ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA
CAMPUS DI CESENA
Corso di Laurea in Ingegneria Informatica

MICROSERVIZI PER IL PROGETTO DI SISTEMI DISTRIBUITI

Elaborata nel corso di: Sistemi Distribuiti LA

Tesi di Laurea di:
SIMONE LAI

Relatore:
Prof. ANDREA OMICINI

ANNO ACCADEMICO 2015–2016
SESSIONE III

PAROLE CHIAVE

Microservizi
Sistemi distribuiti
Applicazione a microservizi
Tecnologie Web
Servizi Web

Alla mia famiglia e alle persone che mi hanno sempre voluto bene

Indice

Introduzione

Obiettivi della tesi

La tesi ha l'obiettivo di esplorare stato e prospettive dei microservizi come tecnica emergente per il progetto e implementazione dei sistemi distribuiti; allo scopo, prima di affrontare direttamente il tema dei microservizi, verrà prima esaminata l'architettura monolitica e poi le architetture SOA, per arrivare ad avere utili considerazioni e confronti con l'architettura a microservizi, introdotta prima cercando di avere una definizione da utilizzare per la trattazione e poi negli aspetti e nelle tecnologie che li hanno portati allo stato attuale. Poi verrà dedicato un capitolo ad un possibile esempio implementativo di applicazione a microservizi, per concludere con possibili prospettive e sviluppi futuri dei microservizi.

Struttura della tesi

La tesi sarà quindi così strutturata :

- Nel capitolo 1 si parlerà di architettura monolitica, partendo dalla sua definizione e definendone vantaggi e svantaggi, sia da una prospettiva

"storica", sia per considerazioni nei capitoli successivi.

- Nel capitolo 2 verranno affrontate le architetture SOA, cercando dapprima le definizioni ed approfondendo il concetto di servizio, poi analizzandone vantaggi e svantaggi.

- Nel capitolo 3 si parlerà dell'argomento principale della tesi, i microservizi, cominciando da un'introduzione che arrivi ad una definizione di sintesi partendo dal fatto che non esiste una definizione univoca di microservizio. Fatto questo, verranno esaminati concetti teorici e pratici importanti per avere una buona architettura per un'applicazione a microservizi, perchè come verrà spiegato, è fondamentale un'ottima base di progettazione dell'applicazione per avere un'efficace architettura a microservizi. Poi la seconda parte del capitolo si concentrerà sugli aspetti tecnologici hardware e software che hanno permesso l'utilizzo pratico dei microservizi. Questo capitolo si concluderà con una breve panoramica sul linguaggio Jolie e sui pattern dei microservizi e su una possibile scelta per la struttura di un'applicazione a microservizi.

- Nel capitolo 4 si presenterà un esempio di applicazione a microservizi distribuita. Essa mostrerà solamente una minima parte di tutti gli aspetti dei microservizi, ma offrirà, insieme ai capitoli precedenti, gli spunti di approfondimento ed espansione dell'applicazione.

- Nel capitolo 5, infine, le conclusioni con gli obiettivi raggiunti ed i possibili sviluppi futuri dei microservizi, dal punto di vista tecnico e produttivo.

Capitolo 1

Architettura monolitica

1.1 Descrizione

Nello sviluppo di un'applicazione, se questa è composta da funzionalità contenute in un unico blocco(detto anche "monolite"), si parla di architettura monolitica.

L'architettura monolitica è nell'informatica quella storicamente tradizionale, essendo presente sin dai primi programmi su mainframe e per alcuni decenni successivi anche l'unica architettura software ed in ogni caso la più semplice.

1.2 Vantaggi

Le applicazioni sviluppate in questo modo, se di dimensioni relativamente ridotte, risultano semplici da sviluppare, potendo pensare solamente ad un'unica unità e tecnologia [1] per tutte le funzionalità richieste dai requisiti

dell'applicazione stessa; in questo modo, di conseguenza, gli sviluppatori si concentreranno sull'utilizzo della singola tecnologia.

Inoltre, risulta immediato eseguire test di funzionamento dell'intera applicazione in un singolo passaggio, proprio per il fatto che questa è monolitica. L'installazione e la distribuzione vengono effettuate semplicemente con operazioni di copia/trasferimento ed esecuzione dei file (talvolta anche dell'unico file da cui è costituita l'applicazione in produzione)

Infine, in caso occorra scalare l'applicazione, basta effettuarne più copie da eseguire simultaneamente.

1.3 Svantaggi

Tuttavia, nel momento in cui l'applicazione nel tempo aumenta di dimensioni, per esempio migliorando ed estendendo le varie funzionalità, aggiungendo nuovi moduli e rispettando ulteriori e nuovi requisiti richiesti da chi la utilizza, l'architettura monolitica porta a svariati svantaggi.

- L'applicazione è implementata utilizzando una singola tecnologia: nel tempo essa diventa obsoleta, meno efficiente, e nel caso in cui passino molti anni dalla realizzazione iniziale e cambi sensibilmente l'hardware su cui viene eseguita, persino completamente inutilizzabile!

- Conseguenza diretta del punto precedente: negli anni nuovi sviluppatori potrebbero conoscere poco o non conoscere affatto la tecnologia e/o il linguaggio di programmazione utilizzati nella prima versione, ciò comporta maggior tempo di aggiornamento della stessa e tempo dedi-

cato ad apprendere le nozioni necessarie per poter iniziare le modifiche. Sono quindi fortemente vincolati alle scelte iniziali di progettazione, architetturali, alle tecnologie utilizzate, alle librerie già presenti e a dover progettare le nuove parti dell'applicazione in modo da non compromettere il funzionamento delle altre parti già presenti. Ogni revisione dell'applicazione, perciò, è fortemente condizionata dalle scelte compiute inizialmente [2] .

- Ulteriore conseguenza tecnologica: per risolvere lo "stallo" dovuto ai punti descritti precedentemente, l'unica soluzione appare una nuova implementazione dell'intera applicazione con una nuova tecnologia, aumentando esponenzialmente tempi e costi di sviluppo;non solo, nel momento in cui venisse sviluppata una terza tecnologia, bisognerebbe nuovamente realizzare l'intera applicazione.

- La modifica di una singola parte o modulo comporta la ridistribuzione dell'intera applicazione, aumentando i tempi in cui non è disponibile in produzione.

- Nel momento in cui l'applicazione è veramente complessa, anche solo capire il codice nel suo insieme diventa difficile e qualsiasi modifica per ampliarla o per correggerne un bug può portare potenzialmente ad azioni sbagliate, complicando ulteriormente e progressivamente la situazione.

- Per quanto riguarda la scalabilità, diversi moduli potrebbero avere requisiti hardware diversi e a volte in contrasto tra loro, portando allo

svantaggio di dover limitare le risorse come compromesso per poter distribuire l'applicazione

- Nel caso in cui venga realizzata una nuova versione, per la ridistribuzione occorre fermare le istanze dell'applicazione in esecuzione, ricopiare più volte la nuova applicazione e riavviarla tante volte quante sono le repliche stabilite per la scalabilità, aumentando i tempi in cui non è disponibile in produzione. Si osserva anche che questo è l'unico modo per scalare l'applicazione in un'architettura monolitica.

Nel corso degli anni, si sono aggiunti nuovi requisiti e possibilità, l'evoluzione della Rete dal www ad IoT (Internet Of Things, "Internet delle cose") dove potenzialmente ogni oggetto può accedere alla Rete e scambiare dati ed informazioni, quindi nuovi dispositivi ed utilizzatori delle applicazioni distribuite: per questi motivi, grandi organizzazioni come Amazon[3], Netflix[4], eBay, Uber, Riot Games e molte altre, negli anni sono progressivamente passate dall'organizzazione interna tramite architettura monolitica ai microservizi.

Per esempio, nella figura seguente viene mostrato l'esempio di tale cambio organizzativo da parte di Amazon

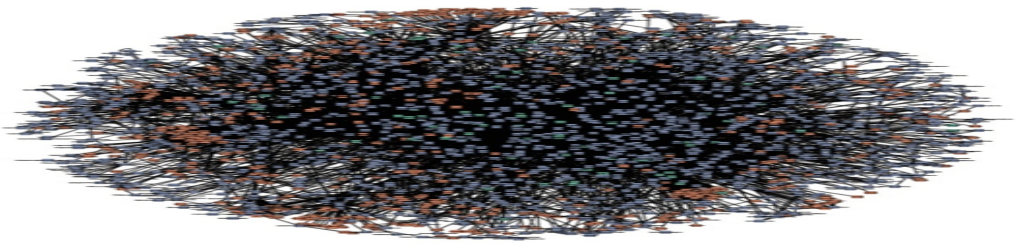

Figura 1.1: Mappa dei numerosi microservizi di Amazon (2009)

Questo passaggio implementativo è stato progressivamente possibile grazie a svariate innovazioni tecnologiche avvenute nel tempo quali SOA, cloud, container e miglioramento dei protocolli di comunicazione. Questi aspetti saranno descritti nel dettaglio nei successivi capitoli, così come la loro utilità nella realizzazione dei microservizi.

Capitolo 2

SOA

2.1 Descrizione e definizioni

Con l'evoluzione dei sistemi e delle applicazioni distribuite, si è sviluppato il concetto di SOA, acronimo di Service Oriented Architecture, cioè architettura orientata ai servizi. Nonostante non esista una sola ed univoca definizione del concetto, è importante notare che la parola chiave di questa sigla è servizi. Un servizio può essere definito come un meccanismo per accedere ad una funzionalità o ad un insieme di funzionalità software attraverso un'interfaccia che ne stabilisce vincoli e politiche di utilizzo. [5]

Per ampliare e specificare meglio i concetti appena descritti, un servizio solitamente dovrebbe [6]:

- Implementare una singola parte del dominio architetturale
- Avere la possibilità di essere usato (ed anche riusato) indipendentemente dagli altri servizi

- Garantire di poter essere utilizzabile ed utilizzato dall'esterno conoscendo solamente la sua interfaccia

- Poter essere usato da differenti tecnologie e linguaggi di programmazione

- Essere registrato in una directory dei servizi. In questo modo, i client che lo utilizzano, riescono a localizzarlo e quindi ad usarlo durante l'esecuzione dell'applicazione cercandolo in tale directory.

- Essere disponibile in Rete, in quanto parte di un sistema distribuito

Perciò, se ad esempio dobbiamo implementare un'applicazione e-commerce, le prime considerazioni SOA riguardano la suddivisione del dominio delle entità in singole parti: quindi si può pensare a dividere il tutto in : servizio clienti, contenente le informazioni sui clienti, la possibilità di aggiungere un nuovo cliente, di leggere la lista delle operazioni, ordini, mail effettuate dallo stesso e così via; servizio ordini, contenente le operazioni di aggiunta di un nuovo ordine, rimozione e gestione degli ordini; servizio catalogo prodotti, contenente tutto ciò che riguarda i prodotti e le loro caratteristiche, le operazioni effettuabili come aggiunta di un nuovo prodotto, modifica e cancellazione ed eventuali altri aspetti richiesti dai requisiti dell'applicazione.

Inoltre, fatto questo passo, occorre notare che per far correttamente funzionare l'applicazione in produzione, vanno aggiunti i servizi che permettano l'integrazione tra le varie entità, cioè predisporre una piattaforma che ne permetta la comunicazione reciproca, attraverso le interfacce già previste ed implementate e la loro composizione attraverso meccanismi di orchestra-

zione, in pratica si tratta di un'infrastuttura in grado di coordinare i vari servizi e di rispondere adeguatamente ai vari tipi di messaggi che riceve, richiamando ogni volta i servizi adeguati. Perciò contiene il modello dei processi applicativi e costituisce una parte importante della logica dell'intera applicazione.

Infine, tale sistema SOA può essere utilizzato attraverso interfacce utente (UI e/o GUI) interne alla produzione (esempio: servizio di inserimento nuovi prodotti da parte dei gestori dell'e-commerce) od esterne e rivolte per esempio ai potenziali clienti dell'applicazione, potendo prevedere anche diverse GUI, differenziate per dispositivo di accesso al sistema (esempi: PC, Smartphone, IoT). In un contesto SOA, ciascun elemento distinto esaminato qui sopra, è destinato ad essere progettato, implementato e messo in produzione da un diverso gruppo di lavoro, solitamente specializzato nel contesto produttivo di cui va ad occuparsi nell'applicazione.

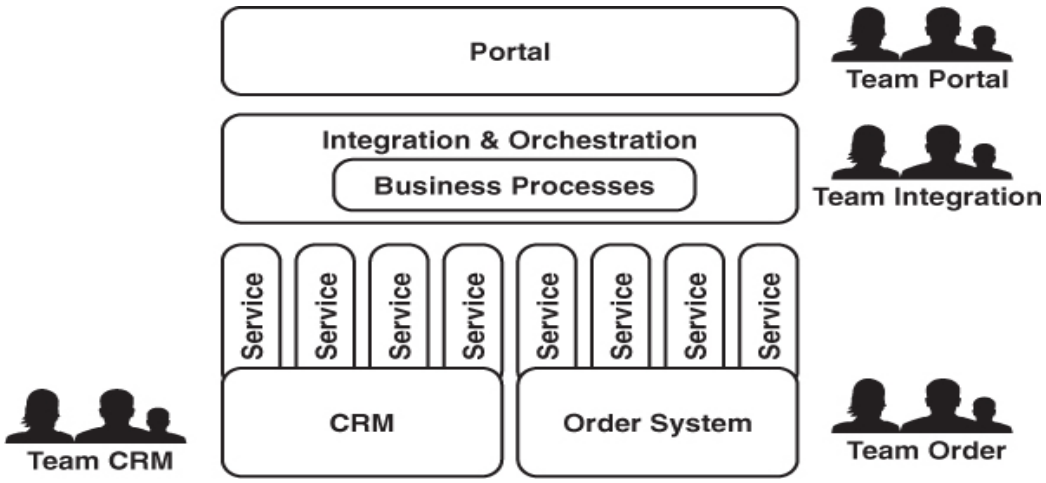

Figura 2.1: Esempio di applicazione SOA

La figura illustra un possibile schema di quanto appena descritto, considerando che nella parte bassa della stessa si ripete il singolo blocco (con il proprio gruppo di lavoro dedicato) per ogni entità del dominio (quindi servizio clienti indicato con CRM in figura, ordini, catalogo, eccetera).

2.2 Vantaggi

Nel passaggio da architettura monolitica a SOA si hanno svariati vantaggi, che si cercherà di sfruttare nei microservizi. Nelle architetture SOA la comunicazione tra i servizi avviene tramite messaggi e/o eventi; questo fornisce come vantaggio rispetto all'applicazione precedente il fatto di permettere un basso accoppiamento, cioè una minima dipendenza e legame tra le varie parti del sistema, mentre nell'applicazione monolitica tale legame è, per costruzione, molto forte. La comunicazione [6] può inoltre essere sia sincrona sia asincrona, ogni servizio perciò può rispondere autonomamente e dinamicamente ai vari tipi di messaggi scambiati nel sistema, e questo permette, di conseguenza, una maggiore flessibilità tramite orchestrazione e , nel momento in cui si debbano aggiungere nuove parti e servizi all'applicazione, una migliore integrazione.

Si possono notare miglioramenti anche a livello organizzativo, dal momento che non si ha più l'obbligo da parte di tutti di guardare a tutto il sistema nel dettaglio per non causare danni ad un'altra parte dell'applicazione facendo modifiche come nel caso monolotico, ma ogni gruppo di lavoro può teoricamente guardare solamente al funzionamento della propria area produttiva e relativo servizio ed alle interfacce con cui vengono definiti i servizi

usati ”esternamente”. Per esempio, se si stesse implementando, in un generico e-commerce, il servizio clienti, e si avesse bisogno di inviare la fattura via e-mail all'utente che ha appena effettuato un acquisto, si sfrutterebbero le interfacce DettagliProdotto dal servizio prodotti e DettagliOrdine dal servizio ordini, oltre ovviamente ai dati ”interni” del cliente.

Infine, è utile sottolineare come SOA abbia una visione architetturale complessiva dell'intera attività produttiva.

2.3 Svantaggi

Nonostante tutto, l'architettura SOA porta con sè anche alcuni svantaggi. In questa sezione si cercherà di esaminarli anche da una prospettiva di confronto con i microservizi; infatti,nei prossimi capitoli, verranno cercate soluzioni che vadano ad eliminare o comunque a limitare il più possibile l'impatto di tali svantaggi nell'implementazione dell'applicazione e nella sua efficienza in produzione.

I primi svantaggi sono una maggiore complessità del sistema ed un maggior costo iniziale in fase di progettazione, questi effetti possono comunque essere ridotti nel medio e lungo termine, grazie ad una maggiore produttività e sviluppo rispetto al caso monolitico.

La logica di un'entità dell'applicazione, in un sistema SOA, è distribuita, quindi cambiamenti ed aggiornamenti anche piccoli portano a dover modificare e coordinare più parti dell'applicazione così come più gruppi di lavoro contemporaneamente; ciò potrebbe significare che gli addetti al settore in cui avviene la modifica siano costretti ad attendere che altri team abbiano cam-

biato di conseguenza la loro parte di servizio e questo si traduce facilmente in tempi più lunghi di sviluppo e messa in produzione.

L'orchestrazione e l'integrazione dei servizi, come descritto nelle sezioni precedenti, è contenuta nello stesso "blocco" e questo può andare bene solo nel momento in cui non si prevedano molte modifiche nè all'interfaccia utente nè ai vari servizi, che sono le parti che comunicano con questo blocco. Viceversa, le modifiche coinvolgono tutte le parti in comunicazione tra loro, con le stesse considerazioni sulla maggiore coordinazione tra gruppi appena descritta. Nei microservizi invece, ogni microservizio è distinto dall'altro , quindi non c'è un microservizio che si occupi di più parti, così come teoricamente è in grado di elaborare la richiesta e fornire una risposta internamente senza dover mai chiamare un altro microservizio. Quest'ultimo aspetto verrà approfondito nel prossimo capitolo.

L'applicazione SOA è sì divisa in servizi, ma è ancora messa in produzione in modo monolitico [6], mentre i microservizi possono essere gestiti individualmente.

Per poter fornire all'utente un'interfaccia UI/GUI, è previsto il servizio apposito,ma questo porta ad uno svantaggio: qualsiasi modifica ad un servizio può portare ad una modifica dell'interfaccia grafica e/o viceversa. Inoltre, l'aggiunta di un nuovo servizio, può potenzialmente portare a dover modificare anche le interazioni tra servizi, sicuramente il blocco di integrazione/orchestrazione e l'interfaccia utente. L'uso dei microservizi limita invece l'impatto di decisioni progettuali come queste, limitandosi alla modifica/sostituzione di poche se non di un'unica piccola parte di applicazione coinvolta.

Infine, una considerazione che se non come uno svantaggio può essere vi-

sta come una differenza notevole tra SOA e microservizi: nelle architetture SOA si progetta il servizio guardando totalmente la visione d'insieme dell'applicazione, mentre nei microservizi si progetta l'architettura per singolo progetto/sezione, dividendolo ulteriormente in microservizi, e questo sarà solamente uno dei concetti esaminati per i microservizi;del resto, a parte alcune analogie con le architetture SOA descritte nel corso di questo capitolo, le idee che portano alla progettazione, alle considerazioni da adottare a seconda dei requisiti, le tecnologie e l'implementazione dell'applicazione saranno profondamente diverse da quelle utilizzate sin qui per le architetture orientate ai servizi.

Ed è esattamente di questo che si occuperanno i prossimi capitoli!

Capitolo 3

Microservizi

3.1 Introduzione ai microservizi

L'architettura a microservizi può essere definita [7] come l'approccio allo sviluppo di un'applicazione composta da tanti piccoli servizi, ciascuno dei quali eseguito da un proprio processo, con un meccanismo di scambio di messaggi leggero. La gestione centralizzata dell'applicazione è ridotta al minimo indispensabile e perciò i suoi microservizi possono essere implementati usando diversi linguaggi di programmazione e diverse tecnologie di gestione dati, oltre ad essere messi in produzione in maniera indipendente dagli altri microservizi.

Tuttavia, non esiste un solo modo per definire i microservizi [6], ma varie definizioni del concetto al variare della prospettiva da cui si analizzano.

In particolare, se si guarda ai microservizi dal punto di vista organizzativo, essi possono essere visti come gestiti da gruppi di lavoro eterogenei e composti da un numero molto ridotto di elementi.

Se invece si esamina l'infrastruttura interna alla produzione, il microservizio dovrebbe essere quell'unità applicativa tale appunto da poter essere sviluppata ed essere distribuita e messa in produzione indipendentemente dagli altri microservizi. Per esempio l'infrastruttura per il microservizio può comprendere la possibilità di test, controllo di versione, aggiornamento e distribuzione automatica, così come utilizzare un database e quindi dovrebbe avere una dimensione tale da poter sfruttare tutti questi elementi. Solitamente, il fatto che si utilizzino varie fasi in maniera automatica, porta a microservizi di dimensioni più ridotta ed un aumento del loro numero.

In realtà, anche se le applicazioni progettate ed implementate a microservizi sono un aspetto recente, l'idea di avere funzionalità distribuite e di dimensioni ridotte può essere ad esempio riportata alla filosofia UNIX, che pone l'accento su tre aspetti, utilissimi per ragionare a microservizi.

Tali aspetti si possono sintetizzare così :

- Un programma dovrebbe svolgere uno ed un solo compito, ma svolgerlo estremamente bene
- I programmi dovrebbero essere in grado di lavorare ed interagire insieme
- Dovrebbe essere utilizzata un'interfaccia universale

Da questi aspetti si ricavano alcuni di quelli che verranno descritti come i "punti fermi" e come considerazioni per cercare di ottenere una buona applicazione a microservizi, che ne sfrutti i vantaggi e limiti (se non elimini) gli svantaggi. Per poter far questo, occorre che l'architettura a microservizi sia pensata e progettata con molta cura.

Importante premessa per le sezioni ed i capitoli seguenti: è corretto sottolineare che non esiste l'architettura perfetta a microservizi, ma che è più corretto parlare di architettura ottimale a microservizi considerando ogni volta la particolare situazione produttiva nel suo complesso in quel momento, cioè per esempio esaminare l'organizzazione, esaminare le persone a disposizione, il loro numero e gli ambiti e le tecnologie da loro meglio usate e conosciute, ed esaminare le tecnologie ed i protocolli di comunicazione ottimali per ogni microservizio, ed altri concetti spiegati nella prossima sezione.

3.2 Concetti da esaminare per una buona architettura a microservizi

L'adozione di un'architettura basata sui microservizi, porta potenzialmente una libertà molto maggiore agli sviluppatori, non più legati a scelte precedenti e liberi di progettare il proprio microservizio senza vincoli [2] per esempio sul linguaggio scelto, sul database utilizzato, sulla piattaforma di sviluppo e potendo concentrarsi solamente ad implementare le poche e specifiche funzionalità richieste al microservizio. Tuttavia, l'intera applicazione in produzione sarà costituita da molti microservizi, in alcuni casi anche centinaia o migliaia, perciò nella pratica rimangono valide le considerazioni appena esposte, ma bisogna osservare che ogni microservizio deve essere in grado, anche indirettamente, di coordinarsi con gli altri microservizi e anche se sviluppato indipendentemante dagli altri, non deve ovviamente compromettere il funzionamento di altri microservizi né del sistema nel suo complesso.

Per cercare un'architettura ottimale per i microservizi, si tiene conto del fatto che i requisiti che dovrebbero rispettare devono essere sempre presenti per i microservizi, per l'applicazione da questi composta, ed anche per tutti i sistemi distribuiti.

Ne risultano le seguenti caratteristiche: stabilità, affidabilità, scalabilità, tolleranza ai guasti, performance, monitoraggio e documentazione. Queste caratteristiche possono essere viste concettualmente come i requisiti di un microservizio in produzione, e in dettaglio:

- Stabilità: un microservizio è stabile se il suo sviluppo, distribuzione, messa in produzione, aggiornamento, cambio ed aggiunta di tecnologia, oppure rimozione di tecnologie deprecate, non causano l'instabilità di altri microservizi e/o dell'intera applicazione. Per ottenere questo è quindi opportuno operare cicli e procedure di sviluppo, distribuzione, introduzione ed aggiornamento tecnologico altrettanto stabili e se possibile standard ed automatizzati.

- Affidabilità: caratteristica fortemente legata alla stabilità, poichè un microservizio oltre che stabile dovrebbe anche garantire che nel momento in cui per qualsiasi motivo non fosse disponibile, questo non abbia conseguenze gravi di instabilità degli altri microservizi con cui comunica nè a maggior ragione dell'intero sistema. Per l'affidabilità occorre che la stessa procedura di produzione sia stabile, progettare il microservizio in modo da mitigare gli effetti prevedendo risposte alternative da fornire a chi ne ha richiesto il messaggio ed anche un buon

microservizio (od un insieme) per cercare microservizi alternativi da chiamare.

- Scalabilità: molto raramente il microservizio e/o l'intera applicazione ricevono una quantità costante di traffico dati, così come difficilmente un'applicazione rimane di dimensioni e complessità costanti nel tempo. Perciò occorre progettare pensando anche a lungo termine, con stime accurate quantitative e qualitative di crescita, identificando potenziali parti critiche che potrebbero rallentare o rendere meno efficiente l'applicazione nel tempo, con strategie di gestione del traffico, delle possibili nuove dipendenze tra microservizi che potrebbero sorgere, e database scalabili.

- Tolleranza ai guasti: i microservizi portano complessità nell'architettura dell'applicazione distribuita che, in quanto tale, può essere soggetta a guasti, anche limitati al singolo microservizio, dovuti per esempio a problemi di rete o più semplicemente a bug nel codice del microservizio. Quindi occorre esaminare l'applicazione cercando quali parti e per quali eventi possano portare a conseguenze produttive serie se non disastrose, identificarli e correggerne gli errori, predisporre strategie di identificazione e rimedio dei guasti, così come la gestione del traffico sapendo che possono capitare problemi e piattaforme di test e di simulazione guasti sui microservizi.

- Performance: anche questa legata alla scalabilità, semplicemente aumentando sensibilmente il numero di richieste contemporanee ad un microservizio, non deve cambiarne troppo l'efficienza, per esempio prefe-

rendo dove possibile chiamate asincrone e parallele a chiamate sincrone, e prevedendo un utilizzo ridotto ed efficace delle risorse.

- Monitoraggio: principio legato all'affidabilità, prevede attività e microservizi di log degli eventi e degli errori, di "tracciabilità" per costruire il percorso di una richiesta nello scambio di messaggi tra microservizi e risposte di ritorno e talvolta implementate graficamente con dashboards in modo da poter essere interpretate da ciascun elemento che lavori all'applicazione e metriche per monitorare la quantità di traffico, di CPU utilizzata , informazioni sui database, sugli errori ed altri aspetti attraverso vari livelli di avvisi graduati in base alla gravità.

- Documentazione: già importante in qualsiasi progetto software condiviso, nei microservizi riveste un'importanza fondamentale. Infatti, i possibili frequenti cambi di codice, sviluppatori, tecnologie, requisiti ed applicazione portano a sviluppi molto veloci e numerosi; quindi è opportuno tenere una documentazione aggiornata e centralizzata del microservizio, in modo che le persone e/o i gruppi di lavoro che interagiscono con il microservizio, sappiano come farlo in maniera produttiva, ed in generale centralizzarla offre a tutti la possibilità di avere la visione di insieme dell'applicazione, senza dover conoscerne i dettagli implementativi e tecnologici.

3.3 Caratteristiche dei microservizi

Come conseguenza delle sezioni precedenti di questo capitolo, si possono stabilire le caratteristiche dei singoli microservizi.

Ciascun microservizio implementa la propria specifica funzionalità e l'architettura a microservizi basa la propria comunicazione attraverso lo scambio di messaggi. Perciò sarà prevista un'interfaccia del microservizio con un punto d'ingresso (o più) che sia in grado di ricevere un messaggio, di elaborarlo, anche dinamicamente e se la comunicazione prevista è bidirezionale, anche di fornire un messaggio di risposta al microservizio che lo ha utilizzato. Naturalmente, tale interfaccia definisce anche le operazioni richiamabili dall'esterno. Inoltre, può essere previsto uno (o più) punti di uscita in cui il microservizio chiamato a sua volta può decidere di utilizzare un'operazione di un terzo microservizio e ad esempio a ritroso riceverne la risposta, fare altre elaborazioni e rispondere al microservizio iniziale con il risultato dell'elaborazione composta dei vari passaggi, potendo così applicare il concetto di microservizio composizione di microservizi. Si può in questo caso fare un'analogia con il paradigma ad oggetti: un oggetto può avere analogie con un microservizio ed i suoi metodi con le operazioni dello stesso. Questo è un esempio, poichè non sempre la corrispondenza è esatta, dipendendo dai pattern di progettazione dell'applicazione a microservizi.

Si può notare come l'unico vincolo per lo scambio di messaggi sia quello che le parti che comunicano riescano ad "intendersi", ed anche che ciò porta a poter avere un microservizio "intermedio" che riceva un messaggio in un formato stabilito, faccia alcune elaborazioni e trasmetta un messaggio in un

formato diverso ad un terzo microservizio!

In un sistema composto a microservizi, non c'è centralizzazione dell'architettura e di conseguenza neanche dei dati, aspetti da cui derivano altre possibili caratteristiche di un microservizio: può comunicare con un database (o meglio con un DBMS) e in questo caso i dati presenti non saranno ovviamente quelli dell'intera applicazione ma solo i necessari per il microservizio, può prevedere una cache interna, per poter velocizzare i tempi di risposta per i messaggi che non cambiano o per fornire una risposta come alternativa ad un guasto.

Inoltre tra microservizi c'è solitamente poca dipendenza e basso accoppiamento (loose coupling).

Infine, nelle applicazioni che prevedano UI/GUI destinate ad interagire con l'utente, queste saranno composte dai vari microservizi, che hanno quindi come ulteriore caratteristica quella di avere l'interfaccia grafica parziale data dal microservizio in questione, componibile in questi microservizi.

3.4 Vantaggi e svantaggi

Un'applicazione a microservizi presenta, come avviene per tutti gli stili architetturali [8], vantaggi e svantaggi; il vantaggio più evidente è che, per costruzione, i microservizi rafforzano il concetto di delimitazione dei moduli in cui è divisa l'architettura e di fatto forza lo sviluppatore a rispettare tale divisione con tutti i vantaggi che ne conseguono. Per esempio, se si ha un'applicazione e-commerce questa sarà composta da un certo numero di microservizi e un buon pattern di progettazione può essere quello di parti-

re dai requisiti complessivi e dividerli secondo le varie aree di funzionalità, in altre parole applicando i principi di DDD (Domain Driven Design)[9], scomponendo l'intera applicazione in aree produttive, e ciascuna di queste in microservizi.

Questa considerazione permette di trasformare un apparente svantaggio in un vantaggio per i microservizi, partendo dalla citazione detta "Legge di Conway" [10] che in sintesi afferma "Qualsiasi organizzazione che progetti un sistema finirà inevitabilmente vincolata a produrre un progetto la cui struttura è una copia della struttura comunicativa dell'organizzazione". In altre parole, data una struttura modulare e gruppi di lavoro specializzati in ciascun modulo ed omogenei, verrà prodotta una struttura a livelli, dove ogni livello riflette esattamente un modulo. Usando i microservizi invece, si è praticamente vincolati a ristrutturare i gruppi di lavoro rendendoli "misti" cioè in modo che in ogni team ci siano le varie specializzazioni. Per chiarire meglio il concetto ed il conseguente vantaggio, si può osservare le seguenti figure [7]:

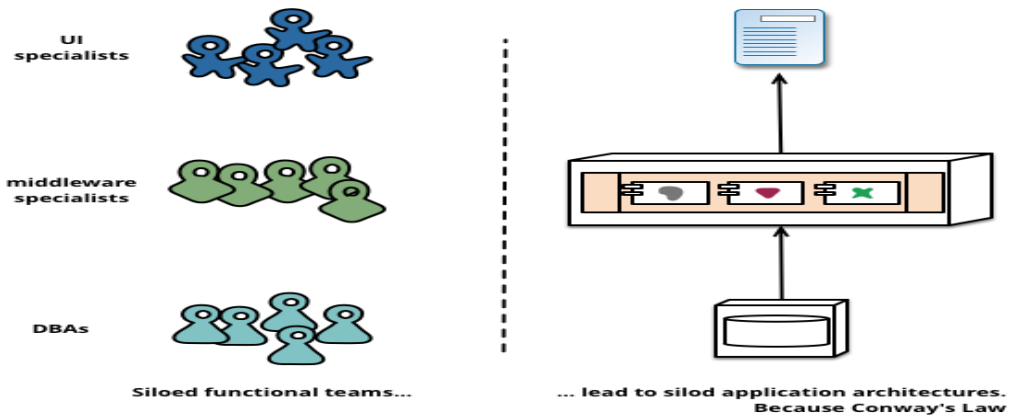

Figura 3.1: Team omogenei, conseguenza: legge di Conway

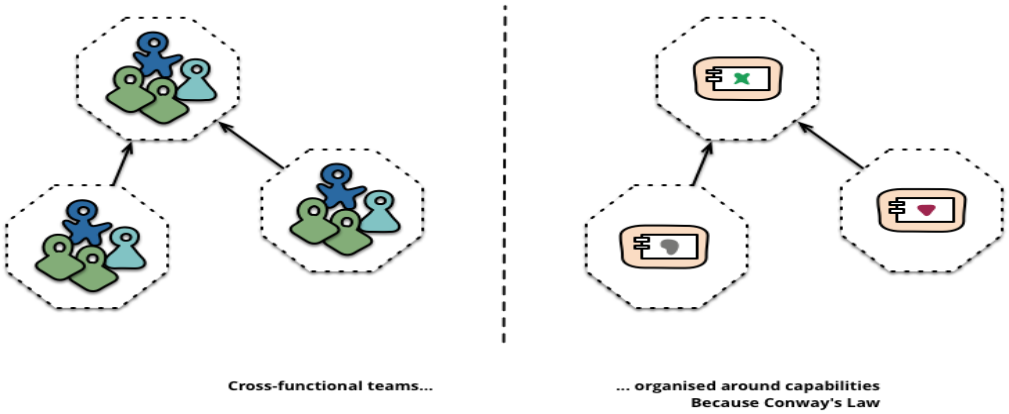

Figura 3.2: Riorganizzazione a microservizi: la legge di Conway ora è vantaggiosa

Lo svantaggio più evidente è invece la maggiore complessità architetturale del sistema a microservizi dovuta anche al fatto che l'applicazione è distribuita, che si traduce di conseguenza in un elevato costo di progettazione e produzione iniziale, cosa a cui si può porre rimedio pensando a lungo termine allo sviluppo dell'applicazione.

Ogni microservizio ha il vantaggio di poter essere sviluppato in maniera indipendente dagli altri così come qualsiasi problema di sviluppo o di distribuzione non si riflette sul resto dell'applicazione; inoltre l'architettura prevede cicli di sviluppo continui e rapidi, perciò in caso di problemi o guasti si può facilmente sostituire od aggiornare il microservizio.

Tuttavia, trattandosi di un sistema distribuito, è soggetto al teorema CAP o teorema di Brewer, il quale afferma che è impossibile avere contemporaneamente le tre caratteristiche della sigla CAP (cioè Consistency, Availability, Partition tolerance) ma occorre sceglierne due su tre. Tali aspetti significano,

nel caso dei microservizi :

- Consistenza: ogni richiesta di lettura, riceve sempre il giusto messaggio attuale oppure un errore
- Disponibilità: ogni richiesta riceve comunque una risposta, senza garanzia che sia quella più recente e quindi correttamente aggiornata
- Tolleranza di partizione: in ogni caso di perdita di messaggi tra microservizi e/o nodi della rete (o loro ritardo superiore ad un timeout stabilito), il sistema continua comunque a funzionare.

Però, tenendo sempre presente che un sistema distribuito può avere problemi di rete, il ragionamento si può ridurre a: dato un problema alla rete, si scelga tra l'avere consistenza oppure avere disponibilità. Quindi se si sceglie la consistenza, si invia un messaggio di risposta corretto se il microservizio ha la sicurezza che sia quello più aggiornato, altrimenti si invia un messaggio contenente un errore e/o un avviso di timeout; se invece si sceglie la disponibilità, un'elaborazione del messaggio viene comunque fatta ed una risposta disponibile viene sempre data, a prescindere dal fatto che sia quella corretta.

Ovviamente, se il sistema distribuito funziona correttamente, tutto lo scambio di messaggi avviene naturalmente e senza errori, ma il teorema è da tener presente e molto utile per la progettazione delle risposte dei microservizi in caso di errore.

Grande vantaggio dei microservizi è la possibilità di avere un'ampia diversità tecnologica nelle varie parti dell'applicazione, che permette varie conseguenze pratiche positive: libertà di scelta delle tecnologie da parte degli

sviluppatori, maggiori possibilità di utilizzare per ogni microservizio la tecnologia ottimale per i requisiti richiesti e/o la tecnologia ottimale per lo sviluppatore.

Infine, insieme ad una maggiore complessità architetturale, i microservizi portano lo svantaggio, da rimediare e cercare di trasformare in vantaggio, di una complessità operazionale molto maggiore rispetto a SOA ed architetture monolitiche e l'introduzione del concetto, introdotto più avanti nel capitolo, di DevOps.

3.5 Introduzione ai programmi/tecnologie utilizzabili per implementare applicazione a microservizi

L'utilizzo dei microservizi in applicazioni produttive concrete è stato reso possibile nel tempo anche dallo sviluppo di alcune tecnologie e programmi, oltre che dei sistemi distribuiti.

L'evoluzione ed il miglioramento dei protocolli di comunicazione per esempio permette nel caso dei microservizi, e non solo, una migliore gestione nello scambio di messaggi e maggiori possibilità implementative, anche per il vantaggio, descritto nella sezione precedente, di potere potenzialmente utilizzare la migliore soluzione possibile nella specifica comunicazione tra microservizi, e quindi più tipi di messaggio, anche in maniera dinamica.

Tra le tecnologie decisive nel passaggio da architettura monolitica a SOA e poi a microservizi, c'è il cloud, utilizzato come concetto anche nei primi

mainframe le cui informazioni potevano essere lette da vari terminali, ma evolutosi negli ultimi decenni, nel momento in cui sono arrivati alcuni fornitori di servizi cloud al pubblico; sfruttare tali servizi comporta una minore complessità "interna" dell'applicazione, così come sono ridotti il lavoro sistemistico, di manutenzione ed i relativi costi. Questi aspetti verranno ampliati nella sezione apposita di questo capitolo.

Infine, per questa introduzione occorre citare anche il concetto di container software, come contenitore di un'entità dell'applicazione virtualizzata, ed evoluzione delle macchine virtuali come descritto successivamente.

3.6 Possibili protocolli di comunicazione tra le varie parti

I microservizi, come già accennato, basano la comunicazione reciproca sullo scambio di messaggi. In questa sezione vengono elencati alcuni casi applicativi concreti e varie possibilità di scelta.

Un caso comune possono essere le applicazioni web, in cui il client è il browser dell'utente e lato server agiscono i vari microservizi che si coordinano tra loro e forniscono una risposta dinamica al variare della richiesta. Per il concetto secondo cui ogni parte che comunica con l'altra deve avere un protocollo in comune, in questo caso specifico la richiesta avrà protocollo HTTP ed approccio REST (Representational State Transfer) ma, da qui in poi, internamente al server e/o al resto dell'applicazione distribuita a microservizi, si sfrutta il vantaggio che non si è più obbligati ad avere l'intera struttura

RESTful e di conseguenza nemmeno l'architettura ROA (resource-oriented architecture , cioè orientata alle risorse), anzi, si avrà massima libertà tecnologica ed architetturale, con l'unico vincolo, nel microservizio "più vicino" al browser, di preparare una risposta che sia una risorsa nel senso inteso dallo stesso, cioè una pagina web, un foglio di stile, uno script ed altro fruibile ed interpretabile in modo da poter essere utilizzabile dall'utente che ha fatto la richiesta iniziale.

Nel caso di un'applicazione desktop ci potrebbero essere ancora meno vincoli, potendo utilizzare anche altri protocolli noti ed anche potendo definire un proprio protocollo applicativo per l'applicazione, così come un proprio formato per i messaggi. In questo caso, inoltre, si potrebbe decidere di avere un client anch'esso a sua volta diviso in microservizi spostando eventualmente in parte (se ci fosse un vantaggio produttivo) carico computazionale dal server al client. Le stesse considerazioni valgono anche per altri dispositivi di connessione.

Un altro aspetto importante di un'applicazione a microservizi può essere l'utilizzo di eventi da registrare per poter fare da tramite e disaccoppiare microservizi che altrimenti sarebbero fortemente vincolati. In concreto, per fare questo si può stabilire un proprio formato e protocollo, oppure utilizzare una delle soluzioni più note in questo senso (perchè utilizzabili con più linguaggi di programmazione e piattaforme) tra, ad esempio:

- RabbitMQ [11], che implementa lo standard AMQP (Advanced Message Queuing Protocol) [12]
- 0MQ/ZeroMQ/ZMQ [13]

- ActiveMQ [14] oppure HornetQ [15], entrambi implementano JMS (Java Messaging Service) [16]

- Feed ATOM [17][18], che però non supportano le transazioni.

3.7 Cenni al cloud

Come avviene per altri concetti descritti in precedenza, non c'è una definizione univoca di cloud,anzi in questo caso viene data a seconda dei punti di vista considerati; basti pensare che in un articolo di qualche anno fa [19] , chiesta una definizione a ventuno esperti del settore IT, si sono praticamente ottenute altrettante definizioni diverse!

Per gli scopi di questa sezione, tuttavia, si accenna al cloud come tecnologia utile per i microservizi e quindi vedendolo come servizio [20] fornito da terze parti tramite un sistema distribuito.

Definito questo aspetto, si possono classificare vari modelli di cloud computing, di cui i tre più noti sono:

- SaaS (Software as a Service): permette l'utilizzo di software nel cloud, per applicazioni come email,desktop virtuali ed applicazioni per la comunicazione;esempi noti sono Google Apps, Amazon Web Services, Oracle.

- PaaS (Platform as a Service): permette il supporto per l'intero ciclo produttivo del software, in questo caso del microservizio, compresi il supporto per la distribuzione e per i test, esempi applicativi possono

essere i database, i server web e gli strumenti di sviluppo; fornitori noti sono IBM Bluemix, Microsoft Azure, Cloudbees.

- IaaS (Infrastructure as a Service): fornisce infrastrutture hardware o concettualmente vicine come connessioni di rete, dischi fissi per la conservazione dei dati e macchine virtuali;esempi noti sono Amazon EC2, Rackspace ed OpenStack.

Inoltre si posso distinguere i tipi di cloud in:

- Pubblici, dove il fornitore esterno gestisce l'intero cloud e nello stesso server condiviso ci possono essere le informazioni ed i dati di più clienti e dove lo stesso cliente accede al solo servizio che lo riguarda e non conosce il resto dell'hardware del server.

- Privati, dove, all'opposto, il cliente ha il controllo completo dei servizi, hardware e software; ed è naturalmente la situazione più comune per grandi realtà produttive che usino servizi cloud esterni.

- Ibridi, dove il cliente sceglie quali servizi tenere in cloud privati (ad esempio dati che ritiene sensibili) e quali tenere in cloud pubblici. Questa soluzione intermedia è usata tipicamente nella situazione in cui il cliente ritenga di voler limitare i costi rispetto alla soluzione privata.

Economicamente, l'utilizzo del cloud per parte dell'applicazione a microservizi può essere utile per risparmiare costi di gestione e manutenzione hardware e software e/o in fase iniziale di passaggio al cloud per una scalabilità pensata in modo graduale.

Tecnicamente, per un'applicazione a microservizi, una buona idea concettuale potrebbe essere ad esempio quella di utilizzare microservizi cloud esterni per le attività di monitoraggio e test degli altri microservizi, poichè in questo modo un guasto al server (o più server distribuiti) contenente i microservizi dell'applicazione non impedisce di accorgersene e mettere nuovamente in produzione il tutto ;viceversa se non fossero disponibili i servizi cloud, l'applicazione sarebbe comunque disponibile.

La decisione di distribuire l'applicazione completamente in strutture cloud è detta anche architettura serverless.

3.8 Container e Docker

Ogni microservizio, per costruzione, viene eseguito in un singolo processo ed è opportuno che venga distribuito e scalato in più host, in modo che la procedura sia automatizzata e comunque rapida e che non ci siano problemi di compatibilità con i diversi sistemi operativi. A questo scopo andrebbe bene anche una macchina virtuale, ma consumerebbe troppe risorse.

Fortunatamente, allo scopo esiste l'alternativa dei Container Linux (LXC) [21], i quali, a differenza delle macchine virtuali non simulano tutto l'hardware di un sistema operativo ospite, ma sono eseguiti direttamente come singolo processo del sistema operativo.

Per comprendere meglio tale differenza, si può osservare la figura [22] mostrata nella pagina seguente:

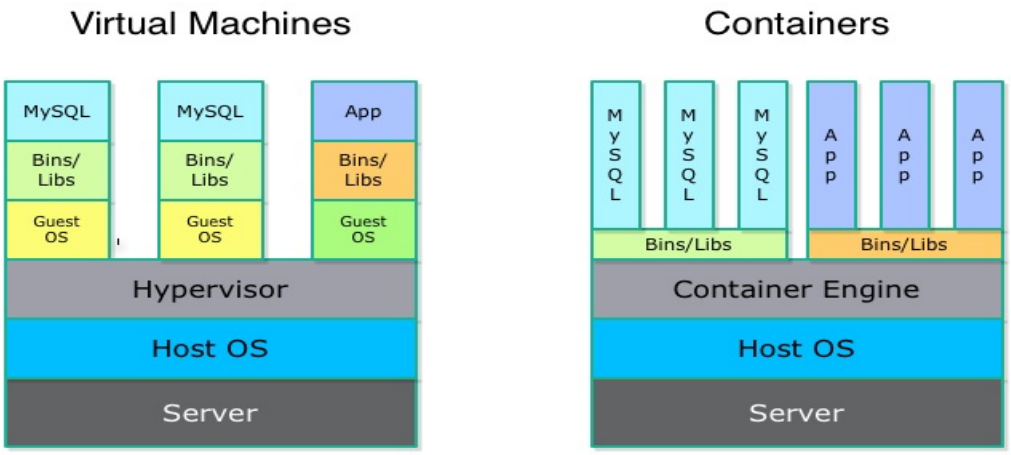

Figura 3.3: Confronto tra macchine virtuali e container Linux

Esempio di utilizzo di container è Docker [23], progetto sviluppato da dotCloud, fornitore di PaaS , poi rinominata in Docker, e rilasciato poi come progetto open source. Per una maggiore compatibilità, è installabile nelle versioni per Linux, Windows e Mac OS X. Nel primo caso userà semplicemente i meccanismi descritti come processo in esecuzione, negli altri casi avvia una leggera virtualizzazione di un kernel Linux, tuttavia è in sviluppo l'ulteriore compatibilità con gli altri sistemi operativi. Inoltre Docker offre anche il supporto cloud per Amazon Web Services (AWS) ed Azure.

Prima di essere messi in esecuzione, i container Docker vanno predisposti per poter "ospitare" il codice del microservizio da rendere disponibile ed attivo. Per questo e per tutti gli altri aspetti, il sito ufficiale di Docker fornisce una documentazione completa [24] .

Si scrive un file particolare, chiamato Dockerfile, che contiene le istruzioni accettate da Docker per costruire un'immagine per il container, spesso sca-

ricata da Docker Hub, registro pubblico delle immagini Docker, o da altri registri, ed eventuali comandi da eseguire all'interno del container in esecuzione. Tutto questo garantisce che, all'esecuzione di docker come processo del sistema operativo, possano essere eseguite le varie immagini contenenti i microservizi, avendo la sicurezza che scaricando l'immagine in altri host si attiva il microservizio nello stesso identico ambiente di sviluppo ed esecuzione, evitando così problemi di incompatibilità e favorendo la scalabilità.

Esistono altri meccanismi tra i quali si può citare docker-compose, che permette di scrivere un file per la composizione di più container collegati tra loro, per esempio un container con il codice composto con il container per il DBMS e quello per i dati del microservizio.

Per ogni operazione elencata e per molte altre situazioni, è possibile utilizzare le interfacce API fornite da Docker [25]; sono state sviluppate dashboards che permettono di utilizzarle (tali GUI chiamano appunto le operazioni messe a disposizione dalle API), facilitando in questo modo la gestione dei container ed il loro monitoraggio. Esempi possono essere Portainer[26] e Rancher[27].

Tutti gli aspetti descritti sin qui nel capitolo portano necessariamente ad un nuovo modo di sviluppare le applicazioni per le procedure di automazione, scalabilità, rilascio frequente di nuovi microservizi, in cui sono incoraggiate a collaborare persone che ricoprono diversi incarichi: sviluppatori, addetti al controllo qualità ed Information Technology (IT).

Questa metodologia è detta DevOps, da Developer ed Operations (sviluppatori ed operazioni) ed è solitamente ciclica per ogni microservizio [28] come in figura:

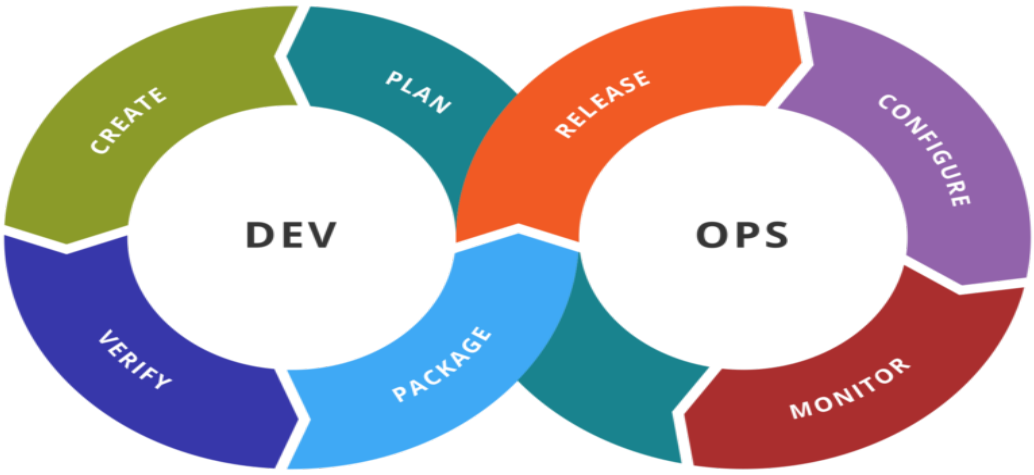

Figura 3.4: Ciclo DevOps

Si parte dallo sviluppatore e dal progetto del microservizio, quindi il punto iniziale per leggere la figura è 'Plan' di 'Dev' , procedendo in senso antiorario poi in senso orario:

- Gli sviluppatori progettano il microservizio
- Implementazione del microservizio
- Verifica con test e debug
- Si predispone il microservizio per il rilascio, per esempio in un container o simili
- A questo punto inizia la parte delle operazioni, soprattutto sistemistiche con il rilascio del microservizio
- Configurazione del microservizio
- Monitoraggio del microservizio

- Ricomincia il ciclo: si pianifica un aggiornamento e/o un nuovo microservizio da parte degli sviluppatori.

Per riferimento di una buona implementazione si possono considerare i '12 fattori' [29] :

- I. Codebase: Una sola codebase sotto controllo di versione, tanti deploy
- II. Dipendenze: Dipendenze dichiarate ed isolate
- III. Configurazione: Memorizza le informazioni di configurazione nell'ambiente
- IV. Backing Service: Tratta i backing service come "risorse"
- V. Build, release, esecuzione: Separare in modo netto lo stadio di build dall'esecuzione
- VI. Processi: Esegui l'applicazione come uno o più processi stateless
- VII. Binding delle Porte: Esporta i servizi tramite binding delle porte
- VIII. Concorrenza: Scalare attraverso il process model
- IX. Rilasciabilità: Massimizzare la robustezza con avvii veloci e chiusure non brusche
- X. Parità tra Sviluppo e Produzione: Mantieni lo sviluppo, staging e produzione simili il più possibile
- XI. Log: Tratta i log come stream di eventi
- XII. Processi di Amministrazione: Esegui i task di amministrazione/-management come processi una tantum

3.9 Jolie come linguaggio orientato ai microservizi

Jolie [30] è un linguaggio di programmazione orientato ai microservizi, il cui nome è in realtà l'acronimo di Java Orchestration Language Interpreter Engine [31] e presenta tre vantaggi rispetto ad altre soluzioni:

- Paradigma: lo sviluppatore scrive e progetta il codice pensando direttamente a microservizi, il tutto racchiudendo le due fasi in un unico passaggio, e senza pensare a come verrà eseguito il microservizio.
- Autosufficienza: eccetto l'installazione della Java Virtual Machine, Jolie non necessita di framework, server o altre strutture aggiuntive. La sigla che dà il nome al linguaggio infatti indica che è basato su Java ed è un linguaggio interpretato.
- Indipendenza: un microservizio Jolie non è legato ad alcun protocollo, anzi è possibile implementare un proprio protocollo e tra quelli supportati c'è anche SODEP (Simple Operation Data Exchange Protocol) [32], sviluppato appositamente per il linguaggio.

Un'altra caratteristica che Jolie si propone è quella di seguire il paradigma linguistico anzichè quello sistemistico, maggiormente diffuso ed utilizzato come descritto precedentemente nei container. L'idea alla base del paradigma è quella di concentrare tutti i principi dei microservizi in un unico linguaggio di programmazione, per un'applicazione in cui ogni singola entità sia comunque direttamente un microservizio pronto per essere eseguito.

Quindi:

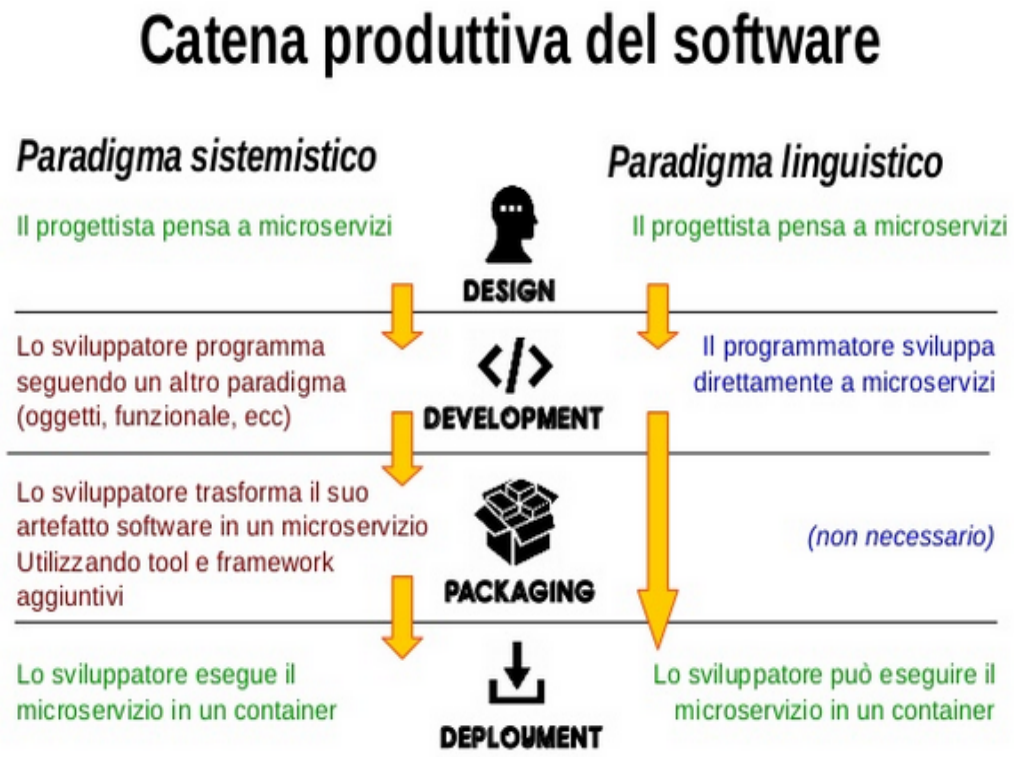

Figura 3.5: Confronto tra paradigma sistemistico e linguistico

Inoltre, per quanto riguarda la composizione, nel paradigma linguistico di Jolie un microservizio può contenere al suo interno altri microservizi (qui interviene l'orchestrazione presente nel nome), e questo rende anche più semplice spostare ciascuno di essi e comporre nuovamente l'architettura complessiva.

I due paradigmi sono compatibili, e la situazione ottimale, che sfrutta tutti i vantaggi, è nell'intersezione tra loro.

Riassumendoli:

- La progettazione e la programmazione sono in un'unica fase (paradigma linguistico)
- Per comporre nuove architetture a microservizi è sufficiente ricombinarli tra loro (paradigma linguistico)
- Libertà di scelta tecnologica (sistemistico)
- Esecuzione semplice e scalabile (sistemistico)
- Compatibilità con il cloud (sistemistico)

3.10 Possibile struttura dell'applicazione basata su microservizi

Per chiudere il capitolo, si esaminerà ora una possibile struttura di una generica applicazione a microservizi, considerando tutti i pattern più diffusi. L'esempio del capitolo successivo prenderà in considerazione una minima parte di questi aspetti, lasciando comunque alla lettrice e/o al lettore la possibilità di ampliare ed approfondire le numerose implicazioni dei microservizi.

Si può notare che parecchi elementi che compaiono nella figura di riferimento, presa dal sito microservices.io [9], sono stati discussi precedentemente; verranno quindi semplicemente aggiunte alcune considerazioni.

Questo metodo di descrizione dei pattern possibili in un contesto specifico prende anche il nome di Pattern Language.

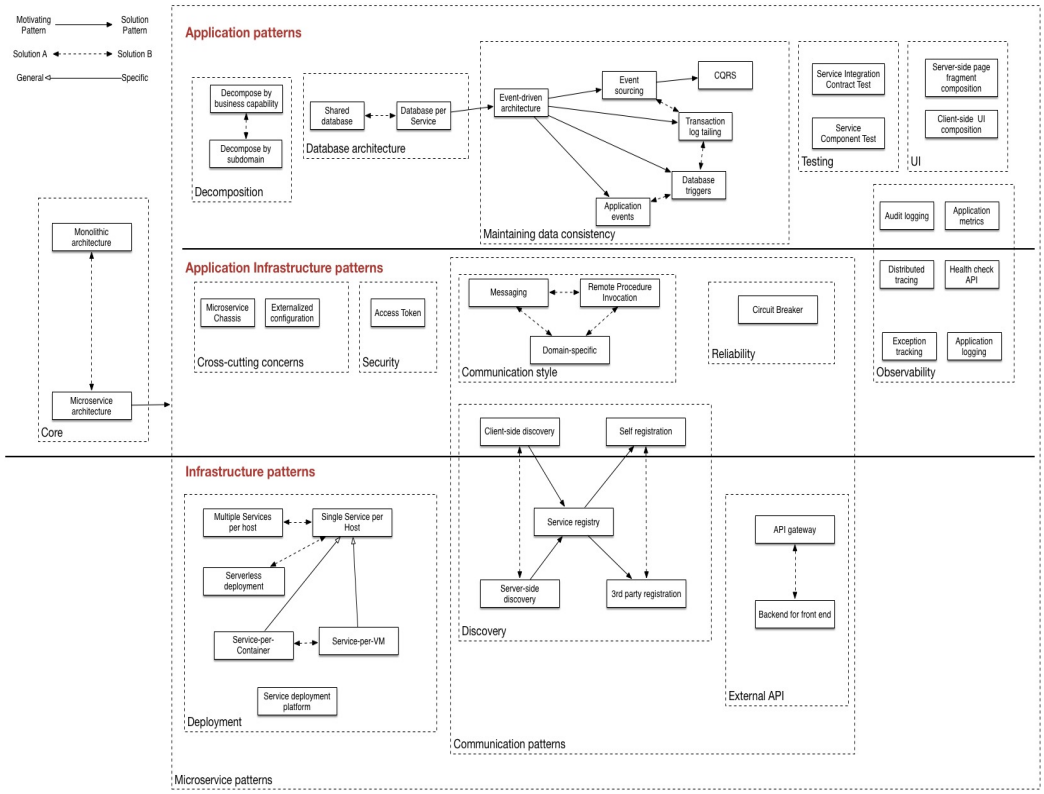

Figura 3.6: Pattern language dei microservizi

Seguendo quindi lo schema una possibile scelta potrebbe essere:

- Microservizi
- Scomposizione: possibili entrambe le opzioni
- Database distribuiti per microservizio e non condiviso
- Utilizzo dei microservizi che gestiscano eventi come già accennato come tramite tra altri microservizi.
- Conseguenza del punto precedente: CQRS (Command Query Responsibility Segregation), i comandi effettuano solo operazioni di scrittura, le query solo operazioni di lettura, divisione in ulteriori operazioni sui

microservizi riguardanti la stessa entità in cui un microservizio gestisce le operazioni di lettura ed uno di scrittura. Questo fornisce anche un vantaggio nel teorema CAP: se fallisce una operazione di scrittura su database, leggo un messaggio dall'altro microservizio ed intanto garantisco una risposta al chiamante originale.

- Utilizzo dei log a loro volta come eventi
- Test sui microservizi
- Interfaccia UI/GUI lato server
- Monitor, Metriche, Tracciabilità con gli strumenti descritti in precedenza
- Possibilità di configurazioni esterne
- Comunicazione tramite messaggi
- Più microservizi per host con possibilità di utilizzare container
- Ricerca dei microservizi da chiamare dinamicamente al variare della richiesta dell'utente tramite registro e lato server
- API Gateway come primo microservizio "incontrato" dalla richiesta lato client
- Aspetto sicurezza: un solo microservizio o una sola serie comunque staccata dai microservizi che la richiedono; possibili vari metodi tramite token

- Pattern circuit breaker: per l'affidabilità dell'applicazione si può inserire un microservizio tra il richiedente ed il microservizio "sensibile" : se qualcosa non funziona, come un numero eccessivo di richieste, un timeout, un guasto o una serie di errori, il microservizio facente funzione di circuit breaker, come dice il nome 'chiude il circuito' quindi non lascia passare ulteriori richieste e risponde con un messaggio di errore al richiedente, 'proteggendo' così il microservizio successivo.

Capitolo 4

Esempio di applicazione a microservizi

4.1 Descrizione dell'applicazione

L'applicazione di esempio consiste nel mostrare all'utente che si collega via browser, una pagina contenente un logo, un menù di navigazione, l'anteprima della notizia più recente, un esempio di catalogo prodotti e la situazione meteo di Cesena, i cui dati sono presi da un servizio esterno, OpenWeatherMap [33].

Per esporre in maniera scorrevole i concetti presi in considerazione, verranno inseriti alcuni dei codici sorgente dei file, descrivendo come riprodurre gli stessi principi per più microservizi (spesso si tratta di operazioni di copia/incolla e modifica di pochissimi particolari).

Come idea generale, segue una figura con un possibile risultato finale :

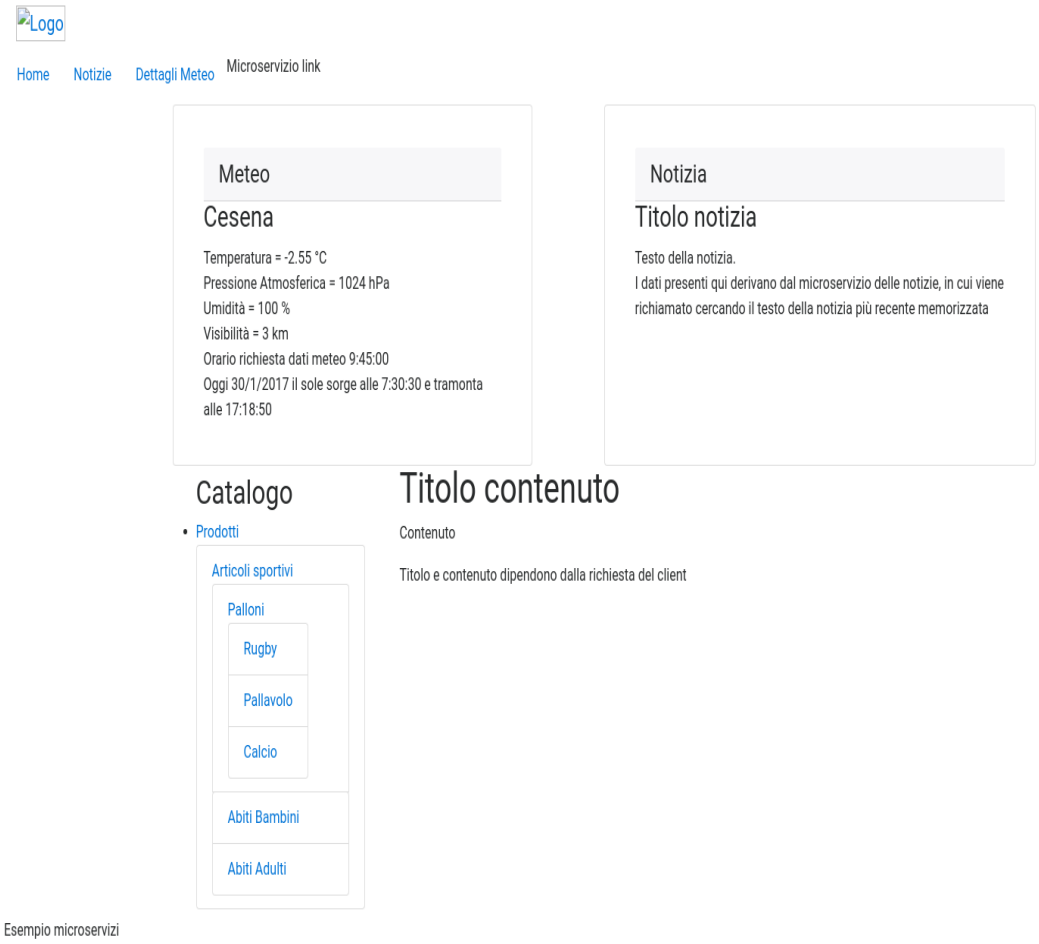

Figura 4.1: Esempio di risultato in risposta all'elaborazione dei microservizi

La parte centrale, quella del contenuto, varierà a seconda della richiesta effettuata dall'utente.

4.2 Progetto della divisione in microservizi

Stabilita la descrizione dell'applicazione, e quindi i suoi requisiti, occorre progettarne i microservizi e la loro distribuzione. In questo esempio di base, si notano subito le parti che possono essere un microservizio, e tenendo presente

che potrebbero essere a loro volta scomponibili in microservizi.

In particolare:

- Microservizio Catalogo: può avere ad esempio le operazioni per leggere l'elenco delle categorie,o visualizzare i dettagli di un specifico prodotto, se richiesto come parametro.
- Microservizio Notizie: può cercare la notizia più recente, leggere l'elenco delle notizie,visualizzare una specifica notizia in dettaglio
- Microservizio Meteo: effettua una richiesta ad OpenWeatherMap, ne riceve un messaggio, lo elabora
- Microservizio Link: per i link del menù di navigazione
- Inoltre, occorrono microservizi per costruire e comporre la GUI per ognuno dei punti elencati sopra.
- Infine, la parte di composizione della pagina, avrà un microservizio dedicato a raccogliere ogni frammento di pagina di ciascun microservizio che preveda una GUI e quindi a restituire in risposta il codice HTML della pagina completa (i dettagli su tutti questi aspetti saranno spiegati nella prossima sezione)
- Microservizio Registro dei microservizi da chiamare, con la funzione, descritta nel capitolo precedente, di "Service Discovery"
- Microservizio "Server": punto di partenza e primo microservizio "incontrato" dal client appena arriva la richiesta;siccome in questo esempio si ipotizza l'utilizzo di un browser come client, tale microservizio sarà in

ascolto con protocollo http sulla porta 80, o in alternativa in https sulla porta 443. Se si prevede l'uso di un client come applicazione desktop complessa, si possono applicare altri principi all'intero progetto, come ad esempio spostare alcuni microservizi ed in particolare registro e/o compositore della pagina interamente lato client.

4.3 Progetto dettagliato della struttura dell'applicazione

L'applicazione sarà quindi così generalmente strutturata: l'utente via browser accede al sito/host dove c'è il microservizio server d'ingresso: questo chiamerà il microservizio apposito per ottenere i parametri necessari a chiamare gli altri microservizi dell'applicazione cioè host, porta, protocollo, formato dati, eventuali parametri tra cui eventualmente il nome del microservizio (raggiungibile comunque in maniera distribuita ed univoca tramite host e porta). Inoltre il registro potrebbe avere la possibilità, nel caso in cui un microservizio venga scalato e replicato, di applicare o fare applicare ad ulteriori microservizi algoritmi di "load balancing".

Ottenuti questi dati, il microservizio server chiama iterativamente e dove possibile in parallelo i vari microservizi, dai quali riceverà rispettivi messaggi di risposta ed eventualmente la loro parte di pagina/UI/GUI da far visualizzare all'utente; in tal caso, verrà passato un messaggio contenente la risposta ricevuta al microservizio compositore della pagina, che assembla tutte le parti insieme ed ottenuta l'intera pagina, la invia come risposta al microservizio

server, che a sua volta la fornisce come risposta all'utente!

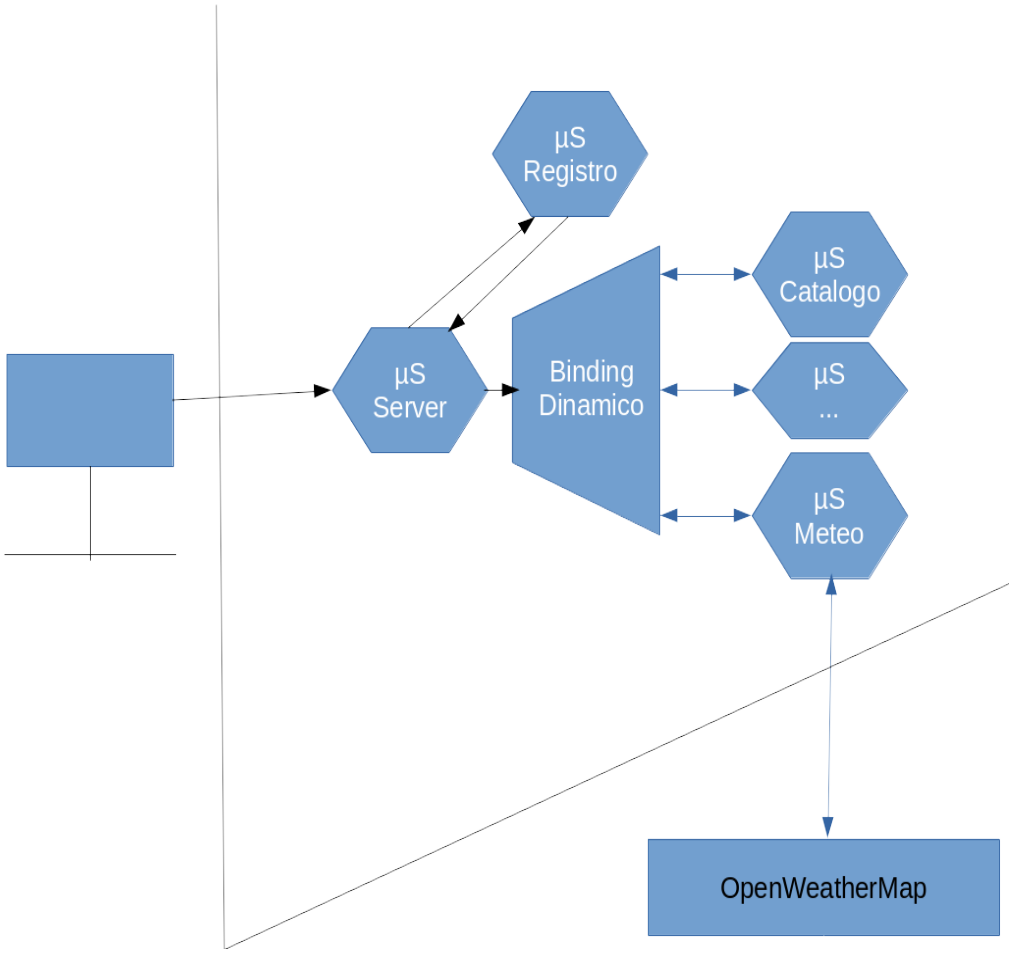

Figura 4.2: Esempio di richiesta e risposta, le linee indicano un passaggio di host, quindi la distribuzione del sistema. Inoltre, ciascun microservizio chiamato tramite binding dinamico può essere in host differenti dal microservizio server

4.4 Implementazione dell'applicazione

In questa sezione viene descritto un esempio di implementazione dell'applicazione, talvolta con il codice utilizzato, in ogni caso con indicazioni alla lettrice e/o al lettore su possibilità implementative ed esempi sul risultato che si attende in linea di principio, lasciando, come in tutti gli aspetti, ampia possibilità di scelte alternative a quelle qui descritte.

Per iniziare, il microservizio server: per il suo ruolo di punto di ingresso iniziale e per il fatto che si serve di altri microservizi con cui compie le principali operazioni di orchestrazione dell'intera applicazione e per la necessità di avere capacità di "binding dinamico" per chiamare altri microservizi, una possibile scelta implementativa ricade nell'uso del linguaggio Jolie. Dal punto di vista dei principi dei microservizi, inoltre, il binding dinamico consente importanti conseguenze per scalabilità e flessibilità: aggiungere nuovi microservizi non richiede alcuna modifica al microservizio server!

microservizi/server.ol

```
1  include "console.iol"
2  include "string_utils.iol"
3  include "protocols/http.iol"
4
5  execution { concurrent }
6
7  type risposta : string
8
9  interface HTTPInterface {
10 RequestResponse:
11    richiesta(DefaultOperationHttpRequest)(risposta)
```

```
12 }
13
14 inputPort input {
15    Location: "socket://localhost:80"
16    Protocol: http{
17       .keepAlive = true;
18       .default="richiesta";//;
19       .format="html"
20    }
21    Interfaces: HTTPInterface
22 }
23
24 type parametri: void {
25   .nome[1,*]: string
26 }
27
28 type lista : void {
29    .id [1,*]: string {
30       .ingresso : string
31       .parametri* : string | int
32       .host : string
33       .porta : int
34       .protocollo: string
35       .formato: string
36    }
37 }
38
39 interface registro {
40 RequestResponse: elencoMicroservizi(parametri)(lista)
41 }
```

```
42
43 outputPort registro {
44    Location: "socket://localhost:9000"
45    Protocol: sodep
46    Interfaces: registro
47 }
48
49 /* Binding dinamico per le richieste ai microservizi */
50
51 type risposta_microservizio : string
52
53 type dati_messaggio_microservizio: void {
54   .microservizio: string
55   .dati: string
56 }
57
58 interface microservizio {
59   RequestResponse: messaggio (dati_messaggio_microservizio) (
       risposta)
60 }
61
62 outputPort microservizio {
63    Interfaces: microservizio
64 }
65
66 // ... per la risposta dalla composizione della pagina [...]
67
68 type pagina:string
69
70 interface pagina {
```

```
71  RequestResponse: aggiorna (risposta_microservizio) (pagina)
72 }
73
74 outputPort pagina {
75    Location: "socket://localhost:2121"
76    Protocol: http {.format = "html"}
77    Interfaces: pagina
78 }
79
80 main {
81
82     richiesta(client)(risposta){
83
84       install( TypeMismatch =>
85             println@Console( "TypeMismatch: " + main.
                  TypeMismatch )()
86        );
87        install( IOException =>
88             println@Console( "IOException: " + main.IOException
                  )()
89        );
90
91       println@Console("Richiesta : "+client.requestUri)();
92       client.requestUri.regex = "/";
93       split@StringUtils(client.requestUri)(s);
94       parametri.nome << s.result;
95       parametri.nome[0] = client.operation;
96
97       elencoMicroservizi@registro(parametri)(lista);
98        println@Console("Microservizi da elaborare : "+#lista.id)
```

```
            ();
99
100         for (i=0, i<#lista.id, i++) {
101               // binding dinamico
102               microservizio.location="socket://"+lista.id[i].host
                      +":"+lista.id[i].porta;
103               microservizio.protocol=lista.id[i].protocollo;
104             microservizio.protocol.format=lista.id[i].formato; //
                     caso http/https specifico il formato
105             dati_messaggio_microservizio.microservizio = lista.id
                    [i].ingresso;
106             dati_messaggio_microservizio.dati = lista.id[i].
                    parametri;
107             println@Console("Microservizio da chiamare : "+lista.
                    id[i])();
108               messaggio@microservizio(
                      dati_messaggio_microservizio)(
                      risposta_microservizio); //invio il messaggio al
                       microservizio specifico
109               println@Console("Microservizio "+lista.id[i]+" ha
                      risposto : "+risposta_microservizio)();
110             aggiorna@pagina(risposta_microservizio)(pagina)
111         };
112
113
114
115         risposta = pagina
116
117     }
118
```

```
119 }
```

In breve, quanto descritto precedentemente riguardo ai requisiti di questo microservizio si può notare dalla riga 80 da cui si legge la richiesta dell'utente, si chiama il registro e si ottiene la lista dei servizi, mentre il "for" dalla riga 100 permette iterativamente e dinamicamente di chiamare i microservizi applicativi ottenuti dal registro, del quale è riportato, sempre utilizzando il linguaggio Jolie, un esempio di "prototipo" che elenca "staticamente" i microservizi ed i loro parametri, e può essere ampliato collegandolo ad un DBMS e/o ad altro microservizio di service discovery o, se si utilizzano container, implementato usando le interfacce API di Docker.

Inoltre è dichiarata una "outputPort microservizio" e relativa interfaccia, oltre ad alcune strutture dati di ingresso e di risposta; non specificando "Location" e "Protocol", si sfrutta il meccanismo per il generico microservizio, utilizzando il binding dinamico.

microservizi/registro.ol

```
1  include "console.iol"
2  include "file.iol"
3  include "string_utils.iol"
4  include "protocols/http.iol"
5  include "runtime.iol"
6
7  execution { concurrent }
8
9  type parametri: void {
10   .nome[1,*]: string
11 }
```

```
12
13 type lista : void {
14    .id [1,*]: string {
15       .ingresso : string
16       .parametri* : string | int
17       .host : string
18       .porta : int
19       .protocollo: string
20       .formato : string
21    }
22 }
23
24 interface registro {
25 RequestResponse:
26    elencoMicroservizi(parametri)(lista)
27 }
28
29 inputPort registro {
30    Location: "socket://localhost:9000"
31    Protocol: sodep
32    Interfaces: registro
33 }
34
35 main {
36     elencoMicroservizi(par)(lista){
37         println@Console("Chiamata operazione elencoMicroservizi
               nel registro!")();
38
39       // Lista dei microservizi costruita staticamente,
            prototipo scalabile richiamando un DBMS e memorizzando
```

```
          i dati
40      // e/o utilizzando API Docker per la lista dei container
          contenenti microsevizi da chiamare
41        lista.id[0] = "default";
42          lista.id[0].ingresso = "/";
43          lista.id[0].parametri[0] = par.nome[0];
44          lista.id[0].host = "localhost";
45          lista.id[0].porta = 10000;
46          lista.id[0].protocollo = "http";
47          lista.id[0].formato = "json";
48       lista.id[1] = "meteo";
49         lista.id[1].ingresso = "/";
50         lista.id[1].parametri[0] = par.nome[0];
51          lista.id[1].host = "localhost";
52          lista.id[1].porta = 10011;
53          lista.id[1].protocollo = "http";
54          lista.id[1].formato = "json";
55           lista.id[2] = "notizia";
56         lista.id[2].ingresso = "/";
57         lista.id[2].parametri[0] = par.nome[0];
58          lista.id[2].host = "localhost";
59          lista.id[2].porta = 10012;
60          lista.id[2].protocollo = "http";
61          lista.id[2].formato = "json"
62
63       // ... eccetera per gli altri microservizi applicativi
64
65    }
66
67 }
```

Questi esempi Jolie permettono alcune brevi considerazioni sul linguaggio: con ”inputPort” si dichiara una (o più) porta in ascolto in ingresso, e qualsiasi messaggio in qualsiasi protocollo e formato riconosciuto da mittente e ricevente indirizzato all'host e alla porta corretti, saranno ricevuti, senza dover pensare ad implementazioni a più basso livello di astrazione, poichè già inserite nel linguaggio; analogamente, con ”outputPort” si dichiara una (o più) porte in uscita, specificando anche il nome dell'interfaccia del microservizio ed il mezzo di comunicazione, nel caso specifico le socket.

Arrivati a questo punto, occorre implementare ed avviare i punti di ingresso degli altri microservizi, che elaboreranno il messaggio (nel caso specifico in formato JSON), chiameranno eventualmente altri microservizi per elaborare i dati e costruire la propria eventuale parte di GUI, e restituiranno un messaggio di risposta al microservizio server. Si può sfruttare il fatto che la comunicazione avviene tramite socket, quindi per ogni outputPort Jolie che scriverà sulla socket, dovrà essere in ascolto il microservizio per la lettura della stessa.

Per fare un esempio pratico di come un'applicazione a microservizi possa coinvolgere varie tecnologie e linguaggi di programmazione, di come si possa sfruttare la libertà di scelta in questo modo, e di come si possa riutilizzare codice già esistente, i punti di ingresso ed il codice rimanente saranno implementati utilizzando il linguaggio PHP [34], che già prevede i meccanismi di lettura di socket concorrenti non bloccanti e di interpretazione di messaggi in formato JSON.

Segue l'esempio generico, commentato con le considerazioni:

```
<?php /* PHP Server socket in ascolto sulla porta
   10101. Funziona come punto di ingresso di un
   microservizio PHP */

$socket = socket_create(AF_INET, SOCK_STREAM, 0) or
   exit('Errore nella creazione della socket!');
socket_bind($socket, 0,10101);
socket_listen($socket);
socket_set_nonblock($socket); //Rendo la socket non
   bloccante
error_reporting(0*E_ALL);
$lista_client=array(); //Eventuale memorizzazione
   lista client connessi

for (;;) {

    if (($client = @socket_accept($socket))!== false)
       {
        socket_set_nonblock($client); //Rendo la
           socket non bloccante

        $pid = pcntl_fork();

        if ($pid == -1) {
```

```
            exit("errore nella fork");
        } else if ($pid) { // processo principale
echo "Ricevuta richiesta ed effettuata fork.";
echo "Torno in attesa di nuovi client!\n";
            continue;
        } else { // elaboro richiesta singolo client
//Visualizzato su standard output, eventualmente
   collegabile ad un microservizio di log
echo "Richiesto microservizio di esempio.\n";
$richiesta = socket_read($client,1024);
$dati_richiesta = json_decode(explode("\r\n\r\n",
   $richiesta."\r\n\r\n")[1],true);
$tipo=""; // Tipo di contenuto della risposta , inteso
    come Content-Type per header http

$parametri = array_reverse(explode("/",
   $dati_richiesta["dati"]."/",-1));

ob_start(); //Preparo output buffer PHP per alcune
   elaborazioni

/* Preparo l'array $risposta, ricodificandolo in
   formato JSON prima di inviarlo come messaggio di
   risposta */
```

```
        $risposta["microservizio"] = "meteo";
        $risposta["ui"] = true;
        $risposta["stato"] = "ok";

        $tipo ="text/html";

/* Qui si possono avere varie istruzioni, richiamare
   altro codice e/o le variabili per costruire la GUI
   */
        #include 'qualcosa.php';
        $risposta["messaggio"]= ob_get_contents();
ob_end_clean();

        $risposta["dati"]="";
        $risposta = json_encode($risposta);

/* Preparo gli header di risposta, qui con stato HTTP
   200 tutto ok */
        $headers_http_risposta = "HTTP/1.1 200 OK\r\n"
            .
                "Content-length: " . strlen($risposta)
                   . "\r\n" .
                "Content-Type: $tipo; charset=UTF-8\r\n
                   \r\n";
```

```
        socket_getpeername($client, $ip,$porta);
          $idc = $ip.":".$porta;
/* Scrivo la risposta sulla socket, quindi la invio al
    client che ha fatto la richiesta */
socket_write($client,$headers_http_risposta.$risposta,
   strlen($headers_http_risposta)+strlen($risposta));

          exit("Elaborazione index client $idc terminata
             !\n\n\n");



        } // elaborazione richiesta singolo client
    } else usleep(100000); // ottimizzazione uso CPU
       dopo ogni richiesta (0.1 s)
} // for
?>
```

Per collegare i microservizi, è sufficiente modificare questo esempio generico, sostituendo la porta 10101 con la porta in cui avviene la scrittura su socket da parte del microservizio che utilizza questo; quindi, in questo caso di applicazione, occorre far corrispondere le porte ottenute dal microservizio del registro (10000,10011,eccetera) e salvare tanti file PHP quanti sono questi microservizi. Inoltre, nella parte di elaborazione della richiesta, dopo la decodifica del messaggio JSON, nella parte dove si include un ulte-

riore file, si possono effettuare molte elaborazioni, successivi richiami ad altri microservizi, ed ottenere la risposta da inviare.

Per esempio, il microservizio meteo è il codice di esempio appena esposto, modificando la porta da 10101 a 10011 ed aggiungendo, dopo "parametri"

```
<?php
$stringa_api_openweathermap = "http://api.
   openweathermap.org/data/2.5/weather?q=Cesena&appid
   =62447b3d5d2b5e5d35499aec63ea35c9&lang=it&type=
   accurate&units=metric";
$dati_meteo = json_decode(file_get_contents(
   $stringa_api_openweathermap),true);
$orario_richiesta = 1+(getdate()['hours']).":".getdate
   ()['minutes']."":".getdate()['seconds'];
$oa = 1+getdate($dati_meteo['sys']['sunrise'])['hours
   ']."":".getdate($dati_meteo['sys']['sunrise'])['
   minutes']."":".getdate($dati_meteo['sys']['sunrise
   '])['seconds'];
$ot = 1+getdate($dati_meteo['sys']['sunset'])['hours
   ']."":".getdate($dati_meteo['sys']['sunset'])['
   minutes']."":".getdate($dati_meteo['sys']['sunset'])
   ['seconds'];
?>
```

ed includendo il file meteo.php al posto di qualcosa.php

```
<div class="card-block">
<h4 class="card-header">Meteo</h4>
<h3 class="card-title text-centered">Cesena</h3>
<p class="card-text">
Temperatura = <?php echo $dati_meteo['main']['temp']; ?> °C<br>
Pressione Atmosferica = <?php echo $dati_meteo['main']['pressure']; ?> hPa<br>
Umidità =    <?php echo $dati_meteo['main']['humidity']; ?> %<br>
Visibilità = <?php echo $dati_meteo['visibility']/1000; ?> km<br>
Orario richiesta dati meteo <?php echo $orario_richiesta; ?><br>
Oggi <?php echo date('j')."/".date('n')."/".date('Y'); ?> il sole sorge alle <?php echo $oa; ?> e tramonta alle <?php echo $ot; ?>
</p>
</div>
```

si ottiene l'elaborazione completa dei dati da spedire al microservizio server.

In tal modo, l'unico microservizio che resta ora da analizzare è quello di composizione della pagina, che ricevendo le varie parti, ha il compito di inserirle nel giusto ordine. In questa applicazione, è stato scritto un microservizio "pagina principale" (chiamato default nel codice del registro) contenente il codice HTML che si pensa essere in comune alla struttura della pagina, e che viene analizzato per primo dal compositore:

```
<!doctype html>
<html lang="it">


<head>
    <meta charset="UTF-8">
    <title>Microservizi</title>
    <meta name="description" content
       ="">
```

```
                <meta name="viewport" content="
                    width=device-width, initial-
                    scale=1, shrink-to-fit=no">
                <meta http-equiv="x-ua-compatible"
                     content="ie=edge">
                <!-- <link rel="favicon" href="
                    icona.png"> -->
                <link rel="stylesheet" href="https
                    ://maxcdn.bootstrapcdn.com/
                    bootstrap/4.0.0-alpha.6/css/
                    bootstrap.min.css" integrity="
                    sha384-rwoIResjU2yc3z8GV/
                    NPeZWAv56rSmLldC3R/
                    AZzGRnGxQQKnKkoFVhFQhNUwEyJ"
                    crossorigin="anonymous">
                <link rel="stylesheet" type="text/
                    css" href="css/flexbox.css">
            </head>
        <body>

        <header class="navbar">
          <a class="navbar-brand" href="#">
            <img src="logo.png" alt="Logo">
          </a>
```

```
</header>

<main class="container">

<nav id="link" class="row">

</nav>

  <div id="meteo" class="card card-block
     offset-xs-1 col-xs-5">

                  </div>

                  <div id="notizia" class="card
                      card-block offset-xs-1
                     col-xs-5">

                  </div>

<section id="pagina" class="row rowflex">
```

```
<section class="rowflex col-md-3">  <!--
   inserisco sezione interna , nel caso in
   cui oltre al catalogo sulla sinistra
   volessi inserire altri contenuti-->

<nav id="catalogo" class="navbar">




 </nav>
</section> <!-- fine sezione interna -->

                    <section id="contenuto"
                       class="offset-md-0 col-
                       md-9">


                    </section>
    </section>
            </main>
            <footer>
Esempio microservizi
            </footer>
```

```
<script src="https://ajax.googleapis.com/ajax/
   libs/jquery/3.1.1/jquery.min.js" integrity
   ="sha384-3
   ceskX3iaEnIogmQchP8opvBy3Mi7Ce34nWjpBIwVTHfGYWQS9jwHDV
   " crossorigin="anonymous"></script>
<script src="https://cdnjs.cloudflare.com/ajax
   /libs/tether/1.3.7/js/tether.min.js"
   integrity="sha384-
   XTs3FgkjiBgo8qjEjBk0tGmf3wPrWtA6coPfQDfFEY8AnYJwjalXCi
   " crossorigin="anonymous"></script>
<script src="https://maxcdn.bootstrapcdn.com/
   bootstrap/4.0.0-alpha.6/js/bootstrap.min.js
   " integrity="sha384-BLiI7JTZm+
   JWlgKa0M0kGRpJbF2J8q+
   qreVrKBC47e3K6BW78kGLrCkeRX6I9RoK"
   crossorigin="anonymous"></script>


      </body>
```

```
</html>
```

Si può osservare che tale struttura contiene alcuni elementi con attributo id, che hanno la funzione di "segnaposto" per gli altri microservizi; il compositore ha lo scopo di prendere i frammenti di pagina dagli altri microservizi ed aggiornare il codice HTML, per esempio se il microservizio meteo fornisce un messaggio JSON che, decodificato ed interpretato, contiene :

```
<head>
        <link rel="stylesheet" type="text/css" href="meteo.css">
</head>

<body>
<section id="meteo">
<div class="card-block">

                        <h4 class="card-header">Meteo</h4>
                        <h3 class="card-title text-centered">Cesena<
                        <p class="card-text">
                        Temperatura = -2.55 C
                        Pressione Atmosferica = 1024 hPa
                        Umidita' =         100
                        Visibilita' = 3 km
                        Orario richiesta dati meteo 9:44:56
                        Oggi 30/1/2017 il sole sorge alle 7:30:30 e
```

```
</p>
</div>
</section>
<script type="text/javascript" src="meteo.js"></script>
</body>
```

avremo che il risultato parziale della pagina sarà (sono riportati solo le parti modificate :

```
<!doctype html>
<html lang="it">

<head>
    <meta charset="UTF-8">
    <title>Microservizi</title>
    <meta name="description" content="">
    <!-- ... -->

    <link rel="stylesheet" type="text/css" href="

</head>
<body>

  <!-- ... -->

  <div id="meteo" class="card card-block offset-xs-1 col-
```

```
            <div class="card-block">

                    <h4 class="card-header">Meteo</h4>
                    <h3 class="card-title text-centered">Cesena<
                    <p class="card-text">
                    Temperatura = -2.55 C
                    Pressione Atmosferica = 1024 hPa
                    Umidita' =         100
                    Visibilita' = 3 km
                    Orario richiesta dati meteo 9:44:56
                    Oggi 30/1/2017 il sole sorge alle 7:30:30 e
</p>
          </div>
          </div>

                    <!-- ... -->
                <footer>
        Esempio microservizi
                </footer>
```

```
    <script src="https://ajax.googleapis.com/ajax/libs/jquery
    <script src="https://cdnjs.cloudflare.com/ajax/libs/tethe
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/4.

  <script type="text/javascript" src="meteo.js"></script> <!-

        </body>

        </html>
```

Ripetendo il procedimento per ogni microservizio, si può ottenere il risultato della figura iniziale di questo capitolo!

Per chiudere la sezione, per mettere in esecuzione da terminale e/o da script shell l'intero sistema, per i microservizi Jolie il comando è

jolie nomeservizio.ol

(quindi jolie server.ol per esempio)

per PHP

/percorso/interprete/php /percorso/nome/file/php

Nel caso si utilizzino porte sino alla 1024, per attivare i microservizi occorerranno i permessi di root.

4.5 Considerazioni sull'applicazione

L'applicazione così implementata è già un minimo esempio a cui poi applicare i principi descritti nel capitolo sui microservizi.

Seguono altre brevi considerazioni:

- Un esempio per introdurre i pattern ad eventi e CQRS potrebbe essere l'introduzione della ricerca di notizie riguardanti un certo prodotto; per i principi dei microservizi non conviene che i microservizi di notizie e catalogo comunichino direttamente, quindi per esempio il microservizio di ingresso per il catalogo scriverà un evento "RichiestaNotizia" con un messaggio contenente il nome del prodotto. I microservizi delle notizie leggeranno l'evento e restituiranno un messaggio con le notizie riguardanti il prodotto. Inoltre si possono separare i microservizi in operazioni di lettura e scrittura come già esposto nel precedente capitolo riguardo CQRS.

- Usando i container, è anche possibile predisporli, tramite Dockerfile e docker-compose,in modo da inserire i sorgenti da compilare per alcuni microservizi, per esempio compilare una certa versione di PHP con parametri e librerie sicure per chi sviluppa; stesso discorso per i server web, anche per parti dell'applicazione, per esempio nginx [35] che può anche garantire https e load balancing, in questo modo si possono avere più punti in cui viene bilanciato il carico delle richieste ed un'applicazione ancor più ottimizzata.

- Per quanto riguarda Jolie, gli implementatori del linguaggio hanno anche scritto un server web, Leonardo [36]

Capitolo 5

Conclusioni e possibili sviluppi futuri

5.1 Obiettivi raggiunti

Nello svolgimento della tesi è stato esplorato il tema dei microservizi da vari punti di vista: prima di tutto quello "storico", partendo con l'analisi delle architetture monolitiche ed analizzandone le applicazioni di utilizzo, vantaggi e svantaggi; poi sono state analizzate le architetture SOA, risultato delle evoluzioni dei sistemi distribuiti e dei requisiti sempre maggiori richiesti alle applicazioni, per poter arrivare a comprendere come si è arrivati gradualmente all'utilizzo dei microservizi.

Un altro punto di vista, quello in effetti più analizzato, è stato quello teorico, con l'analisi dettagliata dei numerosi aspetti che caratterizzano i microservizi, dall'attenzione alla fase di progettazione di un'idea di applicazione con tale tecnica, considerando vantaggi e requisiti da rispettare e man-

tenere alle tecnologie ed ai linguaggi che facilitano tutte le fasi di sviluppo nonostante la maggior complessità architetturale nel suo insieme.

Infine, il punto di vista pratico ed implementativo, dove si è fornito un esempio di applicazione che prendesse in considerazione alcune delle tante scelte possibili per i microservizi, e dove si sono esplorate soluzioni per ampliare l'applicazione stessa.

Pertanto, dati gli obiettivi iniziali della tesi, non resta che, nelle prossime sezioni, parlare delle prospettive dei microservizi, cioè i possibili sviluppi futuri e fare alcune riflessioni conclusive.

5.2 Sviluppi futuri

I possibili sviluppi futuri dei microservizi si possono considerare da due punti di vista: tecnico/implementativo ed organizzativo.

Nel corso del capitolo sui microservizi si è osservato come inevitabilmente l'architettura porti complessità ed in alcuni casi la coordinazione necessaria tra diversi gruppi di lavoro, seppur ridotta di molto con i microservizi, sia comunque necessaria ed inoltre all'aumentare della complessità dell'applicazione non sempre è facile individuare una visione di insieme per capire come scalare ed aggiungere elementi al sistema. A questo scopo, come ulteriore miglioramento dell'orchestrazione e coordinamento nelle architetture a microservizi, è oggetto di ricerca [37] il tema della programmazione coreografica: si ha una visione architetturale complessiva dell'applicazione, specificando con apposite istruzioni del linguaggio la coreografia tra le varie parti, in questo

modo si evitano ”deadlock” nell'applicazione a microservizi per costruzione del linguaggio ed è immediata la visione architetturale complessiva.

Si segnalano due linguaggi orientati alle coreografie oggetto di ricerca :

- AIOCJ [38] (Adaptive Interaction Oriented Choreographies Jolie), in grado di adattare le coreografie in esecuzione e con la caratteristica di descrivere il sistema distribuito nel suo insieme. Compilandolo, fornisce sorgenti di microservizi Jolie per ogni processo.

- Chor [39], linguaggio coreografico con gli stessi scopi e anch'esso fornisce sorgenti Jolie dalla compilazione

AIOCJ, così come Jolie, sono stati e/o sono attualmente oggetto di ricerca all'interno di questa Università.

Mentre dal punto di vista produttivo ed organizzativo, adottare i microservizi porta, oltre ad una riorganizzazione dei gruppi di lavoro come detto precedentemente, anche all'adottare un nuovo approccio ed una nuova mentalità necessaria [40] con cicli di sviluppo ridotti, innovazione e sperimentazione frequente, prodotti come parte di integrazioni e sistemi più ampi, collaborazione interna ed esterna tra produzione e ricerca; questo perchè le industrie tendono e tenderanno a diventare anche ”aziende di software” con la digitalizzazione (concetti noti anche con il nome di Industria 4.0).

5.3 Conclusioni

Concludendo, i microservizi offrono possibilità di utilizzo nelle applicazioni di oggi e sviluppi futuri [41] teorici e pratici che potranno spaziare in svariati

campi della tecnologia informatica e non solo, cambiandone potenzialmente la metodologia di sviluppo ed incoraggiando la collaborazione tra le persone, portando così a progressi e scenari estremamente interessanti!

Ringraziamenti

Bene, eccoci ai ringraziamenti.

Grazie a te, lettrice e/o lettore che hai letto tutta la tesi fino qui, grazie lo stesso se stai leggendo solo i ringraziamenti.

Grazie ovviamente alla mia famiglia per il supporto in tutti questi anni.

Grazie agli amici che dentro e fuori l'università mi hanno supportato (o sopportato, decidete voi) e che cito brevemente in ordine casuale:

Grazie Chicchi, per l'amicizia, l'affetto ed il supporto oltre che per la stima nei miei confronti.

Grazie Cris, per l'amicizia quasi ventennale, per l'affetto e per tutto il resto.

Grazie Merina, per l'amicizia e per la tua diplomazia spontanea, presente negli anni.

Grazie Marcy, per la lunga amicizia e supporto da parecchi anni.

Inoltre, vi ringrazio per l'attesa! Gli amici appena citati sanno che ci sarebbero numerosi ed ulteriori motivi per ringraziarli, tali da poterci scrivere una tesi a parte!

Grazie Cli, per la tua amicizia, la tua diplomazia sincera e per il prezioso supporto nella parte finale di carriera universitaria.

Spero di essere riuscito e di riuscire a dare a tutti voi almeno una parte di quello che ho ricevuto! Grazie ancora di cuore e per qualsiasi cosa per voi ci sono.

Un ringraziamento particolare e doveroso va al Prof.Andrea Omicini, per tutto l'aiuto, il supporto, la disponibilità e la simpatia nel seguirmi nello svolgimento di questa tesi, ed anche per non avermi fatto iniziare dalla fisica nucleare.

Grazie anche alle tante persone che non ho nominato e con cui mi sono trovato bene in quel di Cesena e non solo.

Bibliografia

[1] C. Richardson, "Building monolithic applications." https://www.nginx.com/blog/introduction-to-microservices/, 2015.

[2] S. J. Fowler, Microservices in Production. 2017.

[3] S. Fulton, "What led amazon to its own microservices architecture." http://thenewstack.io/led-amazon-microservices-architecture/, 2015.

[4] C. Richardson, "Introduction to microservices." https://www.nginx.com/blog/introduction-to-microservices/, 2015.

[5] "Soa-rm - 3.1 service." http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf, 2015.

[6] E. Wolff, Microservices - Flexible Software Architecture. 2016.

[7] M. Flower, "Microservices." https://martinfowler.com/articles/microservices.html, 2014.

[8] M. Flower, "Microservices trade-offs." https://martinfowler.com/articles/microservice-trade-offs.html, 2014.

[9] C. Richardson, "Pattern: Decompose by subdomain." http://microservices.io, 2016.

[10] M. Conway, "How do committees invent?." http://www.melconway.com/Home/pdf/committees.pdf, 1968.

[11] "Rabbitmq." https://www.rabbitmq.com/.

[12] "Amqp." https://www.amqp.org/.

[13] "Zeromq." https://www.zeromq.org/.

[14] "Activemq." http://activemq.apache.org/.

[15] "Hornetq." http://hornetq.jboss.org/.

[16] "Java messaging service." https://jcp.org/en/jsr/detail?id=343.

[17] "Rfc 5023 - feed." http://tools.ietf.org/html/rfc5023.

[18] "Rfc 4287 - atom." http://tools.ietf.org/html/rfc4287.

[19] "Twenty-one experts define cloud computing." http://virtualization.sys-con.com/node/612375.

[20] V. M. de la Cruz, "Something about clouds." http://vmartinezdelacruz.com/something-about-clouds/.

[21] "Something about clouds." https://linuxcontainers.org/it/.

[22] http://patg.net/containers,virtualization,docker/2014/06/05/docker-intro/.

[23] "Docker." https://www.docker.com/.

[24] "Docker documentation." https://docs.docker.com/.

[25] "Docker engine api." https://docs.docker.com/engine/api/v1.25/.

[26] "Portainer." http://portainer.io/.

[27] "Rancher." http://rancher.com/rancher/.

[28] https://www.silversands.co.uk/events/test-development-inc-devops-southampton-st-marys-stadium/.

[29] https://12factor.net/it/.

[30] "Jolie." http://www.jolie-lang.org/.

[31] "La rivoluzione dei microservizi." http://www.slideshare.net/italianaSoftware/la-rivoluzione-dei-microservizi-70399420.

[32] "Sodep: Simple operation data exchange protocol." http://jolie.sourceforge.net/contents/sodep.html.

[33] "Openweathermap." https://openweathermap.org/.

[34] "Php." http://www.php.net/.

[35] "nginx." http://www.nginx.org/.

[36] "Leonardo - the jolie web server." https://github.com/jolie/leonardo.

[37] "Microservizi, scenari del prossimo e del lontano futuro." www.slideshare.net/italianaSoftware/micorservizi-scenari-del-prossimo-e-del-lontano-futuro.

[38] "Aiocj." http://www.cs.unibo.it/projects/jolie/aiocj.html.

[39] "Chor." http://www.chor-lang.org/.

[40] "Industria 4.0 - come verrà rivoluzionata l'industria italiana." http://www.slideshare.net/italianaSoftware/industria-40-come-verr-rivoluzionata-lindustria-italiana.

[41] "Meeting on microservices 2016." http://italianasoftware.blogspot.it/2016/12/leseperienza-meeting-on-microservices.html.