Valenziano Andrea - Architettura a Microservizi

Indice

* Introduzione: l’idea dei microservizi
  + Verrà illustrato l'intento originario del concetto di architettura a microservizi: sostituire complesse applicazioni monolitiche con sistemi software costituiti da componenti più semplici e riutilizzabili. L'equilibrio tra velocità e sicurezza è la chiave per comprendere l'essenza dei microservizi.
* Perché un’architettura a microservizi? Vantaggi e principi fondamentali
  + Verranno analizzate le ragioni per cui i primi utilizzatori dell'architettura a microservizi hanno scelto questo approccio. Successivamente, verranno esaminati gli obiettivi e i vantaggi comuni dei microservizi, come si relazionano tra loro e quali obiettivi aziendali possono aiutare a raggiungere.
  + Paragrafi:
    - Principi fondamentali
    - principali vantaggi
    - Le sfide dei microservizi
    - ciclo di vita dei microservizi (Design, develop-deploy, observe)
* Sviluppo di un servizio
  + Si cercherà di illustrare un metodo di progettazione efficace per la stesura di singoli microservizi identificandone dimensioni e funzionalità.
  + Si mostreranno inoltre anche tecniche di comunicazione tra microservizi poiché un servizio aumenta il proprio valore se è in grado di condividere le proprie funzionalità con il sistema.
  + Paragrafi:
    - Confini di un microservizio
    - Definizione di API: Message-Oriented e Hypermedia Driven
    - Il ruolo dei dati nei microservizi
    - Transazioni distribuite e Saga Pattern
    - Microservizi e messaggistica asincrona
    - Rapporti con le dipendenze
* Definizione e sviluppo dell’architettura
  + Si mostrerà come inserire un servizio all’interno dell'architettura, quindi sfruttando container (come docker) e service discovery per identificare ogni singolo servizio all’interno del sistema.
  + Inoltre si spiegherà l’importanza dell’API Gateway per la sicurezza, il routing e l’orchestrazione.
  + Verrà dato spazio anche al funzionamento e all’utilità di alcuni strumenti di monitoraggio.
* Un’applicazione reale: Voucher Management

**Introduzione: L’idea dei microservizi**

Per anni con il termine “rilascio” di un'applicazione software si intendeva quasi esclusivamente la creazione e l’installazione su un server di un singolo pacchetto con all’interno tutto il codice e tutte le configurazioni necessarie per poter utilizzare correttamente l’applicativo. Ancora oggi questo tipo di architettura, detta “monolitica”, è molto utilizzata perché conferisce un senso di stabilità e sicurezza all’intero progetto. Ma tutto ciò comporta anche numerose problematiche, sia per il team di sviluppo e di conseguenza anche per il committente del progetto. Si immagini che la data di scadenza per il rilascio sia imminente e ci si rende della presenza di un bug, la risoluzione di quest’ultimo comporta anche la necessità di doverlo testare con tutto il resto del progetto, allungando notevolmente i tempi. Questo è solo uno dei molti esempi, forse anche uno dei più banali, che hanno portato a cambiare completamente il paradigma delle architetture software. Si è cercato di portare all’estremo il concetto di “divide et impera” cercando di suddividere in blocchi indipendenti tutte le funzionalità dell’applicazione in modo tale da poterle gestire separatamente.

**Perché un’architettura a microservizi? Vantaggi e principi fondamentali**

**Principi fondamentali**

*We can scale our operation independently, maintain unparalleled system availability,*

*and introduce new services quickly without the need for massive reconfiguration[[1]](#footnote-0).*

Questa frase di Werner Vogels riesce ad esprimere in maniera forte e precisa quelli che sono i principi fondamentali su cui si basa lo sviluppo di architetture a microservizi, cioè la possibilità di operare e mantenere singole funzionalità in modo indipendente, riuscendo a scalare e aumentare le risorse rapidamente non impattando sul resto del sistema.

Da questa affermazione si evincono quali sono i punti di forza di questo tipo di architettura, ovvero: autonomia, resilienza, trasparenza, automazione, allineamento.

Questi principi presentano le linee guida chiave per poter portare a termine il processo di design e sviluppo di un’applicazione a microservizi.

**Autonomia**

Per autonomia si intende la capacità che ogni servizio ha di operare e di cambiare in modo del tutto indipendente dagli altri servizi. Questa caratteristica è dovuto dal fatto che ogni servizio è *poco accoppiato* con gli altri, cioè ognuno di essi risulta indipendente dalle implementazioni interne dei servizi con cui comunica. Questo avviene attraverso la definizione chiara di interfacce oppure attraverso la comunicazione ad eventi, utilizzando un modo di comunicare condivisibile tra i vari servizi anche se implementati con tecnologie e linguaggi differenti.

Un altro aspetto che rafforza il concetto di autonomia è quello del rilascio indipendente, che permette ai possibili vari team di sviluppare in parallelo, in modo autonomo appunto, per poi poter rilasciare senza troppe pressioni dovute a tempistiche condivise: ogni microservizio può essere aggiornato rapidamente e in modo frequente anche con con piccoli rilasci che non coinvolgono il resto dell’architettura.

Inoltre, si cerca di rendere autonomo non solo lo sviluppo del singolo servizio, ma anche tutte le altre fasi dalla definizione fino al rilascio, così da permettere ai vari team la possibilità di prendere le decisioni in relazione al proprio contesto e ai loro obiettivi. È proprio questo tipo di approccio che necessaria la formazione di figure competenti sia per lo sviluppo che per la produzione, i cosiddetti *DevOps*.

**Resilienza**

La resilienza è la capacità di rendere il sistema e le funzionalità disponibili all’utilizzo nel caso si verifichino dei guasti o dei rallentamenti. Infatti ogni microservizio permette di isolare queste problematiche circoscrivendole al servizio stesso, in modo tale che gli altri servizi possano continuare a mantenere stabili le loro funzionalità.

L’interazione tra microservizi avviene in modo asincrono e questo permette di evitare che l’assenza o il malfunzionamento di un servizio comprometti a cascata tutte le funzionalità del sistema, ovviamente sarà necessario implementare opportune tecniche di gestione degli errori o dei timeout ( dette *circuit breaker*).

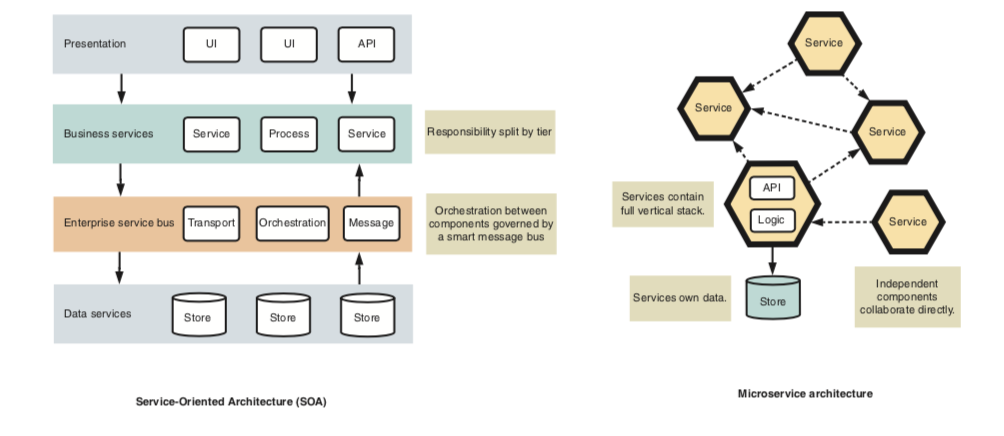
**Trasparenza**

All’interno di un’architettura a microservizi è fondamentale che tutte le sue componenti siano completamente trasparenti, ossia esse devono rendere i loro comportamenti osservabili in modo rapido e semplice. Questo è fondamentale quando ci sono molte interazioni tra microservizi e quindi diventa importante che ognuno di essi esponga le proprie metriche, i log e tenga traccia di tutte le richieste e le chiamate così da poter semplificare la ricerca delle cause di eventuali anomalie e malfunzionamenti.

**Automazione**

Può sembrare controproducente l’idea di riuscire a semplificare lo sviluppo di applicazioni complesse e in continua crescita attraverso la suddivisione in più servizi, infatti l’architettura a microservizi risulta essere più complessa di quella di un’applicazione monolitica. Il trucco sta nella ricerca costante dell'*automazione* e della consistenza nella comunicazione tra servizi. L’automazione permette di assicurare la validità di operazioni sui sistemi e la correttezza dei vari *deploy*, per questo motivo si è visto un incremento nell’utilizzo di tecniche e strumenti di *DevOps* che rendono fattibile lo sviluppo di microservizi a team sempre più ridotti.

**Allineamento**

Un corretto sviluppo di un’architettura a microservizi necessita di un’adeguata distribuzione della struttura dei singoli servizi. Questo allineamento permette ai team di concentrarsi su specifiche aree di business semplificandone così il lavoro. Quindi viene abbandonata l’architettura verticale dell’applicazione Service Oriented (SOA) la quale in realtà viene replicata per ogni singolo servizio eliminando così l’accoppiamento che si creava tra servizi dello stesso livello di astrazione

**Principali vantaggi**

Nel corso degli ultimi anni sono numerosi gli esempi di casi di successo di un’architettura a microservizi (Netflix, SoundCloud, Twitter, ecc…) e tutti continuano a riscontrare molti vantaggi legati sia allo sviluppo che alla progettazione migliorando i rapporti con il cliente e anche la produttività dei vari team.

Tra questi vantaggi troviamo la possibilità di utilizzare tecnologie diverse per esigenze diverse. Infatti aziende che hanno bisogno di inserire un modulo con l’analisi di big data ma possono ricorrere alle tecniche più adeguate indipendentemente dallo stack di tecnologie utilizzate precedentemente.

La complessità di un progetto, purtroppo, è un problema che non può essere eliminato perché risiede nella natura del progetto stesso, tuttavia è possibile evitare che la complessità risulti un grosso ostacolo per lo sviluppo sfruttando altri vantaggi delle architetture a microservizi. La chiave sta nell’indipendeza dei singoli moduli, in questo modo ogni team non ha bisogno di concentrarsi sull’intero sistema complesso e può interagire al meglio e risolvere i problemi legati al più ristretto ambito di competenza.

Questo si traduce anche a livello pratico, infatti ogni modulo deve contenere meno dipendenze possibili verso gli altri moduli durante il processo di compilazione, permettendo così il rilascio rapido di modifiche e miglioramenti. Risulta essere quest’ultimo un grosso beneficio per la produttività dato che non è più necessario dover aggiungere complessità a progetti già enormi per poter rendere effettive le modifiche ed inoltre in questo modo si creano i presupposti per una crescita progettuale senza grosse limitazioni, potendo così sfruttare sempre tecnologie emergenti riuscendo comunque a mantenere la compatibilità con il resto dell’intero sistema.

Ovviamente analizzando queste vantaggiose opportunità sembrerebbe che sviluppare microservizi sia sempre la scelta migliore, in realtà bisogna sempre chiedersi: Perché scegliere un’architettura a microservizi? C’è un bisogno effettivo?

A volte, soprattutto quando si parla di migrare vecchi progetti monolitici verso un’architettura a microservizi, si riscontrano numerose problematiche che, anche dal punto di vista dei costi, ne rendono sconveniente l’utilizzo. Perciò non esiste una risposta oggettiva se utilizzare o meno i microservizi. La scelta deve essere fatta in relazione a fattori come il team, l’azienda e la natura del progetto ed è solo dopo essersi assicurati della possibilità di trarre il massimo profitto che si può procedere nello sviluppo mettendo in pratica quelle che sono le idee chiave di un’architettura a microservizi.

**Le sfide dei microservizi**

Quindi come prendere questa decisione? Quali sono gli effettivi costi progettuali? Quali difficoltà si rischiano di incontrare durante le varie fasi di realizzazione del progetto?

Queste domande sono il preludio ad un’analisi più approfondita sugli effettivi benefici dei microservizi e su quelle che possono essere le sfide che bisogna affrontare per realizzare un’architettura completa e funzionante.

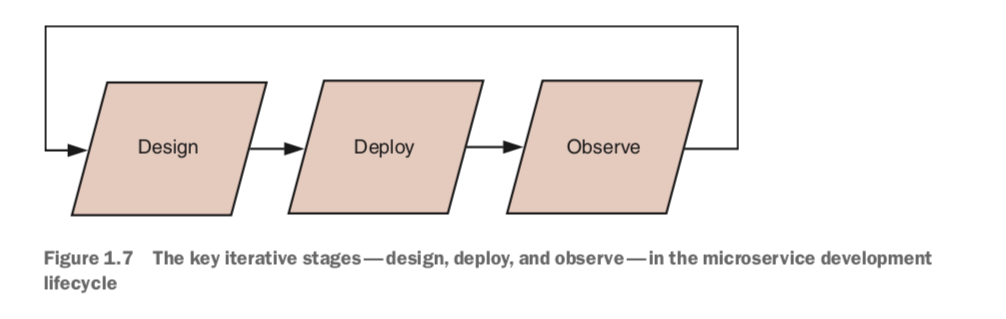
Sicuramente la prima prova da superare riguarda l’identificazione dei microservizi all’interno del contesto progettuale. È necessaria una forte conoscenza di tutto il dominio del progetto per poter individuare i vari moduli di competenza, relative funzioni e i su quali dati dovranno operare. Svolgere bene questa fase permette di far rispettare uno dei principi citati precedentemente, ovvero l’autonomia. Individuando opportunamente i vari microservizi si evita che essi siano in qualche modo dipendenti tra loro. Un’errata analisi porterebbe alla realizzazione di moduli altamente accoppiati che renderebbero l’intero sistema poco propenso alla possibilità di effettuare aggiornamenti in modo rapido, quindi poco flessibile e malleabile.

Una caratteristica fondamentale che ogni microservizio dovrebbe avere è il cosiddetto “contratto”, quello che nella paradigma ad oggetti veniva chiamato “interfaccia”. Il contratto rappresenta una specie di biglietto da visita per un servizio e ha lo scopo di definire in modo chiaro qual è il suo scopo e come interagisce con il resto del sistema. In particolare deve specificare i limiti di informazioni che può ricevere e accettare oltre a riflettere in modo nel modo più fedele possibile quello che potrà essere il reale comportamento di tutti i tipi di implementazione.

Nei paragrafi precedenti si sono illustrati molteplici aspetti positivi delle architetture a microservizi e del loro sviluppo su sistemi distribuiti ad esempio larghezza di banda pressoché infinita, latenza zero, infrastrutture di rete affidabili, ecc.. Tutti questi vantaggi in realtà nascondono diverse problematiche legate alla consistenza dello stato dei vari servizi. Quest’ultimo può essere alterato dal fatto che i dati non risiedono nello stesso luogo fisico e che le operazioni non sempre vengono svolte nell’ordine desiderato. Questo accade perché le interazioni tra servizi avvengono a livello applicativo, in modo indipendente dalle infrastrutture, per questo è necessario porre particolare attenzione alla consistenza di tutto il sistema.

Infine un ulteriore ostacolo allo sviluppo di microservizi risiede nella divisione del lavoro in team. Spesso la divisione dei compiti tra i vari team è davvero molto marcata, tanto che ogni team è completamente autonomo per tecnologie, gestione dei carichi di lavoro e consegne. Per questo può risultare complicato mettere insieme tutti i risultati raggiunti se prima non viene fatta una coordinazione accurata che permetta di gestire e regolare le priorità dettando tempi e responsabilità per i vari team.

**Ciclo di vita dei microservizi**

Le fasi di progettazione e sviluppo di un’architettura a microservizi non si discostano molto da quelle di un'architettura monolitica, almeno a grandi linee. Infatti questi passaggi possono essere riassunti e racchiusi in grandi blocchi di operazioni, ovvero progettazione, sviluppo e rilascio, monitoraggio. 

Ovviamente applicando queste operazioni a servizi molto piccoli esse risultano essere molto rapide e permettono di correggere eventuali difetti molto velocemente.

La fase di progettazione è fondamentale perché è quella in cui vengono definiti gli scopi e le funzionalità di ogni singolo servizio, eventuali errori in questa fase rischierebbero di compromettere tutte le altre fasi. In questa fase vengono fatte anche delle scelte tecniche come in che modo rendere i servizi resilienti, quindi come gestire eventuali errori provenienti dalla rete, dalle infrastrutture e dalla comunicazione con altri servizi. Proprio sulla comunicazione è opportuno scegliere se avvalersi di tecniche di messaggistica sincrona o asincrona oppure di una forma ibrida valutando pro e contro.

La seconda fase racchiude le operazioni di sviluppo e rilascio, che nelle architetture a microservizi sono strettamente collegate e spesso vengono eseguite dalle stesse persone.

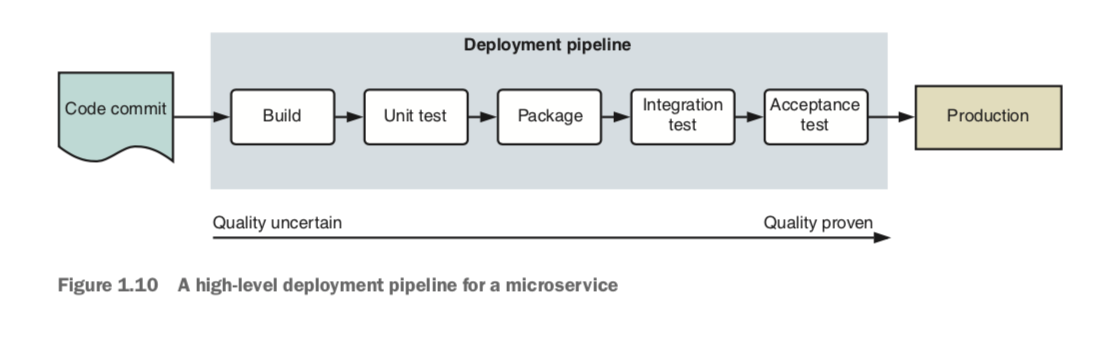
Rilasciare un servizio può risultare complicato soprattutto se la si utilizzano varie tecnologie: è impensabile che si predisponi un server sempre diverso e configurato in modo tale che si adatti ai vari servizi. Occorre standardizzare il processo di rilascio cercando di tenere fisse quali sono le possibili criticità, come ad esempio su quale server sarà rilasciato, con quali altri servizi ha una dipendenza, come parte l’applicazione, ecc…

La soluzione migliore consiste nell’utilizzo dei “container” che permettono di isolare tutti gli ambienti in cui i vari servizi vengono eseguiti in modo tale da poterli adattare ad ogni necessità legate alle diverse tecnologie. Quindi l’applicazione non viene più installata direttamente sul server, ma viene prima “impacchettata” in un container con tutte le dipendenze necessarie a far partire l’applicazione. Il servizio ora, all’interno del suo ambiente è pronto a partire, basta solo installare il container su un server funzionante che prescinde dal tipo di applicazione e dal tipo di container.

È chiaro che il rilascio di un servizio risulta molto più rapido, e questa velocità è amplificata dall’utilizzo delle continuous delivery pipelines, ovvero un set di istruzione da compiere per poter rilasciare l’applicazione. Il continuous delivery è una tecnica che ha lo scopo di permettere un rilascio rapido e completo al termine delle operazioni di sviluppo.

Quindi in una pipeline si predispongono tutte le fasi con tutte le operazioni da effettuare in modo tale da standardizzare ulteriormente i rilasci e da permettere di riconoscere facilmente la posizione nella pipeline di eventuali anomalie.

Le istruzioni partono appena viene effettuato un commit, vengono eseguiti i vari test e ogni volta che viene superato con successo un blocco della pipeline si passa al successivo fino ad installare il servizio nel container e a riavviare la macchina su cui sta girando.



Queste tecniche sicuramente semplificano lo sviluppo, ma inoltre permettono di ridurre notevolmente i costi: infatti in questo modo il rilascio di una nuova funzionalità viene effettuato in tempi ridotti.

Una volta che l’applicazione viene rilasciata è necessario verificarne l’effettivo funzionamento attraverso opportune tecniche di osservazione e analisi. Possono verificarsi problemi dovuti a cause differenti, infatti potrebbe esserci stato un errore nello sviluppo e che quindi può essere risolto con una modifica e un successivo rilascio, ma i problemi possono essere dovuti anche all’infrastruttura, alle macchine o alla rete, in questi casi è opportuno prevederle e gestirle magari con messaggi o schermate di timeout. Il problema reale è che risulta molto più difficile rintracciare questi errori in un’architettura a microservizi rispetto ad una monolitica poiché le transazioni, i dati e le operazioni possono essere condivise su più istanze e su più servizi e quindi la mole informazioni prodotta risulta essere maggiore. Per questo bisogna dotare ogni servizio di opportuni strumenti di monitoraggio che devono funzionare come uno schermo per il servizio in modo tale che tutte le operazioni vengano tracciate e risultino facilmente consultabili. Questi strumenti devono registrare sia l’effettivo funzionamento quindi mostrando e raccogliendo i var LOG a livello applicativo, sia tutte le metriche legate all’infrastruttura e alla rete: come ad esempio il livello di memoria e di CPU impiegato oppure la quantità di pacchetti scambiati all’interno della rete.

In questo modo si conclude quello che a grandi linee è il processo di analisi, sviluppo, rilascio e monitoraggio di un microservizio che con le sue peculiarità ha lo scopo di velocizzare tutto il processo e mantenendo stabilità e consistenza e fornendo le opportune informazioni per poter verificare che tutto funzioni correttamente.

**Principi di progettazione dei microservizi**

Il processo di progettazione di un microservizio passa attraverso delle scelte fondamentali utili a semplificare lo sviluppo e alla riuscita dell’intero processo.

Una di queste scelte consiste nel saper definire quali sono gli effettivi confini del servizio, ovvero, quali funzionalità gli competono e quali lasciare ad altri servizi. Un’ analisi corretta permette di evitare che il servizio abbia un forte accoppiamento con altri servizi aumentandone così l’indipendenza all’interno dell’architettura. Definire bene i confini permette in modo particolare di identificare tutto quello che deve essere salvato nella rispettiva base dati così da poter avere un’idea precisa di quelli che saranno i modelli legati a quel microservizio evitando di dover interferire con i dati strettamente legati ad altri servizi.

Ovviamente i servizi hanno bisogno di comunicare tra loro e per questo occorre occupparsi delle scelte legate alla messaggistica, alla modellazione delle transazioni e alle dipendenze all’interno di un’architettura a microservizi.

**Confini di un microservizio**

Non è possibile quantificare esattamente la “grandezza” di un microservizio perché essa dipende da numerosi fattori legati al contesto in cui il servizio dovrà agire.

Per questo è meglio concentrarsi sulla qualità che sulla quantità, ovvero bisogna tenere in considerazione i casi d’uso in cui il servizio verrà utilizzato. Da questo concetto nasce l’idea di *domain-driven design* (DDD) di Eric Evans:

*“The objects had behavior and enforced rules. The model wasn’t just a data schema; it was integral solving a complex problem.”[[2]](#footnote-1)* Bisogna quindi permettere ai vari modelli di rappresentare non solo la realtà ma che di fare da guida per la definizione dei vari microservizi.

Sicuramente un modo efficace per progettare un microservizio è quello di basarsi su alcune caratteristiche tecniche che esso deve avere: per esempio per in base alle funzionalità e alle prestazioni desiderate si può decidere di utilizzare diversi linguaggi di programmazione per in base alle esigenze perciò ad ogni linguaggio potrebbe corrispondere un servizio. Inoltre ci si può basare anche sulle conoscenze dei vari team o sull’effettiva area geografica su cui opera il servizio, per esempio si può circoscrivere un microservizio quando bisogna sviluppare un particolare processo di marketing all’interno del mercato di una specifica nazione e quindi in quel caso il team esperto di marketing di quella nazione prenderà in carico lo sviluppo definendo così i confini di quel servizio. Queste scelte, benché comode, risultano essere troppo approssimative , bisogna scendere più nel dettaglio e per questo che si sfrutta il concetto di *modello* che sì, rappresenta la realtà, ma che permette anche di costruirci intorno tutta l’architettura. Grazie al modello si riesce a identificare i dati che dovrà rappresentare eliminando quelli che invece non lo riguardano in modo diretto, nel caso invece che si trovino due modelli che agiscono sugli stessi dati si interviene riformulando i confini e riducendo l’accoppiamento tra i due modelli in modo tale che ognuno di essi sia rilasciabile in modo completamente indipendente. Con il DDD si riesce a creare un ecosistema di modelli ben definiti e autonomi, ma questi modelli sono ancora qualcosa di astratto e troppo esteso, è necessario ridurre le dimensioni dei modelli per identificare quelli che sono i reali servizi.

La chiave di tutto sta nel cercare di ridurre al minimo ogni componente e ogni operazione, quindi servizi con minime funzionalità sono rilasciabili in minor tempo e con minor dispendio di risorse. Questa affermazione è alla base della formazione di quelli che sono i cosiddetti DevOps, ovvero coloro che lavorano all’intero processo di realizzazione di un servizio, dallo sviluppo al rilascio, che hanno lo scopo di velocizzare tutte queste operazioni per poter rilasciare sempre più spesso basandosi sui concetti del manifesto Agile:

*Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.[[3]](#footnote-2)*

Tutti questi concetti riassumono perfettamente la tecnica del Continuous Delivery che risulta essere la scelta perfetta per poter sviluppare e rilasciare i singoli moduli una volta progettati cercando di ridurre al minimo le loro funzionalità.

In antitesi a tutto questo si pone questa affermazione di Vaughn Vernon, autore di Implementing Domain-Driven Design:

*Bounded context should be as big as it needs to be in order to fully express its complete ubiquitous language.[[4]](#footnote-3)*

Quindi non ridurre tutto al minimo risulta essere sufficiente, bisogna tenere in considerazione numerose variabili che vanno la semplice definizione del modello, soprattutto per quanto riguarda la gestione dei dati e delle transazioni. È necessario tenere sempre presente quello che è il contesto per il quale si sta sviluppando, esso detta quelle che sono le restrizioni da applicare ai servizi, e che quindi non dipendendono esclusivamente dalla forma maggiormente ridotta del modello definito in seguito alle tecniche di DDD. Sicuramente queste sono un ottimo inizio per la definizione di un microservizio, ma è opportuno proseguire con l’analisi di tutte le altre variabili in gioco.

**Definizione di API: Message-Oriented e Hypermedia Driven**

Si è parlato dell’importanza di definire il contesto dei microservizi per poterli rendere completamente indipendenti, però all’interno di un’architettura la comunicazione tra le varie applicazioni è fondamentale, quindi bisogna utilizzare delle tecniche che permettano di interagire con gli altri componenti del progetto senza rimuovere quella importante caratteristica che è quella del rilascio indipendente.

La soluzione sta nello sfruttare il paradigma Message-Oriented, quindi, come avviene anche nella realtà, i microservizi comunicano tra di loro attraverso “messaggi” che contengono tutte le informazioni necessarie per eseguire determinate operazioni. Queste informazioni sono appunto quei dati sui quali solo un determinato servizio ha accesso diretto, tutti gli altri dovranno inviare un messaggio per poterli richiedere. Durante la progettazione dell’intero sistema è opportuno sia decidere uno standard su cui basare la comunicazione (HTTP, JSON, code asincrone, … ) sia definire il modo in cui raggiungere un determinato servizio, a prescindere dal metodo utilizzato per ogni servizio si espone un “indirizzo” per poterlo raggiungere e si specificano diversi punti di ingresso (endpoint) in base a quali dati richiedere. Il corpo dei vari messaggi consiste quindi di veri e propri oggetti del modello però serializzati in modo tale da poter essere ricevuti da qualsiasi tipo di applicazione mantenendo così la completa indipendenza dal resto dell’architettura.

Sostanziali miglioramenti al paradigma message-oriented vengono apportati grazie all’impiego di implementazioni definite Hypermedia-driven nelle quali i messaggi non sono esclusivamente testuali, ma possono contenere anche diversi tipo di dati (hypermedia appunto) che non si limitano a fornire l’informazione richiesta ma incorporano anche possibili azioni da poter effettuare dopo aver ricevuto il dato richiesto. Questo concetto si basa sull’idea con cui un utente fisico si approccia al web: non è un approccio statico, l’utente non si limita a visionare le informazioni ma riesce ad interagire con esse per esempio attraverso attraverso dei link, e sono proprio i link la vera differenza con la normale comunicazione a messaggi. Infatti in un messaggio non sono presenti solo i dati ma incorporano anche gli indirizzi per poter richiedere altri dati e informazioni collegate a quelle del corpo del messaggio. Questa tecnica è chiamata: Hypermedia As The Engine Of Application State (HATEOAS) nella quale appunto il cuore di tutto l’applicazione sono le API (i messaggi) che permettono di navigare e richiamare tutte le funzionalità dell’applicazione.

Queste caratteristiche ricordano il paradigma ad oggetti dove in un *object* ci sono sia i dati che i comportamenti dell’oggetto stesso, ed è quello che avviene con HATEOAS, sostituendo gli oggetti con le API che non forniscono più dati in modo statico ma permettono un’interazione più completa con tutto il sistema. Inoltre è anche possibile tenere traccia della versione di un’ API in modo tale da rendere l’applicazione retrocompatibile dato che ogni variazione non andrà ad impattare sui sistemi che utilizzano la versione precedente.

Le Hypermedia API sono più simili al modo in cui una persona fisica utilizza il web: evolutive, adattabili, indipendenti dalle versioni, infatti un utente non si preoccupa mai della versione del sito web che sta osservando. Pertanto, lesse sono meno fragili, più facilmente individuabili e si adattano perfettamente per i sistemi distribuiti come appunto i microservizi.

**Il ruolo dei dati nei microservizi**

Spesso quando si vuole progettare un’applicazione la prima cosa su cui si pone maggiore attenzione sono i dati, infatti si cerca sin da subito di individuare tutti i modelli che compongono il sistemi e li si converte nelle varie tabelle che faranno parte del database. Questo approccio centralizzato sui dati non è sempre una scelta positiva se si vuole sviluppare un’architettura a microservizi. Infatti in questo modo si perdono di vista quelle che sono le funzionalità che poi andranno a formare i vari servizi e si rischia così di creare accoppiamento tra essi poiché più funzionalità potrebbero aver bisogno di accedere alla stessa tabella compromettendo così l’indipendenza dei vari servizi.

Una tecnica per ovviare a questo problema può essere quella di creare un livello applicativo sulla base dati che più servizi hanno bisogno di interrogare, quindi viene sviluppato un nuovo servizio come unico punto di accesso al database “condiviso”. Ovviamente questa soluzione può essere applicata solo ai casi più semplici, per situazioni più complesse occorre effettuare uno studio più accurato sui flussi applicativi e soprattutto occorre focalizzarsi non più solo sui dati ma anche sugli *eventi*.

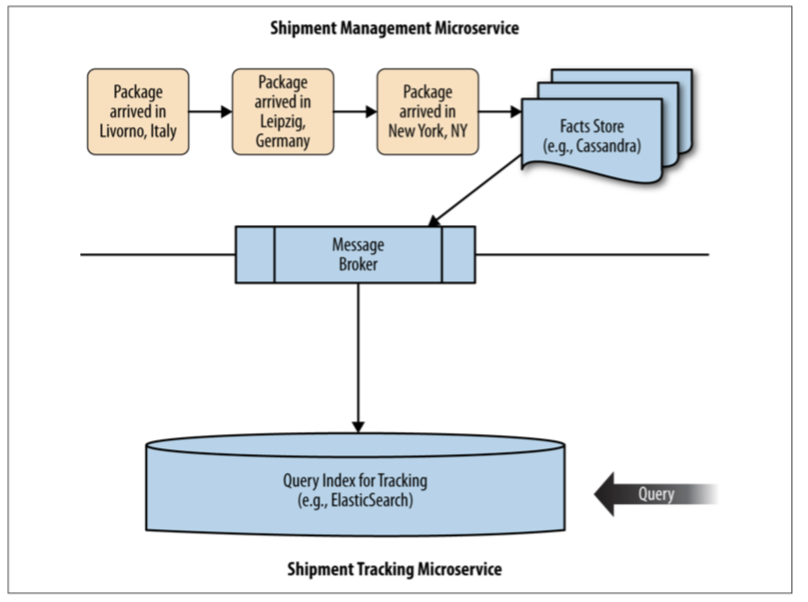
Event sourcing e CQRS

La tecnica dell’ event sourcing consiste proprio nel salvare non più i dati ma i singoli eventi che avvengono per ogni microservizio. Questo è molto simile a quello che fanno le banche che non salvano il valore del saldo sul conto, ma lo calcolano mettendo insieme tutte le transazioni che sono state effettuate precedentemente, quindi sui database non sono presenti i valori statici, ma vengono salvate le transazioni (quindi gli eventi) legati a quel conto. Ovviamente è possibile creare delle istantanee dello stato del sistema per evitare di dover tener conto di tutti gli eventi, sfruttando soltanto quelli successivi all’istantanea.

Questa tecnica porta a notevoli vantaggi, uno di questi riguardo lo sviluppo, infatti in questo modo è molto più semplice individuare lo stato del sistema dopo un determinato evento o in un determinato momento, però questo implica l’immutabilità dei dati: non è possibile modificare il “passato” per effettuare delle correzioni, ma è possibile aggiungere nuovi eventi che rimuovano eventuali errori. Nel caso della banca, a fronte di un saldo errato, vengono emesse delle transazioni “correttive” per riportare la situazione alla normalità.

L’event sourcing è anche utile alla progettazione dell’intera architettura, infatti è grazie a questa tecnica che è possibile disaccoppiare tutti i microservizi tra loro facendo in modo che la gestione dei dati sia mascherata all’esterno di ogni servizio. Questo concetto è alla base non solo della gestione degli eventi ma anche di tutta la gestione dei sistemi: l’utilizzo dei container infatti permette, per ogni container, di poter gestire a pieno l’intero ciclo di vita senza dover conoscere effettivamente come ogni fase sia composta, di quali dipendenze ha bisogno o di quali altri servizi deve richiamare poiché è tutto specificato all’interno del singolo evento.

A completamento della tecnica dell’event sourcing si sfrutta un pattern legato all’interrogazione della base dati: Command Query Responsibility Segregation (CQRS) che permette di gestire dati condivisi su più microservizi. L’idea di base è quella di suddividere le responsabilità, ovvero chi può effettuare le interrogazioni (query) non è detto che possa effettuare modifiche ai dati stessi. Quindi il servizio che gestisce i dati, perché all’interno del proprio contesto, si occupa di effettuare gli inserimenti e le modifiche e notifica ogni evento a tutti gli servizi collegati in modo tale che essi possano crearsi un indice dei dati sempre aggiornato. Per le notifiche occorre utilizzare un Message Broker al quale dovranno sottoscriversi tutti i servizi interessati a quel particolare evento.



L’utilizzo di questa tecnica comunque rischia di complicare l’architettura e quindi conviene utilizzarla solo per ovviare al problema del *data sharing*. In tutti gli altri casi l’approccio CRUD tradizionale permette di delineare in modo semplice il contesto dei vari servizi definendo così l’accesso ai dati senza dover ricorrere a strumenti esterni.

**Transazioni distribuite e Saga Pattern**

Il problema dei dati condivisi non è l’unico che può presentarsi quando si lavora con i microservizi, infatti all’interno di un sistema con numerosi database sono fondamentali le cosiddette *transazioni*. Per transazione si intende un flusso di modifiche ai dati presenti sul database che permette, in caso di errori, di poter tornare allo stato iniziale.

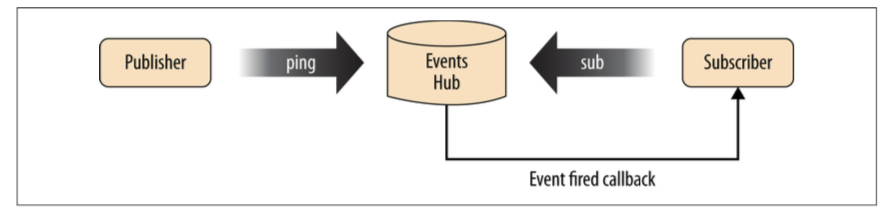
Ovviamente questo non può accadere in modo naturale in un’architettura a microservizi dato che ogni modulo non conosce la struttura del modulo precedente e in caso di errore non può risalire allo stato precedente dei dati. Per questo motivo spesso si ricorre all’utilizzo del Saga Pattern ideato da Hector Garcia-Molina e Kenneth Salem nel 1987 durante il periodo in cui hanno lavorato all’università di Princeton. Questo pattern è molto utile poiché permette un certo grado di “reversibilità” delle transazione anche se esse avvengono in più fasi e su database distribuiti. Il compromesso consiste nel non poter riportare il sistema esattamente allo stato iniziale da cui è partito prima dell’inizio delle transazioni, ma nel creare una *transazione compensativa* che si occupa di modificare il sistema in modo tale da raggiungere nuovamente uno stato consistente.

Il funzionamento si basa sul concetto di azione-reazione: per ogni transazione si salva una chiamata ad una transazione contraria o compensativa in un messaggio chiamato *routing slip* (scappatoia). Appena uno degli elementi del flusso di transazioni fallisce richiama il messaggio più recente nel routing slip ed effettua la transazione compensativa riportando così il sistema ad uno stato equivalente a quello iniziale anche se attraverso un altro percorso.

Grazie alla sua natura altamente distribuita e alla sua alta *fault tolerance* il Saga pattern è la soluzione ideale per sostituire le transazioni tradizionali quando queste coinvolgono più microservizi superandone i confini.

**Microservizi e messaggistica asincrona**

Oltre che per l’accesso ai dati di altri dati database, i microservizi hanno spesso bisogno di comunicare tra loro soprattutto per notificare determinati eventi che interessano più aree del sistema. Uno di questi metodi di comunicazione è la messaggistica asincrona che avviene attraverso l’applicazione del pattern Publish-Subscriber attraverso un broker capace di gestire code di messaggi. In realtà la soluzione ottimale sarebbe quella di passare dal PubSub classico al protocollo WebSub (il vecchio PubSubHubbub ) nel quale si inserisce un microservizio che è direttamente collegato al broker in modo tale che gli altri servizi non debbano legarsi ad una particolare implementazione del Broker (Kafka, RabbitMQ, ecc) generalizzando così le modalità di ricezione e invio di messaggi.



La comunicazione tra microservizi può avvenire anche in modo sincrono, per esempio tramite chiamate HTTP, ed è una soluzione abbastanza valida ma che comunque comporta un alto accoppiamento tra i servizi e, avendo una struttura rigida, non permette di poter sviluppare nuove funzionalità in modo rapido.

La comunicazione asincrona invece è più flessibile e permette di ampliare molto velocemente i tipi di eventi che possono essere inviati ed intercettati dato che chi invia un evento non ha bisogno di sapere anche il modo in cui i vari *consumer* cercheranno di riceverlo.

Questo stile quindi consente un'evoluzione più fluida e riduce l’accoppiamento tra i servizi , però tutto questo ha un costo: Dato che la comunicazione asincrona non ha un comportamento lineare e questo rende le interazioni tra servizi poco prevedibili si richiedono grossi investimenti nel monitoraggio per tracciare in modo adeguato tutto ciò che accade durante lo scambio di informazioni tra servizi attraverso la coda di messaggi.

**Il rapporto con le dipendenze**

Un altro importante argomento legato all’indipendenza dei microservizi è la gestione delle dipendenze che risulta essere un fattore determinante per valutare l’effettiva mobilità di un microservizio.

Una delle dipendenze cruciali è quella legata al database dato che è fondamentale evitare problemi legati alla condivisione di dati. Decidere di centralizzare tutti i dati sarebbe una scelta comoda dal punto di vista della gestione poiché in quel caso sarebbero dei team specializzati ad occuparsi della complicata manutenzione dei database, però questo va contro molti principi dei microservizi. L’ideale sarebbe quello di inglobare all’interno di ogni servizio tutte le dipendenze di cui ha bisogno in modo tale che esso possa essere spostato e installato in qualsiasi ambiente. Ma questa è una tecnica molto onerosa perché se per esempio ci sono 100 servizi che hanno bisogno di un database allora bisogna configurare e installare lo stesso db per 100 volte. Ma tutto questo non è necessario, è sufficiente trovare un compromesso che porti ad un risultato migliore: un servizio non ha bisogno di avere per forza tutte le dipendenza utili al suo funzionamento, ma deve cercare di sfruttare quello che già è presente sul sistema su cui andrà installato. Le funzionalità essenziali possono essere tranquillamente installate una volta per tutti i servizi che ne abbiamo bisogno (ovviamente per i database non si parla di condividere i dati ma solo le loro strutture e le configurazioni).

In questo modo il processo di migrazione di uno o più servizi resta abbastanza rapido ed efficiente e continua a rispettare quelli che sono i paletti principali di un’architettura a microservizi.

**Definizione e sviluppo dell’architettura**

**Rilascio indipendente**

La possibilità di poter rilasciare ogni microservizio in modo completamente indipendente dagli altri presenti nel sistema è uno dei principi fondamentali che porta più benefici all’architettura.

Questa caratteristica permette di poter scalare un servizio in modo diretto ogni qualvolta si ha bisogno di aumentarne o ridurne le prestazioni. Nelle architetture monolitiche, invece, la scalabilità non è garantita in modo così immediato: è necessario saper prevedere tutti i possibili scenari legati all’evoluzione dell’applicativo e acquistare costose macchine sovradimensionate per poter sopperire ad eventuali picchi di utilizzo del sistema stesso.

Questa problematica non è presente in un ambiente cloud dove è molto semplice migliore la capacità o le prestazioni di una macchina in modo completamente trasparente.

1. Werner Vogels, Chief Technology Officer, Amazon Web Services [↑](#footnote-ref-0)
2. Eric Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software [↑](#footnote-ref-1)
3. The Agile Manifesto, Kent Beck [↑](#footnote-ref-2)
4. Implementing Domain-Driven Design, Vaughn Vernon [↑](#footnote-ref-3)