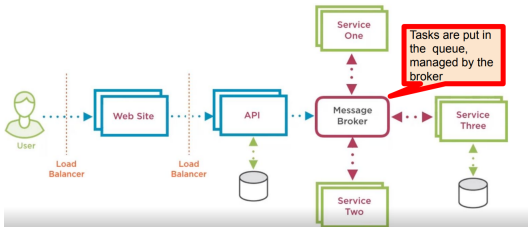

Figure 3: Broker Composition pattern

4.1.2 Aggregate Composition pattern

Il pattern avrà come obbiettivo quello di aggregare tutti i dati che servono per il microservizio, di norma questa cosa può farla il sito, ma nelle versioni più evoluto di questo pattern avremo un componente che si occupa solo di questo

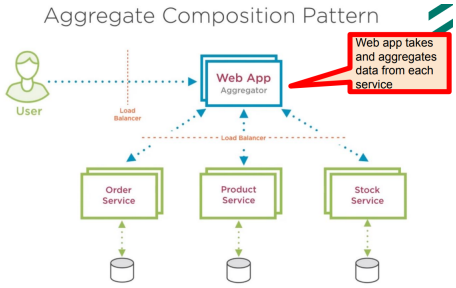

Figure 4: Aggregate Composition pattern

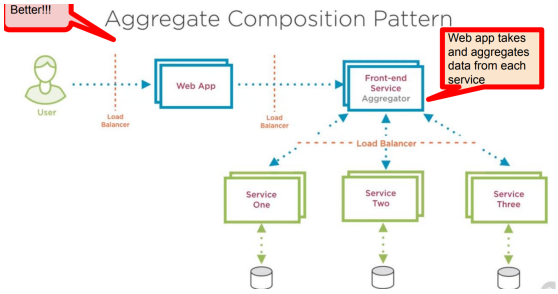

Figure 5: Aggregate Composition pattern improved

4.1.3 Chained Composition pattern

Come suggerisce il nome, questo tipo di composition pattern seguirà la struttura a catena. Il client comunicherà direttamente con i servizi e tutti i servizi saranno

concatenati in modo tale che l'output di un servizio sarà l'input del servizio successivo. L'immagine seguente mostra un tipico microservizio a catena.

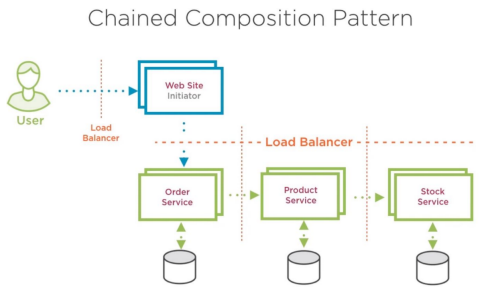

Figure 6: Chained Composition pattern

4.1.4 Proxy Composition pattern

Invece di esporre diversi servizi utilizziamo un api gateway, questo ci permette di effettuare tutte le chiamate al gateway che le smisterà al microservizio opportuno.

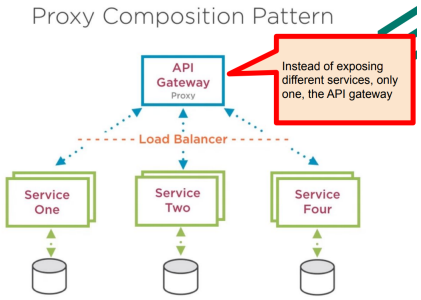

Figure 7: Proxy Composition pattern

4.1.5 Branch Composition pattern

Il microservizio Branch è la versione estesa del pattern aggregatore e del pattern catena. In questo pattern di progettazione, il cliente può comunicare direttamente con il servizio. Inoltre, un servizio può comunicare con più di un servizio alla volta.

4.2 Pattern saga

Si tratta di un pattern il cui scopo è garantire l'ordine di esecuzione delle transazioni che fanno parte della stessa saga, ovvero che devono essere eseguite in un ordine prestabilito e il cui effetto non è compromesso da eventuali transazioni

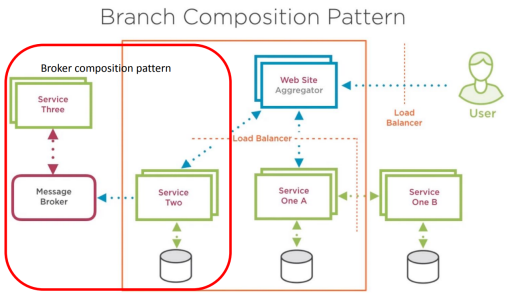

Figure 8: Branch Composition pattern

intermedie appartenenti a saghe diverse. Questo pattern aiuta quindi a gestire la consistenza dei dati nell'esecuzione di transazioni distribuite tra microservizi diversi; coinvolge diversi attori (servizi) che vanno ad agire su una stessa entità tramite singole transazioni atte all'aggiornamento di un dato comune. Ne esistono principalmente due tipi:

- events/choreography → nessun coordinatore, i servizi portano avanti la saga senza avere qualcuno che li controlli, ma lavorando insieme
- commands/orchestration → controllo centralizzato, gestito da un orchestratore

Vediamo un esempio di saga

1. L'utente clicca su "Ordina";
2. L'ordine viene creato;
3. Il servizio di pagamento fa in modo che l'utente possa pagare l'ordine
4. L'utente paga l'ordine;
5. Il servizio di pagamento aggiorna l'ordine;
6. Il servizio degli ordini notifica il ristorante;
7. Il ristorante conferma e immette un orario per il ritiro;
8. Il servizio di delivery notifica il rider;
9. Il rider prende l'ordine e lo consegna;
10. L'ordine viene chiuso.

Ognuna di queste operazioni possono essere eseguite da diversi microservizi che devono quindi collaborare tra loro. Ricordiamo che **Saga è quindi un pattern composite broker** perché i suoi componenti usano un message broker.

4.2.1 Come ci assicuriamo la persistenza dei dati ?

Un esempio di consistenza dei dati proviene dai contesti e-commerce: dopo il pagamento di un ordine, sarà necessario aggiornare sia la tabella relativa ai pagamenti sia quella relativa agli ordini, per fare in modo che l'ordine risulti pagato e possa essere presa in carico per la spedizione; tali aggiornamenti, potenzialmente provenienti da fonti diverse, devono comunque preservare la coerenza dei dati. Preservare la consistenza del dato è una problematica delicata da affrontare. In particolare, ci sono due punti di attenzione che devono essere presi in considerazione in situazioni di questo tipo, ovvero: assicurare la consistenza dei dati in un'operazione distribuita tra sistemi diversi e garantire error safety. Per garantire la consistenza dei dati è necessario assicurarsi che il flusso di esecuzione delle varie operazioni su diversi microservizi rispetti un certo ordine, ciò implica quindi garantire error safety, ovvero la presenza di un meccanismo di remediation che ripristini la situazione corretta in caso di errore. Ed è per questo motivo che il pattern saga usa dei log. Questi log memorizzano tutti i dati degli eventi e degli scambi di messaggi tra servizi; Per farlo dividiamo la transazione in tante piccole richieste che traccio. In caso di errori il saga execution cordinator (SEC)andrà a leggere i log e invierà delle richieste di compensazione (UNDO- anche queste richieste sono servizi rest).

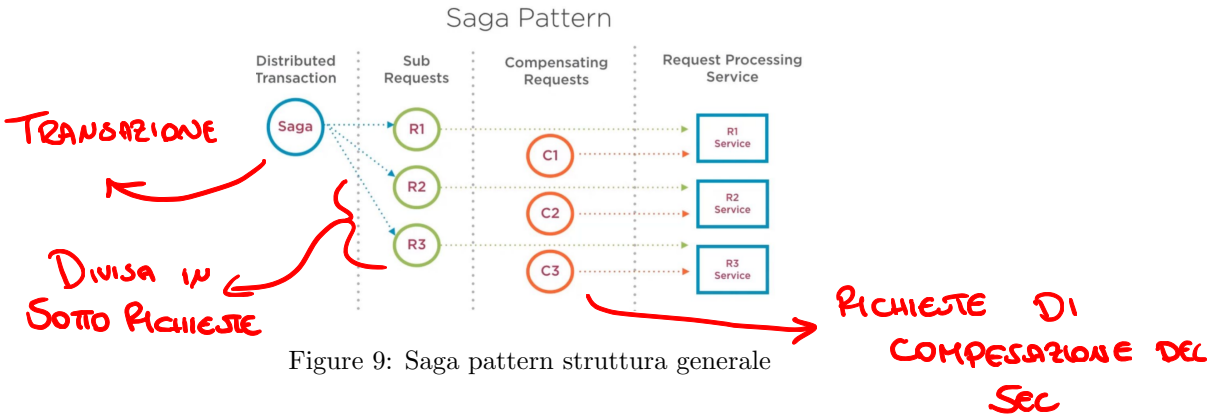

Figure 9: Saga pattern struttura generale

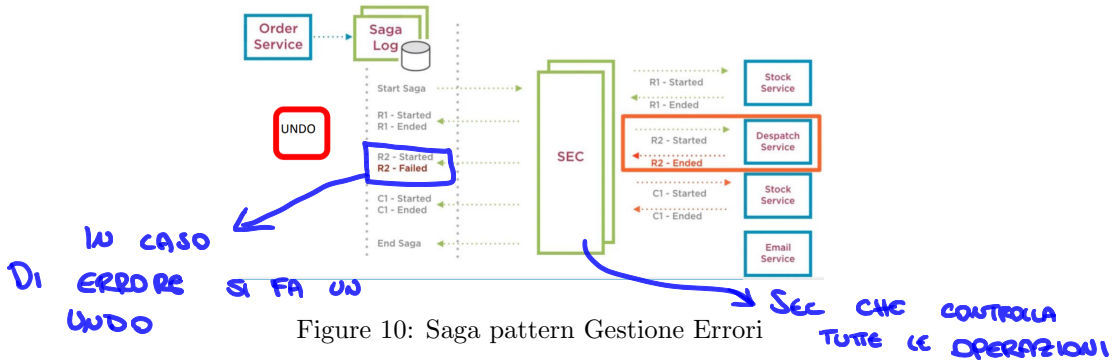

Figure 10: Saga pattern Gestione Errori