Politecnico di Torino

Laurea Magistrale in Ingegneria Informatica
A.a. 2021/2022
Sessione di Laurea ottobre 2022

Analisi e miglioramento dell'architettura di un'applicazione cloud native esistente sfruttando microservizi e service mesh

Politecnico di Torino

Relatore:

Prof. Maurizio MORISIO

Candidato:

Francesco CIARLA

Sommario

Introduzione

Premessa

Quello che distingue un'architettura monolitica tradizionale da una basata su servizi è la suddivisione dell'applicazione nelle sue funzioni base. I microservizi sono quindi un approccio architetturale alla realizzazione di un'applicazione.

I vantaggi portati da un'architettura a microservizi sono flessibilità, scalabilità e riutilizzo di componenti, tuttavia un'architettura di questo tipo ha delle difficoltà intrinseche, come ad esempio la gestione della comunicazione tra i diversi servizi, che in alcune applicazioni possono arrivare ad essere centinaia.

Un service mesh è un livello di infrastruttura che gestisce la comunicazione tra servizi, rendendo possibile la connessione, la messa in sicurezza e il monitoring.

Progetto di tesi

Scopo di questo lavoro è quello di analizzare il sistema esistente ai fini di migliorarne l'architettura e l'efficienza. Il sistema è stato progettato pensando dal principio a tecnologie così dette cloud native come i microservizi e ad un approccio allo sviluppo DevOps collaborativo, tuttavia lo sfruttamento di tali tecnologie è soltanto parziale. È stato quindi richiesto uno studio dell'applicazione al fine di progettare un'adeguata ristrutturazione a partire dai componenti che ne beneficerebbero maggiormente. Ci si è concentrati quindi sullo sviluppo del microservizio ritenuto di maggiore importanza, quello cioè relativo all'autenticazione degli utenti.

L'azienda ed il prodotto

Il lavoro di tesi è stato svolto presso HeadApp, un'azienda informatica, con sede a Torino, la cui mission è l'utilizzo della realtà virtuale e aumentata per supporto a tecnici ed operatori sul campo. L'applicazione pensata e sviluppata per tale obiettivo è Eye4Task, una piattaforma collaborativa che connette tecnici ed esperti a distanza per accelerare i flussi di lavoro e i processi decisionali. Tale piattaforma è resa disponibile alle aziende in white label, on Cloud oppure on-promise, ed è installabile sia su smartphone che su smart glasses. La forza del prodotto è proprio nell'utilizzo con smart glasses, usando questa soluzione infatti l'operatore può eseguire a mani libere le attività utilizzando comandi vocali per richiedere e ricevere supporto.

Capitolo 1 – Cloud computing

1.1 Virtualization e lightweight virtualization

Molto usata oggi giorno nell' implementazione di server e datacenter è la virtualizzazione, tale tecnologia è stata introdotta negli anni Sessanta da IBM al fine di permettere la condivisione di risorse su hardware costosi. Tra gli anni Settanta e Ottante ha perso importanza per via della diffusione dei "personal computer", salvo poi riacquisire rilevanza negli anni Novanta, quando si è incominciato ad utilizzare tale tecnologia proprio in ambito cloud, principalmente per ragioni di carattere economico. La crescente richiesta di specifiche configurazioni hardware per applicazioni diverse ha infatti trovato una soluzione proprio nell' utilizzo della virtualizzazione. Le aziende hanno iniziato a comprare in grande quantità componenti hardware comuni, di facile reperibilità e poco costosi, definiti "Common Off The Shelf" (COTS), sopra i quali è possibile implementare macchine virtuali che simulino hardware con le caratteristiche desiderate.

I principali vantaggi portati dall'utilizzo della virtualizzazione all'interno dei moderni server sono:

- **Isolamento:** Applicazioni critiche possono essere eseguite in sistemi operativi differenti e isolati tra loro, servizi diversi possono essere eseguiti sullo stesso host ma con un maggior grado di isolamento rispetto al passato, applicazioni o servizi malevoli non possono compromettere altri servizi in macchine virtuali differenti.

- **Consolidamento:** Sistemi operativi completamente differenti possono essere eseguiti contemporaneamente sullo stesso hardware, limitando in questo modo l'utilizzo di risorse, sfruttando al meglio CPU multi-core ed ottimizzando il consumo di energia.

- **Flessibilità e agilità:** Un controllo completo sull'esecuzioni delle macchine virtuali è possibile, si può mettere in pausa e far ripartire l'esecuzione del sistema operativo ed è possibile migrare la macchina virtuale da un host ad un altro per assegnargli più risorse o per alleggerire il server stesso. È inoltre possibile duplicare una macchina virtuale in esecuzione in modo da gestire ed indirizzare un picco di carico.

Possiamo affermare che il vantaggio principale di tale tecnologia sia nella possibilità di creare un sistema autonomo che evolve autonomamente e si adatta al cambiamento.

La virtualizzazione tuttavia porta anche alcuni svantaggi, come ad esempio un overhead nell'avvio di alcune applicazioni, dovuto al fatto che ognuna di esse ha il proprio sistema operativo, che consuma ulteriori risorse in termini di memoria e CPU; tale overhead è considerato accettabile per la maggior parte delle applicazioni e dei sistemi operativi.

Il software utilizzato per la gestione del processo di virtualizzazione è l'hypervisor, chiamato anche Virtual Machine Monitor (VMM). Nello specifico esso si occupa della virtualizzazione delle risorse hardware come ad esempio memoria e CPU, assegna ad ogni macchina virtuale un set di risorse, garantisce che ognuna di esse non possa accedere a risorse non proprie e funge da arbitro nell'accesso a risorse condivise. Un hypervisor consiste in una versione ridotta di un sistema operativo, spesso Linux, ed offre un set di driver nativi per gestire l'hardware. Può essere soggetto ad attacchi, ma essendo un sistema molto semplice risulta essere molto più facilmente difendibile di un normale sistema operativo.

Nell'articolo *Formal Requirements for Virtualizable Third Generation Architectures* [1] gli autori Popek e Goldberg forniscono un insieme di condizioni che rendono un'architettura capace di gestire la virtualizzazione. In tale articolo [1] una macchina virtuale è definita come un duplicato efficiente e isolato di una macchina reale. Lo stesso articolo [1] definisce inoltre un VMM attraverso tre caratteristiche principali:

- Ogni programma eseguito sotto il controllo del VMM deve produrre lo stesso risultato dello stesso programma eseguito in un sistema reale.

- Un sottoinsieme statisticamente dominante di istruzioni del processore virtuale deve essere eseguito dal processore reale senza interventi di tipo software da parte del VMM.

- Il VMM deve avere controllo completo sulle risorse del sistema reale, ciò significa che non è possibile che un programma possa accedere a risorse esterne allo spazio allocato per lui, e anche che in certe circostanze il VMM possa riprendere il controllo di risorse già allocate in precedenza.

Seguendo il concetto di "Full Virtualization" il sistema operativo della macchina virtuale non deve essere "consapevole" di essere virtualizzato.

Gli stessi Popek e Goldberg nel 1974 definirono un primo paradigma per la virtualizzazione chiamato Trap&Emulate (T&E).

Prima di vedere nel dettaglio tale paradigma diamo alcune definizioni. Definiamo un'istruzione privilegiata come un'istruzione che non può essere eseguita ad un livello di protezione inferiore a quello massimo e un'istruzione sensibile come un'istruzione che fornisce informazioni sullo stato fisico del processore; affinché la virtualizzazione sia possibile è necessario che tutte le istruzioni sensibili siano privilegiate.

Seguendo il paradigma T&E, il sistema operativo della macchina virtuale (guest OS) è eseguito ad un livello non privilegiato; quando viene lanciata un'istruzione privilegiata una trap viene lanciata dal processore ed intercetta dal VMM: questo emula l'istruzione ed in seguito restituisce il controllo al guest OS.

L'hypervisor ha anche il compito di intercettare le system call da parte delle applicazioni utente e dare il controllo al guest OS: per via di questo procedimento una singola syscall viene eseguita in un tempo 10 volte superiore a quello necessario al sistema operativo della macchina fisica (host OS).

Il T&E ha un problema: esso non può essere applicato a tutti i tipi di architetture, in particolare non può essere applicato a quelle architetture che non soddisfano i requisiti forniti da Popek e Goldberg, come ad esempio l'Intel x86. Tale processore infatti presenta istruzioni

(come POPA o POPF) che non chiamano una trap se eseguite ad un livello non privilegiato, e non possono quindi essere intercettate ed emulate dal VMM.

Architetture come l'x86 risultano quindi non-virtualizzabili, quindi per risolvere tale problema sono state pensate diverse soluzioni che riportiamo di seguito.

- Analizzare tutte le istruzioni e intercettare dinamicamente quelle sensibili, soluzione implementata tramite "Interpretetion" e "Dynamic Binary Translation" (DBT): Il primo metodo non ha avuto grande successo, in quanto introduce un overhead troppo elevato, il DBT invece ha avuto un successo enorme ed è stato il più usato fino all'avvento della para-virtualizzazione esso introduce comunque un overhead, ma non prevede la modifica del sistema operativo da virtualizzare.

- Modificare il sistema operativo della macchina virtuale: soluzione usata dalla para-virtualizzazione. Le performance sono vicine a quelle ottenute da sistemi fisici, ma richiede una modifica del sistema operativo da virtualizzare.

- Trasformare le istruzioni sensibili in privilegiate: questa soluzione richiede un supporto hardware ("Hardware-assisted virtualization" HVM). Un esempio sono il VT-x di Intel e l'SVM di AMD.

Dynamic Binary Translation (DBT)

Dynamic Binary Translation è stata la prima tecnica proposta da VMware nel 1998. Tale tecnica non necessita di alcuna modifica del sistema operativo né di alcun supporto hardware. L'idea di base prevede che il VMM traduca dinamicamente le istruzioni non-virtualizzabili dell'x86 in istruzioni virtualizzabili a run-time. Ciò che viene tradotto è il codice binario e la traduzione avviene mentre il codice è in esecuzione. Come già detto questa tecnica introduce un overhead significativo rispetto ad altre; in alcuni casi è comunque possibile ridurre il tempo di traduzione usando del caching per determinate istruzioni.

Para-virtualizzazione

Usando questa tecnica il guest OS viene modificato per poter essere virtualizzato; possiamo dire che esso è "consapevole" di venire eseguito su di una macchina virtuale (si abbandona in questo caso il concetto di "full virtualization"). Nello specifico tutte le system calls vengono sostituite da hypervisor calls (hypercall), così come le istruzioni non-virtualizzabili, e vengono introdotti meccanismi all'interno del kernel per facilitare la comunicazione con l'hypervisor. L'implementazione di tale tecnica è più semplice e veloce rispetto alla DBT, tuttavia non è possibile utilizzarla per tutti i sistemi operativi.

Hardware-assisted virtualization (HVM)

Questa tecnica si propone di risolvere i problemi principali delle precedenti: non tutti i sistemi operativi possono essere modificati per implementare la virtualizzazione e la DTB risulta avvolte troppo lenta. HVM propone di usare efficientemente Trap&Emulate utilizzando un supporto hardware. L'idea principale è quella di evitare le istruzioni sensibili, o rendendole privilegiate, o lasciando che il VMM sia in grado di capire dinamicamente quali istruzioni debbano lanciare una trap.

In ambito server il sistema operativo più usato è sicuramente Linux, risulta essere quindi molto rilevante il modo in cui viene implementata la virtualizzazione su tale sistema. Una componente molto importante in questo contesto è QEMU: nato come semplice emulatore è divenuto una componente importante per il meccanismo di virtualizzazione in ambiente Linux, spesso utilizzato insieme a KVM.

Il "Kernal-based Virtual Machine" (KVM) è un modulo kernel di Linux (kvm.ko) che fornisce l'infrastruttura principale per la virtualizzazione trasformando il Linux kernel stesso in un hypervisor. KVM fa leva sull'utilizzo della HVM, il codice della macchina virtuale è eseguito direttamente sulla macchina fisica ed è necessario che Intel VT-x o AMD-V siano abilitati sulla CPU. L'hypervisor associa uno o più CPU ad una macchina virtuale, ogni macchina virtuale crea un singolo processo utente ed ogni CPU virtuale (vCPU) è un thread del sistema operativo della macchina host.

Un altro ingrediente fondamentale per la virtualizzazione in Linux è Libvirt, una libreria contenente diverse API, e che permette operazioni sulle macchine virtuali come creazione, modifica, monitoraggio, controllo e migrazione. Su Libvirt si basa anche Virtual Machine Manager, applicazione che offre anche un'interfaccia grafica per la gestione delle machine virtuali, oltre a supporto per hypervisor sia locali che remoti.

La virtualizzazione basata sull'utilizzo di hypervisor è stata usata per anni come lo standard per le virtualizzazioni su server; negli ultimi anni tuttavia la virtualizzazione basata sull'uso di container ha assunto sempre maggior rilevanza soprattutto grazie alle sue caratteristiche lightweight. Mentre il tipo di virtualizzazione descritta fin ora opera a livello hardware, la virtualizzazione basata sull'uso di container opera a livello software.

Un container non è in grado di eseguire un sistema operativo Windows in ambiente Linux come potrebbe avvenire in una macchina virtuale, ma permette di implementare l'isolamento di processi a livello del sistema operativo della macchina host, evitando l'overhead dovuto alla virtualizzazione dell'hardware e producendo immagini che occupano meno spazio sul disco rispetto ad una macchina virtuale.

Un altro vantaggio dei container rispetto agli hypervisor può essere trovato nel consumo di energia per quanto riguarda operazioni di rete (operazioni di grande rilevanza in ambito server). Ciò è ben descritto in *Power Consumption of Virtualization Technologies: an Empirical Investigation* [2], articolo di Roberto Morabito del 2015. Nello studio [2] è descritto un esperimento per calcolare i diversi consumi di energia tramite vari benchmark di tecnologie basate su hypervisor (KVM e Xen) e tecnologie basate su container (Docker e LXC). Dalle conclusioni dell'autore [2] si può dedurre che il consumo di energia per hypervisor e container è simile per quanto riguarda carichi sulla CPU o sulla memoria; delle differenze ci sono invece

per quanto riguarda operazioni di rete, per le quali nella maggior parte dei casi il consumo di energia dei container è minore di quello degli hypervisor.

Uno dei software più utilizzati oggi per la creazione ed il mantenimento di container è Docker.

Lo scopo principale di Docker è quello di semplificare la distribuzione e l'esecuzione delle applicazioni creando dei contenitori leggeri, facilmente trasferibili e contenenti tutto il necessario affiche l'applicazione possa essere eseguita ovunque; in questo modo anche il lavoro di sviluppo dell'applicazione è semplificato, se essa funziona bene localmente allora funzionerà allo stesso modo anche sul server (o in ogni altro luogo), essendo eseguita in ogni caso all'interno del container Docker.

In tale contesto è importante definire i concetti principali per l'utilizzo di Docker.

Docker Image rappresenta un modello immutabile per la creazione di un container e può essere inserito o prelevato usando un Docker Registry.

Docker Container rappresenta l'istanza di un'immagine, può essere avviata, stoppata o fatta ripartire mantenendo internamente i cambiamenti apportati al filesystem.

Docker Registry invece, è un repository software all'interno del quale viene tenuta traccia di varie immagini e sul quale si può lavorare tramite operazioni come push, per inserire un'immagine o pull, per prenderne una in esso contenuta. Uno degli esempi più rilevanti di Docker Registry online è DockerHub.

Un container Docker deve copiare al suo interno l'intero filesystem necessario all'esecuzione della specifica applicazione, così da garantire isolamento e portabilità. In questo modo infatti per l'esecuzione dell'applicazione non sarà necessario accedere a file esterni al container stesso (presenti sulla macchina host), perché il container avrà internamente tutto il necessario per l'esecuzione.

Docker utilizza un filesystem a livelli, infatti un'immagine complessa può risultare dalla composizione di più layer. È possibile creare un container partendo direttamente dai suoi elementi costitutivi, e ciò può essere fatto grazie ad un Dockerfile, una sorta di "ricetta" per la creazione del container stesso.

Oltre vantaggi ai già citati, Docker permette di semplificare il processo di continous delivery (distribuire software molto spesso e con meno errori) e di migliorare la sicurezza del sistema, in quanto ogni parte di esso può essere isolata e può esservi maggior controllo su ogni componente.

Per la gestione di applicazioni che necessitano di più container, Docker mette a disposizione un tool chiamato Docker Compose che permette di creare e gestire tutti i vari servizi usando un solo comando con il supporto di un file nel formato YAML.

Vedremo nel prossimo paragrafo come il compito svolto da Docker Compose possa essere eseguito anche da orchestrator come ad esempio Kubernetes. Nonostante questo, il tool offerto da Docker resta rilevante nella gestione di applicazioni multi-container, soprattutto per via della sua maggiore leggerezza.

Vedremo ora come vengono gestiti i diversi componenti (che siano container o macchine virtuali) all'interno di datacenter tramite l'utilizzo di orchestrator.

1.2 Sistemi operativi per il Cloud

Macchine virtuali e container hanno reso la gestione delle risorse molto più flessibile rispetto al passato, con il tempo si è passato da singoli server ad agglomerati più grandi detti data centers. Con l'avvento dei data centers pubblici e con la nascita di Amazon AWS nel 2006, si è iniziato a parlare di Cloud Computing. Si usa il termine "Cloud" poiché dato che la computazione avviene in un luogo che l'utente finale non conosce, è come se questa venisse fatta "fra le nuvole".

Il passaggio dalla vecchia infrastruttura basata su singoli server al cloud può essere spiegato tramite la "Pet vs Cattle analogy": prima i server venivano trattati come degli animali domestici, erano unici, non rimpiazzabili e ci si prendeva molta cura di loro. Ora invece si guarda al cloud come ad un gregge: ogni server non è più unico e insostituibile, ma rappresenta una risorsa rimpiazzabile.

Secondo la definizione data dal NIST (Nationale Institute of Standards and Technology) «il cloud computing è un modello per abilitare, tramite la rete, l'accesso diffuso, agevole e a richiesta, ad un insieme condiviso e configurabile di risorse di elaborazione (ad esempio reti, server, memoria, applicazioni e servizi) che possono essere acquisite e rilasciate rapidamente e con minimo sforzo di gestione o di interazione con il fornitore di servizi». Il NIST definisce anche cinque caratteristiche principali del cloud computing: ampio accesso alla rete, misurazione dei servizi utilizzati (per il pagamento), grande elasticità, condivisione delle risorse e self-service su richiesta.

Si possono distinguere quattro principali modelli di fruizione del cloud computing:

- HaaS (Hardware as a Service): con questo modello i provider mettono a disposizione l'hardware, lo spazio di archiviazione, le strutture, la connessione ad internet e tutto il necessario per la gestione della struttura fisica. L'utente finale ha quindi totale controllo sull'implementazione ed è l'unica responsabilità dell'infrastruttura software. Questo modello è il meno utilizzato da parte delle aziende.

- IaaS (Infrastructure as a Service): viene offerta un'infrastruttura completa (rete, computing e storage); gli utenti possono creare macchine, reti e dischi virtuali ed installare sistemi operativi e applicazioni a loro piacimento (es. Amazon EC2, Rackspace Hosting).

- PaaS (Platform as a Service): vengono offerti diversi servizi di base, come database o compilatori per vari linguaggi. L'utente finale può sfruttare tali servizi per lo sviluppo di nuovi software, ma dettagli su sistemi operativi, connessioni di rete e scaling sono nascosti (es. Google App Engine, Windows Azure).

- SaaS (Software as a Service): con questo modello all'utente finale vengono offerte applicazioni complete e funzionati in modo che egli dovrà semplicemente accedervi ed usarne le varie funzionalità (es. Google Docs, Microsoft Office.com).

Mentre fino allo scorso decennio lo standard *de facto* per i server era il sistema operativo Linux, con il passaggio al cloud è stato necessario trovare dei sistemi operativi in grado di gestire

il posizionamento delle varie componenti all'interno di un datacenter, l'avvio, la messa in pausa e lo spostamento delle macchine virtuali ed i container, la migrazione da un server ad un altro e la suddivisone dello storage e della rete. A tale scopo sono stati sviluppati i CMS (Cloud Management System), dei framework software che permettono all'amministratore di un datacenter di controllare l'intera infrastruttura, sia dal punto di vista hardware che software. Ogni cloud provider ha il proprio CMS, ma più diffusi e più importanti al momento sono OpenStack e Kubernetes, entrambi open source, il primo più orientato al coordinamento di macchine virtuali, il secondo invece al coordinamento di container.

OpenStack

OpenStack è stato sviluppato nel 2010 grazie alla collaborazione fra la NASA e RackSpace. La sua architettura è basata su vari moduli (come è possibile vedere in figura 1), ognuno dei quali fornisce funzionalità diverse e può essere facilmente aggiunto o rimosso. Esso permette di gestire la scalabilità del sistema, mentre l'isolamento dei vari processi permette di evitare guasti a cascata.

Per gestire la comunicazione fra i vari moduli viene utilizzata una message queue (RabbitMQ). La coda di messaggi viene preferita ad una connessione diretta, ad esempio di tipo TCP, per diverse ragioni: la destinazione di un dato messaggio potrebbe cambiare posizione nel tempo, una macchina virtuale potrebbe essere stata spostata da un server ad un altro, il messaggio che si vuole inviare potrebbe essere diretto a più di un'unica destinazione, e potrebbe non essere conosciuto a priori il numero di destinatari interessati al messaggio stesso; l'utilizzo di un bus condiviso per i messaggi aiuta inoltre il monitoraggio del sistema.

I moduli principali di OpenStack sono:

- Nova: può essere definito il componente principale ed è quello che si occupa della computazione nonché la gestione dell'intero pool di macchine virtuali.

- Neutron: fornisce la connettività di rete e fa sì che ogni componente possa comunicare con gli altri in modo efficiente e veloce.

- Glance: fornisce un repository per il mantenimento e la gestione di immagini. Quando c'è bisogno di creare nuove macchine virtuali, tali immagini sono usate come template per la creazione.

- Horizon: è la dashboard di OpenStack, che fornisce un'interfaccia grafica e che dunque è l'unico componente che l'utente "vede" e con il quale interagisce effettivamente.

- Keystone: fornisce autenticazione e autorizzazione per tutti i servizi offerti da OpenStack

- Swift: fornisce un metodo di archiviazione per file e oggetti.

- Cinder: fornisce archiviazione persistente per le macchine virtuali.

- Ceilometer: fornisce l'infrastruttura per la misurazione di dati relativi a performance, ai dati scambiati e alle connessioni; tali dati vengono usati per prezzare l'utilizzo del cloud da parte degli utenti.

- Heat: è il componente che fa da orchestrator, dando la possibilità di fare il setup di un servizio tramite un semplice file XML.

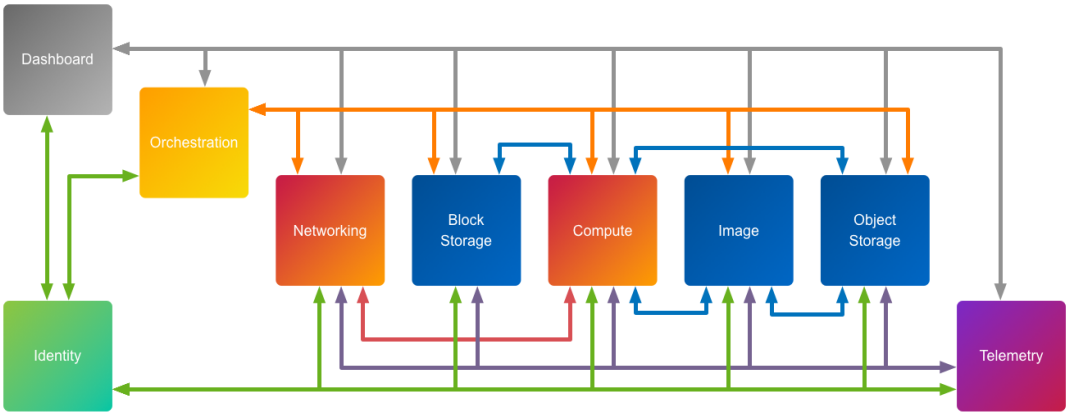


Figura 1

Kubernetes

Con il tempo OpenStack sta perdendo attrattiva a favore di Kubernetes, che è candidato per diventare in un prossimo futuro lo standard per i sistemi operativi sul cloud. Kubernetes, spesso abbreviato con la sigla k8s, introduce un nuovo e diverso livello di astrazione, differente da quello basato su servizi a livello infrastrutturale di OpenStack e altri toolkit precedenti.

L'approccio implementato da k8s è totalmente dichiarativo, seguendo la filosofia dell'Infrastructure as a Code: lo sviluppatore deve semplicemente descrivere la logica da applicare, senza descrivere il control flow da seguire. Il mantenimento dello stato desiderato sul sistema è completamente demandato ad un Control-loop che controlla costantemente che le specifiche desiderate siano rispettate. Questo tipo di monitoraggio viene fatto da specifiche risorse di Kubernetes chiamate controller. Tutte le risorse in k8s hanno la stessa struttura e vengono descritte all'interno di file YAML tramite diversi campi come "apiVersion", "kind" (che descrive il tipo di risorsa descritta nel file), "spec" e "status" (che rappresentano rispettivamente lo stato desiderato dell'utente e lo stato corrente della risorsa), e "labels" e "annotation" (che permettono di aggiungere informazioni supplementari).

Definiamo le principali risorse disponibili in Kubernetes:

- Pod: rappresenta l'elemento minimo di un cluster Kubernetes ed è l'unità fondamentale per lo scheduling. Ogni pod permette al suo interno l'esecuzione di uno o più container ed è obbligatoriamente eseguito su un singolo nodo. La maggior parte dei pod ha al suo

interno un solo container; quando ne possiede più di uno, quelli aggiuntivi sono detti "sidecar container", ed estendono le funzionalità del container principale. Tutti i container all'interno dello stesso pod condividono lo stesso stack TCP/IP e quindi possono comunicare fra di loro senza la necessità di servizi aggiuntivi.

- Namespace: questa risorsa permette la creazione di cluster logici. Laddove normalmente non esistono firewall fra pod differenti, usando un namespace possiamo invece limitare la comunicazione fra diversi gruppi di pod.

- ReplicaSet: garantisce la presenza di un numero specifico di repliche fornito dall'utente. Quando una replica non è più attiva viene distrutta e al suo posto ne viene creata un'altra. Questa risorsa può essere vista come l'implementazione del Cattle pattern: non si è più interessati al singolo elemento del gregge poiché ognuno di essi è rimpiazzabile.

- Deployment: di solito gli utenti non interagiscono direttamente con una replica set ma con un elemento che si pone ad un livello più alto; questa risorsa è proprio il Deployment. Esso è responsabile per il deployment di una determinata versione del software ed offre un modo per applicare le modifiche sui pod e sui replicaSet.

- DeamonSet: rappresenta un pod speciale. Ogni volta che un nuovo nodo viene aggiunto al cluster Kubernetes, questo pod viene schedulato sul nuovo nodo. In questo modo ogni nodo ha sicuramente un pod in esecuzione al suo interno.

- Horizontal Pod Autoscaler: risorsa che permette l'autoscalig delle repliche. Al suo interno possono essere specificati numero minimo e massimo di repliche desiderate, a quale risorsa va applicato l'autoscaling e l'obiettivo che si desidera raggiungere.

- Service: rappresentano un meccanismo per esporre un'applicazione in esecuzione ad un set di pod. Tale risorsa permette di disaccoppiare il fatto che un servizio sia visibile dal fatto che esso sia in esecuzione. Esistono quattro tipi di service: ClusterIP, il service basico, esposto solo all'interno dello stesso cluster; NodePort, il servizio esposto sull'IP di ogni nodo ad una porta statica; LoadBalancer, il servizio esposto esternamente usando il load balancer del provider del cloud; ExternalName, che permette di accedere al servizio tramite l'externalName specificato (es. external.example.it). Particolari tipologie di service sono gli Handless Service, i quali vengono usati quando non c'è bisogno di load balancing o di un front-end per il servizio, e che vengono creati specificando "None" alla voce "clusterIP".

Un cluster Kubernetes è formato da diversi nodi, uno dei quali viene identificato come il master node (chiamato Control Plane), mentre gli altri invece sono chiamati Workers. Tutte le funzionalità di k8s sono esposte tramite API REST, le quali possono essere invocate inviando richieste all' APIServer.

Come possiamo vedere dalla figura 2 Control Plane e Workers si differenziano fra di loro oltre che per i ruoli ricoperti anche per le componenti da cui sono costituiti. All'interno del Control Plane troviamo:

- Kube-apiserver: esso rappresenta il front-end del Control Plane ed espone tutte le API che possono essere invocate sia dal kubectl CLI che dalla GUI opzionale di Kubernetes.

- Kube-scheduler: è il componente incaricato di decidere a quale nodo deve essere assegnato un pod appena creato tenendo conto di vari parametri, tra cui le risorse richieste dai vari pods e quelle disponibili sui vari nodi, i vincoli hardware e software, il posizionamento dei dati e molti altri.

- Etcd: database chiave-valore distribuito che fornisce un modo per archiviare i dati all'interno del cluster; è veloce, sicuro, semplice e affidabile.

- Kube-control-manager: è il controller contente la logica del control loop. Esso osserva lo stato del cluster utilizzando il kube-apiserver e quando lo stato si discosta dalle specifiche fornite dall'utente, cerca di modificarlo per portarlo a quello desiderato.

- Cloud-control-manager: componente che serve a garantire una buona integrazione con il cloud provider; esso infatti esegue tutti i controller che interagiscono con il cloud provider sottostante. Alcuni controller di questo tipo sono ad esempio quelli che configurano le regole per i load-balancing o quelli che notificano quando un nodo diventa attivo o viene disattivato.

Per quanto riguarda invece i Worker nodes, gli elementi di cui sono costituiti oltre ai container ed i possibili "sidecar container" sono:

- Kubelet: tiene sotto controllo lo stato dei singoli pod in esecuzione sul nodo. Esso fornisce al kube-apiserver informazioni riguardo il ciclo di vita di ogni pod e nel nodo stesso.

- Kube-proxy: componente che serve a gestire le network rules su ogni nodo e ad implementare il forwarding ed il load-balancing sul cluster.

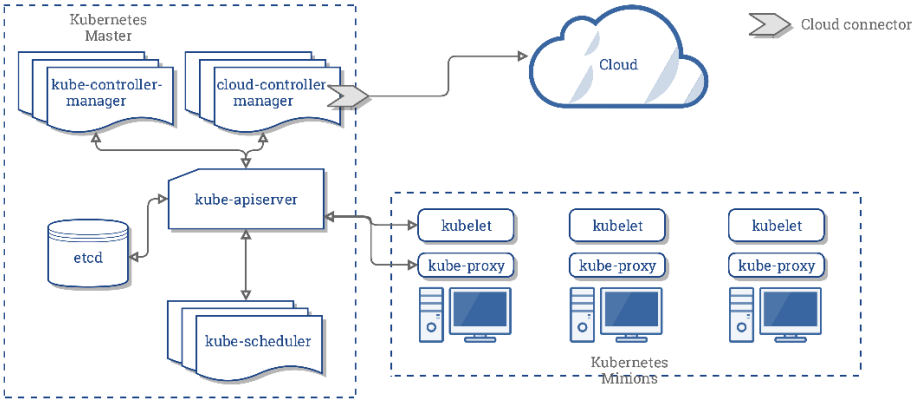


Figura 2

Per quanto riguarda il networking all'interno dei datacenter ci sono due possibili modelli per il rounting: Overlay Model e Direct Routing. Il primo è adottato da OpenStack e non prevede l'interazione con la rete fisica del provider sottostante. In questo modello i pacchetti vengono scambiati usando il tunnelling (può essere fatto sia a livello di rete 2, preferita da OpenStack, che a livello di rete 3); in tal modo la rete del provider non ha bisogno di conoscere i singoli IP delle macchine virtuali all'interno dei server fisici. Kubernets al contrario predilige il Direct Routing: qui il tunneling non viene usato, quindi la rete fisica ha bisogno di conoscere gli IP delle macchine all'interno dei server fisici (questo modello è possibile solo a livello di rete 3 dato che gli IP devono essere conosciuti dalla rete fisica). A livello implementativo, come abbiamo già visto, i due CMS usano due approcci differenti: OpenStack delega il compito al modulo preposto (Neutro), mentre k8s usa un approccio funzionale, per il quale il comportamento specificato dall'utente viene supportato da un provider di rete esterno.

1.3 Vantaggi e svantaggi del cloud computing

Dopo aver visto quali sono le tecnologie alla base del cloud computing, quali sono i diversi modelli esistenti e in che modo le risorse vengono gestite, possiamo riassumere i vantaggi portati da questa tecnologia e andare a vedere i rischi ad essa correlati [3].

Abbiamo visto che le categorie principali in cui possiamo dividere i servizi offerti in cloud sono Infrastructure as a Service (IaaS), Platform as a Service (PaaS) e Software as a Service (SaaS). Tutte e tre queste categorie permettono alle aziende di non dover investire in hardware o in altri tipi di infrastrutture: basta comprare uno di questi servizi da cloud provider. Si ha quindi un risparmio in termini di costi, in quanto costruire una infrastruttura da zero costa sicuramente di più rispetto a pagare per un servizio offerto da un cloud provider, ed anche i tempi amministrativi per il reperimento dei componenti hardware e software sono ridotti. Il modello

di pagamento è generalmente “pay as you go”, quindi si paga solo per le funzionalità che si usano; in aggiunta i costi di mantenimento sono azzerati del tutto.

Vi è sicuramente un miglioramento in termini di accessibilità: le risorse sono accessibili da tutto il mondo, in ogni momento e da qualunque dispositivo, l’affidabilità è maggiore, infatti i servizi sono costruiti su modelli ridondanti ed il backup ed il recovery dei dati sono favoriti dalla struttura delle infrastrutture. Un altro grande vantaggio è dato dagli aggiornamenti automatici garantiti dai servizi cloud: un’azienda non dovrà in questo modo preoccuparsi di tenere aggiornati i sistemi manualmente.

Passiamo ora agli svantaggi che il cloud computing porta con sé, alcuni dei quali non sono altro che rovesci della medaglia dei vantaggi appena visti.

Per cominciare l’accesso per servizi in cloud richiede una buona connessione ad internet con un buon bandwidth, soprattutto nel caso in cui si stia lavorano con il download o l’upload di file. La necessità di una buona connessione ad internet potrebbe portare anche a casi di interruzione del servizio quando questa è rallentata o del tutto interrotta.

Abbiamo anche visto che le aziende non hanno più la responsabilità di doversi occupare dell’infrastruttura, con un evidente vantaggio in termini di costi e manutenzione; questo tuttavia implica anche un controllo limitato o quasi assente sulle infrastrutture stesse da parte dell’azienda che le utilizza, assieme a delle restrizioni sull’utilizzo dei servizi stessi offerti dai provider. Inoltre la dipendenza da alcune piattaforme o il così detto vendor lock-in rendono in alcuni casi molto difficile la migrazione da un provider ad un altro a causa di problemi di compatibilità, interoperabilità e supporto.

Un ulteriore punto essenziale da considerare per il passaggio al cloud è la sicurezza. Nel caso di cloud pubblico è compito del provider assicurarsi che i dati gestiti siano al sicuro; dal punto di vista degli utilizzatori è quindi essenziale scegliere un provider che sia conforme alle politiche di sicurezza dei dati. Per avere più sicurezza, naturalmente, spesso ci si trova a dover pagare di più e provider che offrono un livello maggiore di sicurezza sono di frequente più costosi. È tuttavia possibile adottare un approccio ibrido per ottenere maggiore sicurezza e cercare di minimizzare i costi; in questo caso tutti i dati generici sono mantenuti sul cloud pubblico, mentre i dati critici per il proprio business sono mantenuti all’interno di un cloud privato.

1.4 Cloud-Native Application

Con lo sviluppo del cloud computing è nata anche la necessità di sviluppare applicazioni che potessero sfruttare a pieno le potenzialità offerte da tale tecnologia. Questo tipo di applicazioni vengono definite Cloud-Native e sfruttano i vantaggi offerti dal cloud in termini di elasticità, scalabilità e automazione. Una Cloud-Native Application (CNA) può quindi essere definita come un’applicazione intenzionalmente disegnata per lavorare sul cloud.

Al di là questa definizione generica, possiamo prendere in considerazione la definizione data da Nane Kratzke e Peter-Christian Quint nell'articolo *Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study* [4], nel quale vengono prese in considerazione la maggior parte delle pubblicazioni fatte sull'argomento a partire dalla nascita del cloud computing nel 2006. In tale articolo [4] si afferma che un'applicazione cloud native dovrebbe essere IDEAL: [i]soleted state, [d]istributed, [e]lastic, in modo da poter scalare orizzontalmente, [a]utomated managed, cioè gestita in maniera automatica, e [l]oosely couple, cioè scarsamente connessa con altre applicazioni. Citando la definizione data dagli autori: «a cloud-native application (CNA) is a distributed, elastic and horizontal scalable system composed of (micro)services which isolates state in a minimum of stateful components. The application and each self-contained deployment unit of that application is designed according to cloud-focused design patterns and operated on a self-service elastic platform». Nell'articolo [4] vengono anche chiariti i vari termini utilizzati nella definizione:

- Elasticità: è la capacità di un sistema di adattarsi ai cambiamenti del carico di lavoro andando ad aggiungere o rimuovere risorse in maniera automatica, così da poter gestire le richieste nel modo migliore in qualunque momento.

- Scalabilità: si differenzia in scalabilità strutturale e di carico. La prima riguarda la capacità del sistema di aumentare le risorse di una data dimensione senza la necessità di modifiche strutturali. Con scalabilità di carico, invece, si intende la capacità di procedere con l'esecuzione senza variazioni anche nel caso di un aumento del traffico.

- Microservizi: piccoli servizi, ognuno dei quali esegue un suo task e che comunicano tra di loro utilizzando meccanismi lightweight (andremo del dettaglio dei microservizi più avanti).

- Standard container: come abbiamo già visto, lo scopo di un container è quello di incapsulare un software e tutte le sue dipendenze, in modo da poter essere portabile e contenere tutto il necessario senza dover dipendere da risorse esterne per l'esecuzione.

- Componenti stateful: sono usati per la creazione di diverse istanze in grado di sincronizzare il loro stato interno per fornire un comportamento univoco.

- Piattaforma elastica: middleware utilizzato per l'esecuzione di applicazioni custom, che offre meccanismi di comunicazione e data storage (K8s e OpenStack).

Secondo quanto riportato da M. Yousif nell'articolo *Cloud-Native Applications—The Journey Continues* [5], inoltre, con cloud native si intende anche qualunque tipo di approccio che renda un'applicazione capace di sfruttare i benefici del cloud computing. Nello specifico l'autore afferma [5]: «[…] I define a cloud native application as an application that has been built from the ground up as a distributed architecture that leverages every aspect of the cloud for better performance, reliability, scalability, and every other *ility you can think of. All other approaches, such as finding a way to make a legacy application benefit from cloud features, are varieties of

cloud native. For example, when transforming a legacy application into microservices that run in containers, it becomes sort-of cloud native as it may not include all the tight connections in its software architecture to exploit the cloud».

1.5 12-factor App rules

La 12-factor app è una metodologia di sviluppo per la costruzione di microservizi in ambito Software as a Service (SaaS). Tale metodologia è stata ideata dagli sviluppatori di Heroku nel 2011 ed ha come obiettivo quello di gestire i problemi più comuni dei SaaS in termini di scalabilità e mantenimento; lo scopo è in definitiva quello di fornire delle linee guida per la costruzione di applicazioni cloud-native [6].

Come suggerisce il nome stesso vengono definite dodici regole [6].

I. Codebase

L'applicazione deve avere un singolo codebase ed esso deve essere mantenuto sotto un sistema di version control come ad esempio Git, Mercurial o Subversion.

Un codebase è quindi un singolo repository o un insieme di repository sotto lo stesso root commit e vi è una corrispondenza uno ad uno tra esso e l'applicazione. Questo non vuol dire che non vi possano essere diverse istanze della stessa applicazione; in questo caso ogni istanza viene chiamata deploy (in figura 3 uno schema esemplificativo). Un singolo deploy può essere ad esempio il software in produzione o la copia mantenuta da un singolo sviluppatore.

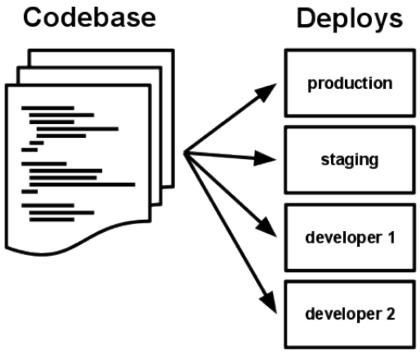


Figura 3

Se ci sono più codebase si parla di sistema distribuito e non più di applicazione; in questo caso ogni componente può essere definito applicazione ed ognuno di essi può aderire individualmente al 12-factor app metodology. Se al contrario più applicazioni condividono lo stesso codebase si ha una violazione del 12-factor.

II. Dipendenze

L'applicazione deve contenere al suo interno tutte le dipendenze necessarie alla sua esecuzione e non deve fare affidamento su nessuna risorsa esterna.

Le dipendenze dell'applicazione vengono dichiarate tramite un manifesto dedicato e viene utilizzato anche un tool per l'isolamento delle dipendenze a runtime in modo da assicurarsi che non ci siano dipendenze implicite che creino interferenze. Un beneficio importante portato da questa regola sta nel fatto che essa semplifica di molto la configurazione iniziale per i nuovi sviluppatori: ad essi basterà infatti fare un chackout del progetto ed installare solo le dipendenze necessarie guardando quelle presenti nel manifesto. Un'applicazione che segue la 12-factor inoltre non deve mai basarsi su qualunque tool di sistema, in quanto tale tool potrebbe non essere presente su tutti i sistemi sui quali l'applicazione girerà in futuro.

III. Configurazione

L'applicazione deve essere completamente agnostica riguardo all'ambiente sul quale viene eseguita.

Deve essere possibile spostare l'applicazione su ambienti diversi senza dover cambiare il codebase. Per fare ciò è importante gestire nel modo migliore la configurazione dell'applicazione, cioè tutto ciò che può variare da un deployment all'altro. La configurazione comprende: valori per connettersi ad un database o altri backing service, credenziali per servizi esterni, come ad esempio, Amazon S3, o valori definiti per i singoli deployment, come l'hostname. Salvare questi valori come variabili all'interno del codice andrebbe a violare la prima regola, in quanto si avrebbero codebase differenti per differenti deployment. Per questa ragione si preferisce utilizzare file di configurazione non coinvolti dal version control in formato yaml, un esempio potrebbe essere il file "config/database.yaml". Questo approccio ha tuttavia degli svantaggi: è infatti molto semplice includere in un repo un file di configurazione che non dovrebbe essere lì, ed inoltre essi sono dipendenti dai vari linguaggi/framework. In definitiva il modo migliore per gestire la configurazione è attraverso l'utilizzo di variabili d'ambiente, le quali sono molto semplici da cambiare da deployment in deployment senza dover toccare direttamente il codice.

IV. Backing Service

Bisogna spostare al di fuori dell'applicazione tutti quei servizi che non sono core component dell'applicazione stessa; tali servizi devono essere accessibili appunto "as a Service".

Con Backing Service si intende un qualunque servizio che viene consumato dall'applicazione tramite la rete durante la sua esecuzione. Esempi di backing service sono i database (es. MySQL e MondoDB) o servizi di messaggistica (es. RabbitMQ), servizi di SMTP per la posta o servizi di cache. Un'applicazione 12-factor non fa distinzione fra servizi in locale e servizi di terze parti: entrambi sono risorse connesse, accessibili via URL e credenziali salvate

nell'apposito file di configurazione. Una risorsa può essere collegata o scollegata da un deployment a piacimento.

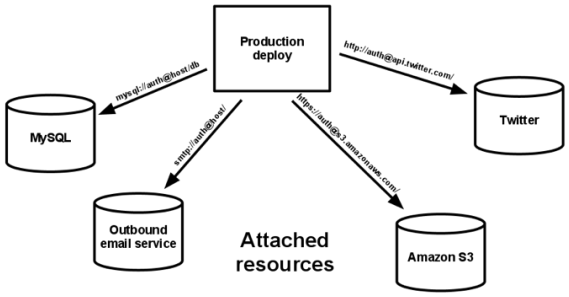


Figura 4

V. Build, release, esecuzione

Le fasi di build ed esecuzione dell'applicazione devono essere ben separate e una release deve essere facilmente identificabile.

Il processo di che porta il codebase a diventare un deploy viene identificato in tre fasi principali:

- Build: il codice del repo viene trasformato in una build eseguibile. Prendendo una determinata versione del codice di una determinata commit, i binari vengono compilati assieme agli asset appropriati.

- Release: la build prodotta nella fase precedente viene combinata con le impostazioni di configurazione del deployment desiderato.

- Esecuzione: l'applicazione viene eseguita nell'ambiente di destinazione.

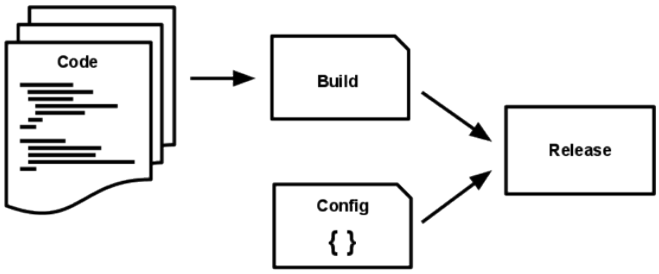


Figura 5

Mettendo in atto la separazione netta fra le tre fasi sarà impossibile effettuare dei cambiamenti nel codice in fase di esecuzione, dato che non si possono applicare queste modifiche "all'indietro". I tool per il deployment offrono spesso la possibilità di gestire le release,

ad esempio è possibile effettuare un rollback verso una release precedente. Ogni release deve possedere un ID univoco di rilascio (si può usare un timestamp o un numero incrementale) e non può essere modificata dopo la sua creazione. La fase di build viene sempre avviata da uno sviluppatore, mentre la fase di esecuzione può essere avviata anche autonomamente.

VI. Processi

L'applicazione deve eseguire solo processi stateless, che non condividono nulla con altre istanze. Tutti i dati persistenti devono essere salvati in un backing service, come ad esempio un database.

Lo spazio di memoria di un processo deve essere visto come una singola transazione breve. Un' applicazione 12-factor non deve mai dare per scontato che qualcosa messo in cache possa essere disponibile successivamente. Un'evidente violazione di questa regola è rappresentata dalle così dette sticky-sessions. Questo tipo di sessioni sono usate da diverse applicazioni web e prevedono di salvare in cache i dati di sessione di un singolo utente aspettando poi future richieste identiche dallo stesso utente.

VII. Binding delle porte

L'applicazione deve essere completamente self-contained e non deve affidarsi ad alcun servizio presente nell'ambiente di esecuzione.

Un'applicazione web eseguita su di un web server deve esportare HTTP come un servizio, effettuando un binding specifico ad una porta e rimanere in ascolto per richieste in entrata. HTTP non è l'unico esempio: quasi ogni software può essere eseguito tramite uno specifico binding tra processo e porta dedicata. Usare il binding permette inoltre ad un'applicazione di diventare il back service di un'altra tramite, per esempio, l'URL dedicato.

VIII. Concorrenza

L'applicazione deve preferire lo scaling orizzontale rispetto a quello verticale.

In una applicazione 12-factor i processi sono first class citizen. Sfruttando questo modello il programmatore può sviluppare applicazioni in grado di gestire senza problemi diversi livelli di carico di lavoro, assegnando ad ogni processo un tipo di lavoro specifico. Un processo può ad esempio gestire le richieste HTTP del web service, mentre compiti più lunghi da eseguire in background possono essere assegnati ad un altro processo. L'adesione a questa regola non esclude che un processo internamente possa sfruttare il mutithreading o l'esecuzione asincrona, semplicemente questo non basta. L'applicazione deve essere adatta inoltre all'esecuzione su più macchine fisiche. L'adesione a questo modello rende l'applicazione pronta a scalare orizzontalmente quando necessario.

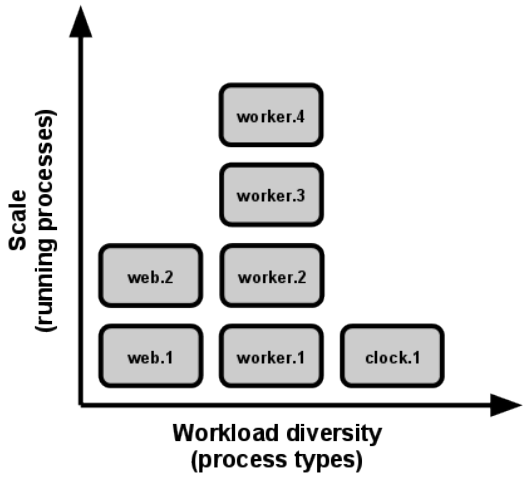


Figura 6

IX. Rilasciabilità

I processi di un'applicazione devono poter essere avviati o fermati quando necessario senza creare problemi all'utente.

Grazie a questa regola sono semplificati i processi di scaling, deploy e cambio della configurazione. I processi dovrebbero inoltre minimizzare il tempo di avvio: in questo modo si avrebbe un vantaggio nel momento della release ed in termini di robustezza dell'applicazione; facendo così il process manager potrebbe infatti muovere l'applicazione più velocemente verso una nuova macchina fisica. Per la stessa ragione la terminazione del processo dovrebbe essere graduale e sicura quando il process manager manda il segnale di SIGTERM o quando ci sono crash improvvisi dovuti ad esempio a problemi dell'hardware sottostante.

X. Parità fra sviluppo e produzione

Bisogna mantenere lo sviluppo, lo staging e la produzione il più possibile simili fra di loro.

Possiamo raggruppare le principali differenze tra gli ambienti di sviluppo e di produzione in tre categorie principali:

- Tempo: può essere necessario lavorare sul codice per giorni, settimane o anche mesi prima di andare in produzione.

- Personale: gli sviluppatori scrivono il codice mentre gli ops effettuano i deployment.

- Strumenti: gli sviluppatori usano uno stack, come Nginx o SQLite; per i deployment vanno invece installati Linux, Apache o MySQL.

Un'applicazione 12-factor va pensata per il rilascio continuo, dunque queste differenze vanno tenute al minimo. Per le differenze temporali c'è bisogno che il codice per il rilascio venga scritto in breve tempo (ore o addirittura minuti); per le differenze personali, invece, è necessario che anche gli sviluppatori siano coinvolti nella fase di deployment affinché possano osservare il comportamento di ciò che hanno scritto; per le differenze strumentali infine, bisogna mantenere i due ambienti di lavoro il più possibile simili.

Questa pratica viene chiamata comunemente "Continuous Integration/Continuous Delivery" (CI/CD) e può essere portata avanti tramite dei tool specifici; il tool più utilizzato attualmente è Jenkins.

XI. Log

I log devono essere trattati come stream di eventi.

Un log può essere infatti definito come uno stream di eventi aggregati e ordinati cronologicamente. Gli eventi vengono presi da tutti i processi attivi e da tutti i backing service. Non sarà quindi l'applicazione a doversi occupare dell'output stream, ma ogni processo dovrà occuparsi di scrivere ogni suo evento interno su stdout. Lo stream di eventi può essere visionato su di un terminale in tempo reale, indirizzato verso un file o può essere inviato ad un sistema di analisi per controllare il comportamento dell'applicazione. Le analisi sugli stream possono ricercare specifici eventi del passato, creare grafici per rappresentare specifiche tendenze oppure attivare degli allert basati su regole definite dall'utente.

XII. Processi di amministrazione

I processi di amministrazione devono essere separati dai normali processi. I primi sono processi che lo sviluppatore può voler eseguire saltuariamente. Essi sono ad esempio: migrazione del database, esecuzione di una console per avviare un codice arbitrario o monitorare un determinato comportamento, ed esecuzione di alcuni script specifici. Tali processi devono essere avviati in un ambiente identico a quello in cui lavorano gli altri processi, quindi su di una specifica release, partendo dalla stessa codebase e le configurazioni del dato deployment.

Capitolo 2 – Microservizi e Service mesh

Dopo aver fatto una panoramica sulle tecnologie utilizzate e le funzionalità offerte dal cloud computing, passiamo ora a parlare più nel dettaglio dei microservizi e dei problemi di questi ultimi, risolvibili tramite l'utilizzo del service mesh.

2.1 Microservizi

Abbiamo parlato spesso di microservizi; vediamo ora nello specifico cosa si intende con questo termine.

Seguendo la definizione data dal NIST, «a microservice is a basic element that results from the architectural decomposition of an application's components into loosely coupled patterns consisting of self-contained services that communicate with each other using a standard communications protocol and a set of well-defined APIs, independent of any vendor, product or technology».

Secondo quanto affermato nell'articolo *Microservices* su IEEE Software del 2018 [7] «[…] microservices are small applications with a single responsibility that can be deployed, scaled, and tested independently».

Da entrambe le definizioni deduciamo che la caratteristica principale di un microservizio è quella di svolgere compiti elementari rispetto al contesto dell'intera applicazione nella quale esso è inserito. Un'applicazione complessa, che un tempo sarebbe stata costruita seguendo un approccio monolitico, sarà ora formata da diversi microservizi interconnessi fra loro.

Abbiamo visto che i microservizi sono componenti essenziali per la realizzazione di applicazioni cloud native; tuttavia, continuando a seguire l'analisi riportata nell'articolo appena citato [7], possiamo anche affermare che essi presentano delle difficoltà oltre che alcuni svantaggi. La difficoltà maggiore è sicuramente rappresentata dalla scomposizione delle vecchie architetture monolitiche in servizi basilari; va inoltre considerato che con questo nuovo tipo di architettura aumenta anche la complessità del testing, del versioning, ed in generale della gestione dei vari stati dell'applicazione. Per quanto riguarda gli svantaggi, invece, possiamo notare come i microservizi necessitino per la loro gestione di esperienza e di un diverso tipo di tecnologia da dover applicare e quindi apprendere. Lo scopo principale secondo gli autori rimane quindi quello di dividere sistemi monolitici in microservizi attraverso un refactoring dell'intera applicazione; ridisegnare l'applicazione da capo sarebbe infatti molto più rischioso. Attraverso il DevOps sono poi rese disponibili tecniche per lo sviluppo, per il deployment e per la gestione di sistemi basati su microservizi, tecniche che si sono sviluppate sempre più negli ultimi anni. L'articolo [7] prosegue affermando che, come in ogni evoluzione software, il passaggio ad un'architettura a microservizi potrebbe influenzare alcuni importanti qualità:

- Sicurezza: dal momento che i dati devono essere scambiati tra microservizi, c'è bisogno che tale comunicazione sia sicura, per esempio tramite l'utilizzo di tecniche di criptazione. Anche i meccanismi di autenticazione dovrebbero essere implementati.

- Performance: in generale le performance offerte da applicazioni basate su microservizi sono inferiori a quelle offerte dalle stesse applicazioni basate su sistemi monolitici. Le performance, tuttavia, possono dipendere nel caso di microservizi da diversi fattori come la connessione di rete o l'overhead dovuto alla virtualizzazione.

- Affidabilità: i microservizi, essendo utilizzati in sistemi distribuiti, sono intrinsecamente meno affidabili di sistemi monolitici.

- Disponibilità: essa ora non dipende più soltanto dalla disponibilità del singolo microservizio, ma anche dall'integrazione tra i vari microservizi. D'altro canto, il tempo necessario per il deployment è ridotto, quindi la disponibilità generale è maggiore rispetto al passato.

- Manutenibilità: questo è uno dei punti di forza dell'approccio a microservizi grazie al fatto che ogni singolo microservizio è indipendente dagli altri.

- Testabilità: parlando di singolo microservizio sicuramente la fase di testing risulta essere più semplice rispetto ad un'intera applicazione monolitica. Se parliamo invece dei test sull'integrazione tra i vari microservizi la complessità cresce notevolmente.

Non abbiamo una definizione formale per un'architettura basata su microservizi; tuttavia, è possibile identificare caratteristiche che accomunano architetture di questo tipo. Gli autori James Lewis e Martin Fowler hanno descritto le principali fra queste caratteristiche all'interno del loro articolo online *Microservices* [8]. Andiamo ad analizzarle.

Componentizzazione tramite Servizi

Un componente può essere identificato come un'unità software che può essere rimpiazzata o aggiornata indipendentemente da altre unità. Tenendo a mente tale definizione possiamo affermare che il metodo utilizzato in un'architettura a microservizi per componentizzare il software consiste nello scomporre quest'ultimo in servizi. La ragione principale che porta all'utilizzo dei servizi è il fatto che è possibile fare il deployment di ognuno di essi indipendentemente dagli altri; in questo modo per apportare delle modifiche basterà fare nuovamente il deploy del servizio interessato senza dover necessariamente modificare l'intera applicazione.

Organizzazione basata sulle capacità aziendali

Solitamente la divisone di un'applicazione in più moduli segue la Conway 's Law, la quale afferma che «any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure». La divisione dell'applicazione segue quindi la struttura interna dell'organizzazione dalla quale l'applicazione viene sviluppata. Questo non vale però per le architetture basate su microservizi: in questo caso la divisine in microservizi è fatta in base alle capacità aziendali. Ogni servizio richiede un'implementazione ad ampio spettro di software per una determinata aria aziendale. Di conseguenza i team devono essere multifunzionali e contenere l'intera gamma di competenze richieste. Le figure 7 ed 8 rappresentano i due diversi tipi di divisione.

Prodotti non progetti

In un modello "a progetto", lo scopo è quello di produrre un pezzo di software che ad un certo punto viene considerato completato. Usando architetture a microservizi si tende a evitare questo tipo di modello preferendone uno "a prodotto", nel quale un team dovrebbe possedere e accompagnare un prodotto durante tutto il suo ciclo di vita. Il team di sviluppo ha quindi completa responsabilità riguardo al prodotto, seguendo il motto di Amazon: «you build, you run it».

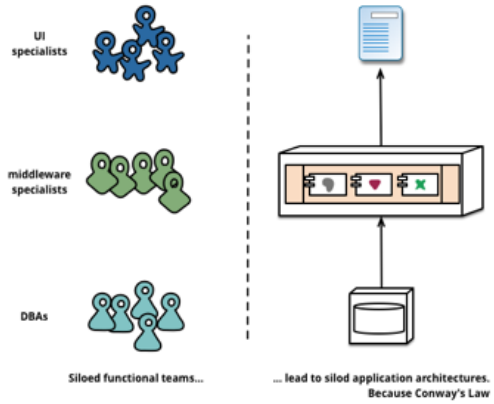


Figura 7

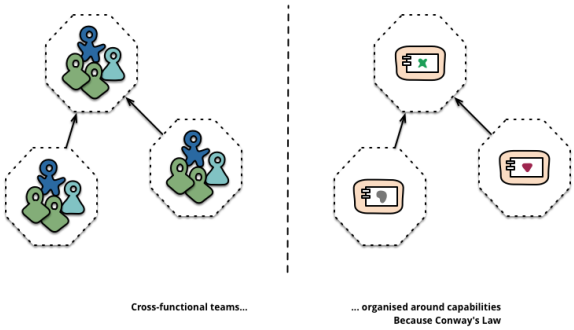


Figura 8

Smart endpoints e dumb pipes

Esistono meccanismi di comunicazione fra processi caratterizzati da una complessità significativa; tali meccanismi non sono utilizzati nell'ambito dei microservizi. In queste nuove architetture si preferisce l'utilizzo di protocolli di comunicazione definiti "dumb", con scarsa complessità e senza alcuna logica contenuta al loro interno. A contenere tutta la business logic e tutta la complessità sono quindi gli endpoint. I protocolli più comunemente utilizzati per la comunicazione sono HTTP request-response con API e messaggistica lightweight. Un altro approccio comune è quello di utilizzare un message bus implementato tramite, per esempio, RabbitMQ o ZeroMQ. Una delle più grandi difficoltà nel passaggio da un'architettura monolitica ad una a microservizi è quindi il passaggio da metodologie di comunicazioni "smart" a metodologie "dumb".

Gestione decentralizzata

Utilizzando una gestione centralizzata si tende a standardizzare alcuni tipi di piattaforme. Costruendo microservizi si preferisce un diverso tipo di approccio in cui gli sviluppatori, invece di usare una serie di standard, preferiscono produrre tool utili che possono essere utilizzati anche da altri sviluppatori per risolvere problemi simili. Con l'utilizzo dei microservizi sono nati anche nuovi tipi di contratti orientati al consumo effettivo fatto dall'utente. Il rovescio della medaglia della decentralizzazione sta nella maggiore responsabilità che i team di sviluppo si trovano a dover sostenere: essi sono responsabili di tutti gli aspetti del software 24/7.

Gestione dei dati decentralizzata

Allo stesso modo in cui viene decentralizzata la logica nelle architetture a microservizi si tende a decentralizzare anche il data storage. Le applicazioni monolitiche preferiscono un singolo database per i dati persistenti; il modello a microservizi predilige invece un database per ogni singolo servizio. Ai singoli microservizi vengono quindi assegnate istanze diverse dello stesso database o database completamente diversi l'uno dall'altro.

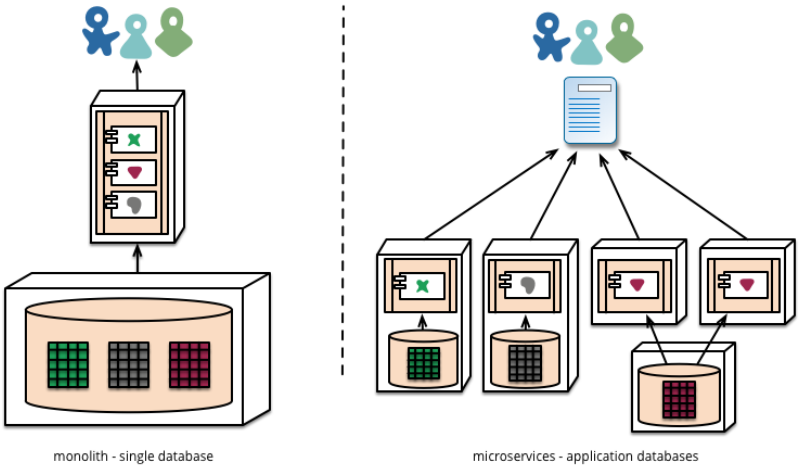


Figura 9

Avendo una struttura di storage decentralizzata, il problema principale da affrontare è quello degli aggiornamenti dei dati. Un approccio comune per risolvere tale problema è l'utilizzo delle transazioni; esse consistono in operazioni atomiche e quindi garantiscono consistenza fra i vari database. L'utilizzo delle transazioni introduce tuttavia un forte ritardo temporale nelle operazioni; esse inoltre sono di difficile implementazione all'interno di un modello a microservizi e spesso si preferisce non utilizzare affatto le transazioni per poi compensare in seguito eventuali incoerenze nei dati.

Automazione delle infrastrutture

Molti prodotti costruiti su un'architettura a microservizi vengono sviluppati seguendo il modello Continuous Integration/Continuous Delivery. I team che utilizzano questa metodologia fanno largo uso di automazione delle infrastrutture (in figura 10 un esempio di pipeline).

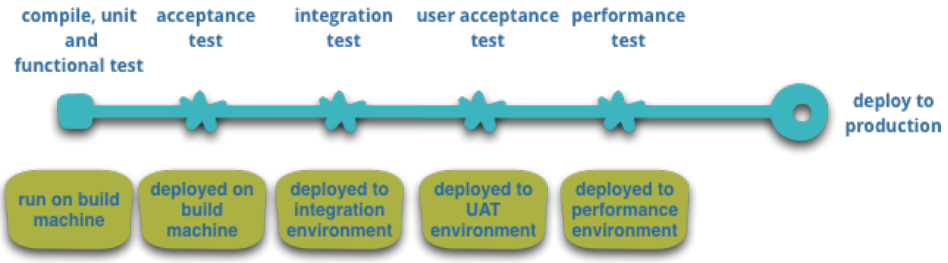


Figura 10

Per assicurarsi che il software funzioni bene, vengono eseguiti molti test automatici; inoltre quando un pezzo di software viene promosso allo stato successivo della pipeline vi è un deployment automatico nel nuovo ambiente. In questo caso, a differenza dei precedenti, non vi è grande differenza fra l'approccio usato in architetture monolitiche rispetto a quello usato in architetture a microservizi. Possiamo vedere i due approcci a confronto in figura 11.

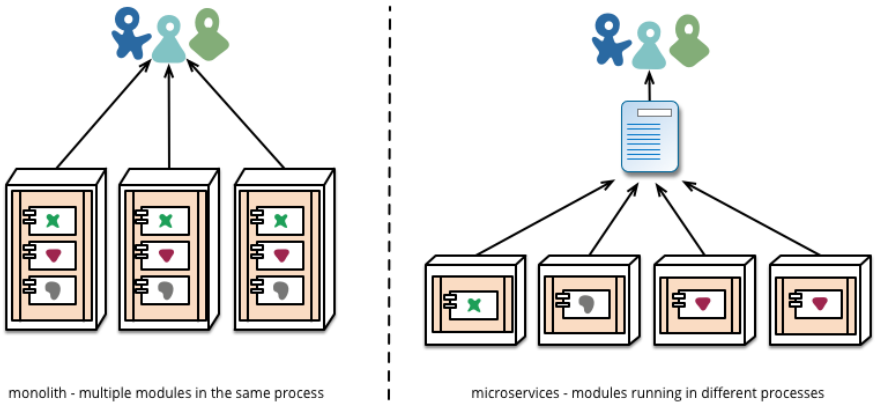


Figura 11

Design for failure

Usare i microservizi come componenti basilari di un'applicazione porta uno svantaggio significativo: l'applicazione deve essere in grado di tollerare il malfunzionamento di uno o più servizi; questo introduce maggiore complessità nella gestione. Dato che i servizi possono fallire in ogni momento è importante rilevare il loro malfunzionamento il prima possibile e se possibile ripristinarlo in maniera automatica. Per questa ragione nelle architetture a microservizi viene data maggiore enfasi al monitoring real-time dell'applicazione ed al testing dei casi in cui un microservizio fallisce. Alcuni tipi di monitoring semantico forniscono un sistema di allerta in grado di avvisare il team di sviluppo per tempo, di modo che questo possa intervenire al più presto.

Design evolutivo

Gli esperti di microservizi hanno spesso un background di progettazione evolutiva; la scomposizione in servizi è quindi vista come uno strumento ulteriore per consentire agli sviluppatori di controllare il cambiamento delle applicazioni senza rallentare il processo. Enfasi è data anche alla sostituibilità dei microservizi che può essere vista come un caso particolare del concetto più generale di evoluzione e cambiamento del software.

Le tecnologie basate sui microservizi si stanno evolvendo in fretta; un esempio con sempre maggiore rilevanza è il serverless computing. Con questo tipo di architettura le applicazioni sono avviate solo quando necessario. Quando l'utente avvia un codice, il provider assegna dinamicamente le risorse e l'utente paga il servizio solo fino alla fine dell'esecuzione. All'interno di questo modello ad occuparsi delle attività di routine è il provider del servizio; tali operazioni comprendono la gestione del file system e del sistema operativo, il bilanciamento del carico, il monitoraggio, la gestione della scalabilità e l'applicazione delle patch di sicurezza [9]. Grazie all'utilizzo dei microservizi è possibile realizzare servizi completamente o in parte serverless. I prodotti serverless sono assegnabili a due categorie [9]:

- Backend as a Service (BaaS): tramite questi servizi gli sviluppatori possono attingere a vari servizi e applicazioni di terze parti, ad esempio per l'autenticazione, la crittografia o i database. Tali servizi sono spesso accessibili spesso tramite API.

- Function as a Service (FaaS): questa modalità offre maggior controllo agli sviluppatori, i quali possono creare applicazioni personalizzate anziché sfruttare quelle

preconfezionate di terze parti. Gli sviluppatori possono chiamare le applicazioni tramite API gestite dal provider attraverso gateway API.

2.2 Analisi dei problemi derivati dai microservizi

Avendo molti microservizi che comunicano tra di loro e si evolvono, può diventare difficile per uno sviluppatore tenere traccia di ognuno di essi, gestirli nel migliore dei modi e apportare dei cambiamenti nel caso qualcosa vada storto. Vediamo quali sono i principali problemi che bisogna affrontare in questi casi.

Sicurezza

Secondo quanto riportato nell'articolo *Security in microservice-based systems: A Multivocal literature review* del 2021 [10], la sicurezza è divenuto un problema molto più rilevante rispetto a quanto avveniva per le applicazioni monolitiche. Viene stimato che per un'applicazione monolitica ogni 100 KLOC siano presenti in media 39 vulnerabilità, mentre in un'applicazione a microservizi le vulnerabilità salgono a 180.

Il fatto che le diverse funzionalità siano divise in differenti servizi ha allargato notevolmente la superficie di attacco; va inoltre considerato che la sicurezza è una proprietà da applicare globalmente all'intera applicazione e che non può essere rappresentata dalla somma dei vari livelli di sicurezza applicati alle singole parti; questo naturalmente rende il tutto più difficile da gestire. Nell'articolo [10] vengono identificati i principali problemi relativi alla sicurezza, oggetto di discussione in vari studi degli ultimi anni.

In primo luogo, come già anticipato in precedenza, a costituire una possibile superficie di attacco sono le comunicazioni tra i vari microservizi, le quali avvengono attraverso la rete. Se un microservizio è compromesso esso può infettare anche altri microservizi con cui è in comunicazione. Per contrastare questo problema la soluzione più semplice è l'utilizzo del protocollo TLS per la comunicazione; tuttavia alcune infrastrutture (tra cui anche Kubernetes) offrono dei meccanismi di autorizzazione inter-service. Utilizzando tale soluzione il problema diventa la coordinazione e l'aggiornamento delle regole di autorizzazione su di un sistema distribuito.

Un altro problema riguarda l'autenticazione ed in particolare dove debbano essere salvate le informazioni di autenticazione. Se esse venissero salvate in un authentication server, per esempio, sarebbe necessario aggiornare il server ogni qual volta venisse aggiunto o rimosso un nuovo utente o un nuovo microservizio; se le informazioni fossero invece mantenute all'interno di ogni singolo microservizio, allora ognuno di essi dovrebbe essere aggiornato ad ogni modifica. Anche per questo motivo la scomposizione in microservizi è un compito complesso. Esistono alcune soluzioni che definiscono dei processi per la scomposizione in grado di rispettare i vincoli di sicurezza e scalabilità.

A costituire una vulnerabilità ulteriore potrebbe essere l'utilizzo di immagini da repository pubblici o privati che potrebbero essere infetti. Una soluzione proposta per questo tipo di vulnerabilità è la così detta Moving Target Defenses (MTD), con la quale specifici componenti di sistema vengono modificati, in modo da disorientare gli attaccanti.

Vediamo ora quali sono i principali meccanismi di sicurezza da prendere in considerazione:

- Autorizzazione: descrive chi (utente o processo) può accedere a quale risorsa ed in che modo. Le principali soluzioni per questo tipo di problema richiedono un'autorizzazione di tipo RBAC (Role-Based Access Control), OAuth, PBAC (Policy-Based Access Control), XACML (eXtensible Access Control Markup Language) o Multilevel Security.

- Identity Management: si tratta il processo di identificazione di gruppi, singole persone o unità software attive; è molto importante per supportare autenticazione, autorizzazione e logging/auditing. L'identity management viene utilizzato per funzioni di autenticazione in protocolli come TLS, MTLS, SASL, X509 Certificates e SAML. Gli standard più indicati per l'autenticazione in applicazioni a microservizi sono OAuth, OpenID Connect, Single Sign-In (SSO), JSON Web Token (JWT).

- Controllo di accesso: si tratta della verifica che un'entità che richiede l'accesso ad una risorsa sia legittima ed abbia i diritti necessari. È formata da due parti importanti: autenticazione e autorizzazione.

- Comunicazione sicura: viene spesso ottenuta tramite l'utilizzo della criptazione e dalla mutua autenticazione; la prima fa in modo che i messaggi in transito tra una mittente e un destinatario non possano essere letti da terze parti; la seconda permette di stabilire una connessione sicura. I metodi più utilizzati sono: alcuni tipi di crittografia, Secure Data Exchange, 5G, NFV Security o Network Segmentation.

- Filtering: usato per limitare la connessione tra un sito e possibili siti malevoli; viene spesso implementato tramite l'utilizzo di firewall. Esso non è stato pensato per le applicazioni o microservizi; tuttavia, ci sono XML e application firewal che filtrano gli aspetti applicativi.

- Logging & Monitoring: con logging si intende la registrazione di eventi nel sistema, mentre il monitoring è usato per rilevare comportamenti anomali che possono essere indizio di un attacco in corso. La soluzione più comunemente usata è l'utilizzo di Intrusion Detection Systems (IDS).

- Controllo di esecuzione: un processo in esecuzione deve essere protetto da altri processi e inoltre la sua esecuzione deve essere ridotta ad un dominio specifico. Tale metodo non viene utilizzato di frequente.

- Security Information Management: le informazioni necessarie all'autenticazione e all'autorizzazione devono essere protette; se queste venissero violate, infatti, perderebbero la loro funzione primaria. Anche l'utilizzo di tale metodo è poco diffuso.

La sicurezza in ambito di applicazioni a microservizi è un tema sempre più importante. Negli ultimi anni ci si è tuttavia concentrati sulla messa in sicurezza di specifiche parti del sistema a sfavore di una metodologia per la costruzione di un'intera applicazione sicura.

Bilanciamento del carico lato client

In un'applicazione basata su un'architettura a microservizi è vitale distribuire il carico di lavoro tra le diverse repliche in modo da non sovraccaricare una replica in particolare; una distribuzione del carico tale che nessuna istanza sia sovraccarica viene chiamata load balancing. Il load balancing può essere fatto sia lato server che lato client: nel primo caso si parla di server side load balancing, nel secondo invece di client side load balancing.

Un'architettura di questo tipo offre una migliore gestione del pagamento in base all'utilizzo. In primo luogo offre una capacità infrastrutturale on demand praticamente illimitata: invece di sovrastimare il provisioning per poter gestire un carico di lavoro che non si conosce a priori, gli utenti possono utilizzare le risorse messe a disposizione dal cloud provider solo quando necessario. In secondo luogo con questo modello gli utenti pagano solamente per le risorse usate effettivamente e non per quelle necessarie a sostenere il picco di carico. Per finire, un'infrastruttura cloud è molto più grande di molti data center privati; l'economy of scale dei componenti hardware e della manutenzione aiutano quindi a mantenere bassi i costi. Tali caratteristiche rendono il cloud la soluzione adatta per applicazioni web a causa del loro carico di lavoro molto variabile. Secondo quando riportato nell'articolo *Client-side load balancer using cloud* di Sewook Wee e Huan Liu [11], scalare un'applicazione sul cloud è più difficile che farlo in un'infrastruttura tradizionale per diverse ragioni.

Per prima cosa in un'applicazione tradizionale è possibile scegliere fra diverse opzioni l'infrastruttura ottimale, mentre i cloud provider offrono solo un set limitato di componenti infrastrutturali fra cui scegliere. Le componenti di un'infrastruttura cloud, inoltre, come ad esempio le macchine virtuali, richiedono a volte uno scaling orizzontale per incrementare la capacità di carico. Come abbiamo già visto, il controllo sull'infrastruttura è molto minore in ambito cloud di quando non fosse per applicazioni enterprise; per concludere le macchine virtuali potrebbero avere malfunzionamenti frequenti.

Per via di queste caratteristiche, le applicazioni web in cloud tendono ad essere eseguite su cluster con diversi server standard, il che richiede una buona soluzione per il load balancing. La soluzione offerta nell'articolo è di tipo client side load balancing, grazie alla quale un client è in grado di prendere decisioni sull'indirizzamento del carico basandosi sulla lista di back-end web server e le informazioni sul loro carico.

Selezione del servizio migliore

All'interno di un'applicazione a microservizi potrebbero essere presenti diverse varianti di un determinato servizio. Ciò potrebbe essere dovuto alla presenza di diverse versioni di tale servizio che esistono contemporaneamente, oppure alla presenza su cloud di una nuova versione del servizio che però è ancora in fase di testing. Diventa essenziale a questo punto selezionare il servizio migliore da utilizzare in un caso specifico. La nuova versione del servizio va ad esempio utilizzata dal programmatore che intende testarla, ed essa non deve poter essere usata dall'utente finale in quanto potrebbe presentare dei problemi; la vecchia versione deve essere invece utilizzata dall'utente finale.

Tolleranza ai guasti e Circuit Breaking

Con il termine tolleranza ai guasti (in inglese "fault tolerance") intendiamo la proprietà che garantisce ad un sistema di continuare a funzionare anche se una delle sue parti smette di farlo; nel nostro caso le parti interessate sono i microservizi. I principali tipi di guasto che si possono verificare sono [12]:

- Due o più microservizi non riescono a comunicare fra di loro.
- Non è possibile accedere al database.
- I server sono down.
- L'applicazione non è in grado di connettersi al file system.
- Delay nell'esecuzione di alcuni servizi.

Al fine di garantire una buona tolleranza ai guasti c'è bisogno di prestare attenzione ad alcuni fattori nell'implementazione dell'applicazione stessa. Alcuni esempi sono [12]:

- Evitare guasti a cascata: bisogna garantire che il fallimento di un microservizio non causi effetti indesiderati su altri microservizi o sull'intera catena. Deve inoltre essere possibile rimettere in funzione il microservizio andato in errore.

- Evitare single point of failure: bisogna evitare che l'intera applicazione dipenda eccessivamente da un singolo componente. Nel caso questo avvenga comunque, è importante che sia possibile aggiustare tale componente nel minor tempo possibile.

- Gestione graceful degli errori: ogni microservizio deve gestire nel modo migliore gli eventuali errori terminando e lanciando un errore/eccezione in modo che sia possibile capire quale sia la causa dell'errore stesso.

- Avere buoni log: avere dei buoni log permette di capire quali siano le cause che hanno portato a determinati errori, e poter quindi evitare che una situazione simile si ripresenti in futuro.

- Eseguire un roll forward invece che un roll back: mentre nelle applicazioni tradizionali una buona pratica in caso di guasti è eseguire un roll back per tornare alla versione precedente al presentarsi del guasto, nelle architetture a microservizi è consigliabile eseguire un roll forward. Questo tipo di architettura favorisce infatti il continuous delivery nel quale viene fatto il deploy dei microservizi con frequenza e rapidamente; in caso di guasti è quindi possibile eseguire il deploy di una nuova versione del microservizio invece che tornare ad una versione precedente.

Ci sono diversi pattern che garantiscono la resilienza dei microservizi; uno di questi, e forse il più utilizzato, è il circuite breaking. Come suggerisce il nome, questo pattern impedisce le domande verso e le risposte da un microservizio quando questo smette di funzionare [13]. In questo modo il guasto sulla rete o su di un servizio in particolare non può propagarsi sull'intero circuito. Il pattern prevede che un client invochi un servizio remoto passando per un proxy chiamato Circuit Breaker. Il proxy rileva errori nel funzionamento del servizio; quando il numero di errori rilevati è superiore ad una certa soglia, il Circuit Breaker blocca la comunicazione per un certo periodo di tempo. Allo scadere del tempo di time-out il Circuit Breaker apre la comunicazione ad un ridotto numero di richieste: se tali richieste non falliscono allora il normale funzionamento viene ristabilito, altrimenti il tempo di time-out ricomincia da capo.

Il pattern prevede quindi tre stati per un Circuit Breaker [13]:

- Closed: questo stato è quello in cui il Circuit Breaker si trova durante il normale funzionamento dell'applicazione. Richieste e risposte passano tranquillamente attraverso il proxy ed esso conta il numero di failure in un determinato periodo di tempo. Quando il numero raggiunge la soglia lo stato passa ad "open".

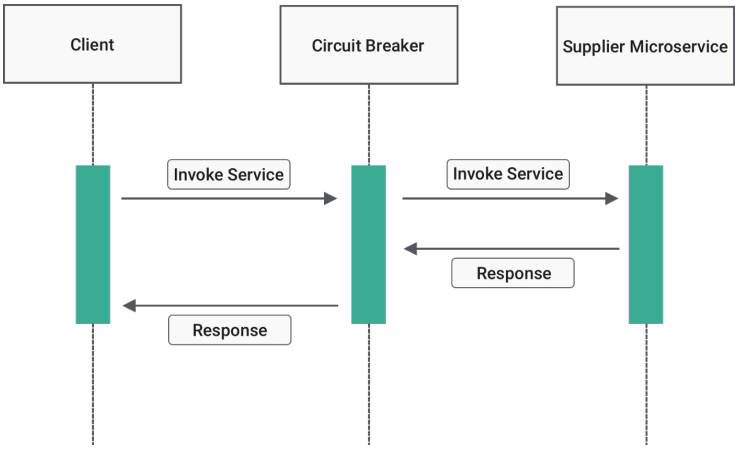


Figura 12

- Open: quando il Circuit Breaker è in questo stato le richieste verso il microservizio falliscono e viene lanciata un'eccezione. Dopo il periodo di time-out lo stato passa a "half-open"

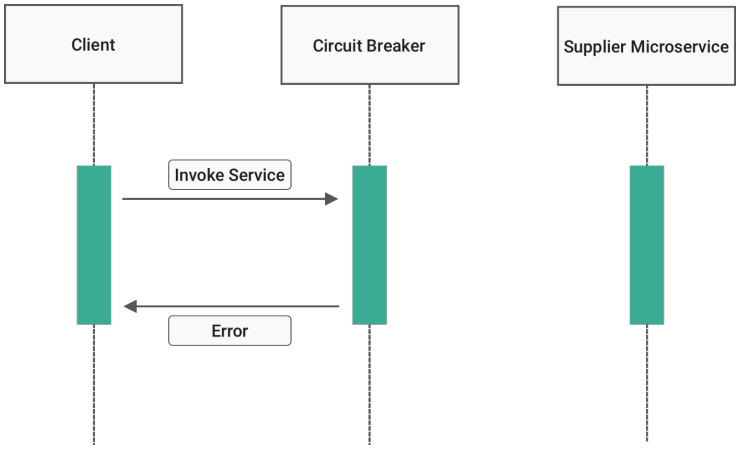


Figura 13

- Half-Open: un numero limitato di richieste vengono indirizzate verso il microservizio. Se le richieste non falliscono lo stato passa a "closed".

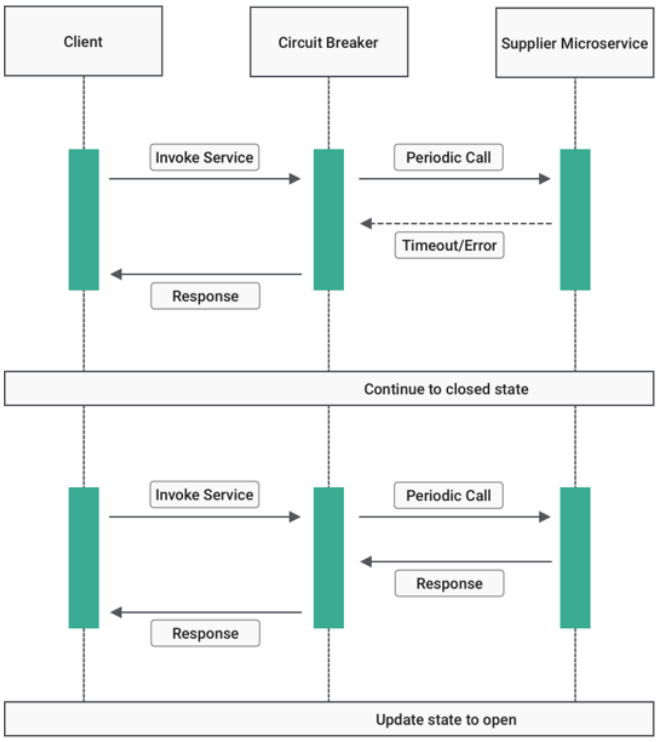


Figura 14

2.3 Service Mesh

Abbiamo visto come le applicazioni moderne si servono di diversi servizi (denominati microservizi), ognuno dei quali svolge un preciso compito servendosi talvolta di altri servizi. In questo tipo di architettura potrebbe però succedere che uno specifico microservizio si sovraccarichi perché riceve un carico eccessivo di richieste. Un esempio potrebbe essere il servizio responsabile del database: essendo il componente che contiene tutte le informazioni necessarie all'applicazione, esso riceve richieste dalla maggior parte degli altri microservizi. Per gestire il sovraccarico dovuto ad una tale situazione ci si avvale di una service mesh [14], che ha il compito di reindirizzare le richieste da un servizio ad un altro in modo da bilanciare il carico complessivo su tutti gli elementi. Così facendo una service mesh consente di controllare in che modo le diverse componenti di un'applicazione condividono i dati. Essa costruisce un livello infrastrutturale integrato direttamente all'interno dell'applicazione ed è quindi in grado di documentare le interazioni che avvengono fra le sue varie componenti, ottimizzando in questo modo la comunicazione.

Abbiamo già visto come in un'architettura a microservizi i singoli servizi, pur comunicando tra loro, sono totalmente indipendenti, ed il mancato funzionamento di uno di essi non compromette il funzionamento dell'intera applicazione. La comunicazione tra i vari microservizi è quindi una componente essenziale per il funzionamento di questo tipo di applicazione. La logica che implementa tale comunicazione può essere inserita all'interno di ogni singolo servizio senza il bisogno della service mesh. Essa diventa invece utile quando la comunicazione si fa più complessa; infatti il suo utilizzo all'interno di una applicazione cloud-native consente di includere un numero anche elevato di microservizi all'interno di un'applicazione che risulta essere funzionale.

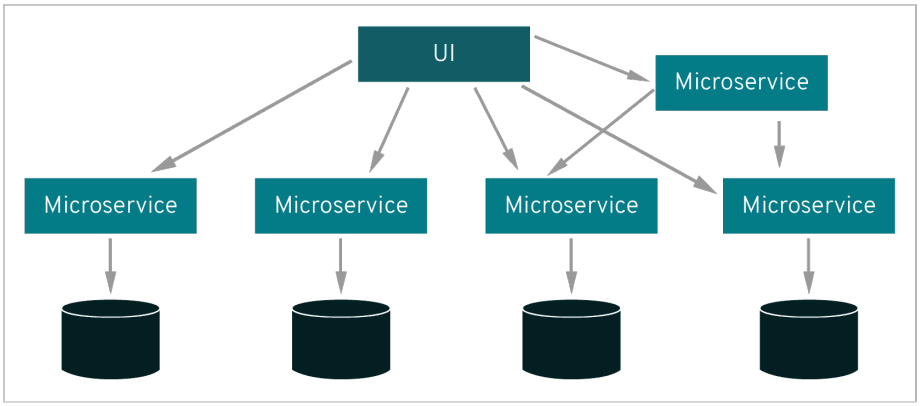


Figura 15

Andiamo a vedere come funziona nello specifico una service mesh [14]. Essa estrapola la logica che regola la comunicazione tra i vari microservizi e la astrae a livello di infrastruttura; per farlo si serve del concetto di proxy. Un proxy si inserisce nel mezzo della connessione fra due servizi e ne gestisce la comunicazione. Al fine di essere inseriti all'interno dell'infrastruttura, tali proxy sono inseriti come "sidecar" affiancati ai vari servizi (abbiamo visto cosa si intende con "sidecar containers" parlando dell'architettura di Kubernetes). Tutti i proxy "sidecar" quindi, disaccoppiati dai vari servizi, formano una rete di mesh. Se non si utilizzasse la service mesh, ogni microservizio dovrebbe contenere internamente la logica necessaria a regolare la comunicazione service-to-service. Il suo utilizzo permette quindi agli sviluppatori di dedicarsi completamente agli obiettivi aziendali e di diagnosticare nel modo migliore eventuali problemi di comunicazione, poiché la logica che gestisce tale comunicazione è distaccata dal servizio stesso.

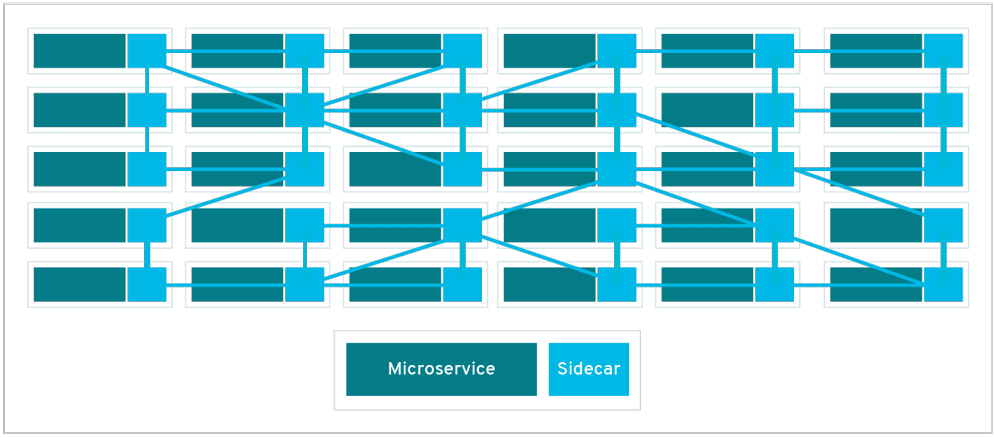


Figura 16

L'utilizzo della service mesh permette anche di ottimizzare la comunicazione fra i vari servizi [14]. Ogni nuovo servizio aggiunto all'applicazione rende più complessa la gestione della comunicazione e introduce nuovi possibili problemi. Individuare le cause di problemi riscontrati all'interno di un'architettura a microservizi complessa può risultare quindi quasi impossibile senza l'utilizzo di una service mesh; essa è infatti in grado di catturare ogni singolo aspetto della comunicazione fra i servizi e di rappresentarlo sotto forma di parametri prestazionali. Le informazioni che la service mesh fornisce nel tempo riguardo le comunicazioni possono essere usate per migliorare l'efficienza e l'affidabilità di queste ultime.

Istio

Uno dei metodi più utilizzati per l'implementazione di una service mesh su di un'applicazione a microservizi è tramite la piattaforma open source Istio. Essa può essere eseguita sia on premise che in ambienti sul cloud, all'interno di container Kubernetes, su microservizi o su macchine virtuali.

L'architettura di Istio è suddivisa in un piano dati ed un piano di controllo [15].

- Piano dati: viene affiancato un "proxy sidecar" ad ogni microservizio tramite deployment; tale proxy ha il compito di indirizzare le richieste da e verso il microservizio a cui si affianca. Questi sono i proxy che costituiscono la service mesh e che intercettano le comunicazioni fra i servizi.

- Piano di controllo: gestisce e configura i vari “proxy sidecar”, i componenti necessari per applicare le policy e quelli per acquisire i dati di telemetria.

Istio offre anche strumenti per facilitare il passaggio da applicazioni monolitiche ad applicazioni cloud native, e le sue funzionalità permettono di eseguire una infrastruttura a microservizi distribuita. Tali funzionalità sono [15]:

- Gestione del traffico: esso permette di configurare delle regole per controllare il flusso di traffico e delle chiamate API.

- Sicurezza: oltre a fornire il canale di comunicazione di base, Istio fornisce anche autenticazione, autorizzazione e crittografia della comunicazione tra i servizi.

- Visibilità: esso mette a disposizione dashboard personalizzabili per il monitoring ed il logging delle comunicazioni. Grazie al monitoring è possibile vedere l’impatto che hanno le attività di Istio sulle prestazioni dell’intera applicazione.

Capitolo 3 - Creazione di un microservizio per l'autenticazione

Passiamo ora a vedere il lavoro svolto in azienda per la creazione di un microservizio che gestisce l'autenticazione degli utenti sull'applicazione.

L'applicazione sulla quale si è lavorato è Eye4Task, la quale permette agli operatori sul campo di connettersi con esperti in remoto o di registrare ciò che avviene davanti ai propri occhi a mani libere tramite l'utilizzo di smart glasses.

L'applicazione è scritta interamente in Kotlin, viene utilizzato Spring Boot come framework e Maven come project manager; vengono sfruttati i servizi offerti dal cloud di Amazon AWS anche se in concomitanza con la trasformazione in un'architettura a microservizi si sta procedendo ad una migrazione verso Microsoft Azure. Essa è stata pensata sin dalla sua creazione per un utilizzo delle tecnologie cloud native, ma queste non sono sfruttate a pieno: l'applicazione presenta infatti ancora una struttura monolitica, e c'è stato quindi bisogno di iniziare il processo di trasformazione in un'architettura a microservizi.

3.1 Scelta del microservizio da realizzare e studio preliminare

Il primo passo è stato quindi quello di individuare il microservizio più adatto per iniziare il processo di transizione. La scelta si è rivelata semplice, si è optato in fatti per il microservizio per l'autenticazione, in quanto tale componente necessitava già di un'integrazione. In seguito a specifiche richieste da parte di clienti dell'azienda, si era infatti deciso di affiancare all'autenticazione tramite semplice token JWT, già presente nell'applicazione, un'autenticazione tramite pattern OAuth2. Il primo metodo sarebbe stato utilizzato per l'autenticazione di server che richiedessero accesso all'applicazione, il secondo metodo sarebbe invece stato usato per gli utenti che volessero autenticarsi.

Una volta scelto il microservizio da implementare è stato necessario uno studio preliminare su tutti i concetti base necessari per la realizzazione del servizio stesso. Vediamo una panoramica di tali concetti.

Spring Boot e Spring Security

Come abbiamo già visto, per l'implementazione di Eye4Task è stato utilizzato il framework Spring Boot. Tale framework semplifica la creazione di applicazioni in linguaggio Java

(può essere utilizzato anche con Kotlin, come nel nostro caso), fornendo un approccio modulare [16].

Come la maggior parte dei moderni framework Spring Boot implementa la così detta Inversion of Control (IoC), in italiano inversione del controllo. Questo significa che sono le componenti presenti all'interno del framework stesso a richiamare il codice scritto dal programmatore al contrario di quanto avviene normalmente, quando è il codice del programmatore che richiama altri componenti, come ad esempio le funzioni di una libreria [16]. In Spring in particolare tale meccanismo viene portato avanti grazie all'utilizzo della così detta Dependency Injection (DI). Utilizzando questo pattern è sufficiente dichiarare le dipendenze di cui un componente ha bisogno: quando questo verrà istanziato un injector si occuperà di risolvere le dipendenze di cui necessita. Se è la prima volta che si tenta di iniettare una dipendenza, l'injector istanzierà tale dipendenza, la salverà in un contenitore e la restituirà; se non è la prima volta che la dipendenza viene invocata l'injector restituirà la copia salvata nel contenitore [16].

Per la realizzazione del microservizio oltre a Spring Boot è stato necessario utilizzare anche Spring Security. Questo framework è focalizzato sul fornire autenticazione e autorizzazione alle applicazioni Java. Il funzionamento di Spring Security è basato sull'utilizzo di Servlet Filters. Quando un client manda una richiesta all'applicazione, il framework decide quale filtro e quale servlet applicare basato sul path specifico dell'URI. Solo un servlet può gestire una singola richiesta, ma i filtri formano una catena ordinata [17].

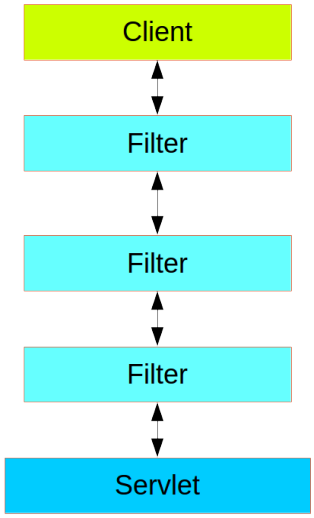


Figura 17

L'ordine dei vari filtri è molto importante ed è gestito attraverso dei @Beans di tipo Filter e con l'annotazione @Order. Spring Security viene installato come un FilterChainProx all'interno di un'applicazione Spring Boot. Dal punto di vista del container, Spring Security è un filtro singolo, ma al suo interno può contenere filtri addizionali, ognuno con un ruolo specifico [17].

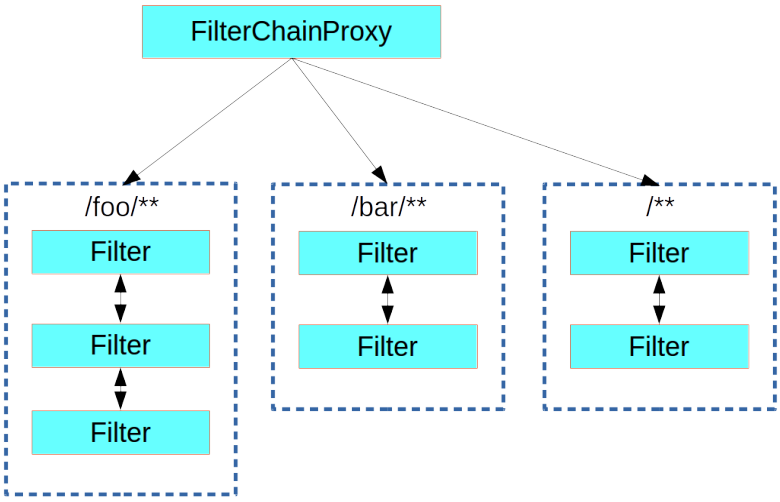


Figura 18

Maven

Come project management per lo sviluppo dell'applicazione è stato utilizzato Maven; uno strumento usato per la build automation prevalentemente di progetti in Java. Le fasi principali che vengono automatizzate sono: la compilazione in binario, il packaging dei binari, l'esecuzione di test, il deployment e la documentazione relativa al progetto. In questo modo si riduce il carico di lavoro sul programmatore e diminuisce la possibilità che commetta errori.

La caratteristica principale sulla quale si basa il funzionamento di Maven è l'utilizzo del file pom.xml (acronimo di Project Object Model). Tale file serve per descrivere la struttura del progetto ed è diviso a sua volta in cinque parti: relazione fra i diversi file pom, build settings, project information, build environment e configurazione dell'ambiente Maven.

Un altro componente importante è rappresentato dai goal: ognuno di essi rappresenta una task specifico che contribuisce al building ed alla gestione del progetto. Abbiamo poi una cartella repository tramite la quale lo sviluppatore può gestire le librerie, e per finire un file setting.xml, usato per la configurazione dei repository, dei proxy e delle librerie [18].

REST API

Una API REST è un'interfaccia di programmazione delle applicazioni (API) conforme ai vincoli dettati da un'architettura di tipo REST. Il termine REST è l'acronimo di REpresentational State Transfer. Per adeguarsi all'architettura REST, una API deve avere le seguenti sei caratteristiche [19]:

- Usare un'interfaccia uniforme: le risorse devono quindi essere univocamente identificate da un singolo URL.

- Deve essere di tipo client-server: deve esserci quindi una chiara distinzione tra client e server. Tutto ciò che riguarda l'interfaccia è nel dominio del client. Tutto ciò che riguarda accesso ai dati, gestione del carico e sicurezza è nel dominio del server.

- Deve effettuare solo operazioni stateless: in nessun caso deve essere salvato o gestito in altro modo uno stato interno all'API.

- Deve effettuare del caching delle risorse: tutte le risorse devono poter permettere caching a meno che non venga espressamente indicato che questo non sia possibile.

- Deve avere un layered system: REST prevede un'architettura composta da diversi livelli di server, ognuno con un preciso ruolo.

- Deve poter fornire code on demand (quest'ultima caratteristica è l'unica facoltativa): se richiesto il server può mandare codice eseguibile al client.

Token JWT

Il JSON Web Token (JWT) rappresenta uno standard (RFC 7519) per lo scambio di informazioni tra vari servizi basato su uno schema in formato JSON. Le informazioni vengono scambiate sotto forma di token che può essere firmato in vari modi. Si può utilizzare l'algoritmo HMAC oppure una coppia di chiavi pubblica/privata utilizzando formati standard come RSA o ECDSA. Tale tipologia di token viene spesso usata per l'autenicazione su server web o all'interno di meccanismi come OAuth2.

Un JWT è composto da tre parti fondamentali [20]:

- Header: contiene la tipologia di token (in questo caso JWT perché si tratta di un JSON Web Token) e il tipo di algoritmo di crittografia utilizzato (HS256 nell'esempio in figura 19)

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Figura 19

- Payload: Contiene le informazioni da scambiare, esse possono essere distinte in tre tipi diversi:

    1. Parametri registrati: rappresentano informazioni predefinite riguardo al token. Alcuni esempi sono elencati di seguito.

        - iss (issuer): stringa contenente il nome identificativo dell'entità che ha generato il token.

        - sub (subject): stringa che contiene l'oggetto del messaggio.

        - aud (audience): array contenente valori che indicano le abilità del token. In fase di validazione del token è possibile bocciare tutti i token che non hanno un valore di audience specificato.

        - exp (expiration): numero intero che indica fino a quando il token sarà valido.

        - nbf (not before): valore intero che indica la data dopo la quale il token sarà valido.

        - iat (issued at): valore intero che indica la data nella quale il token è stato generato. Permette di conoscere l'età del token.

        - jti (jwt id): identificativo univoco del token.

    2. Parametri pubblici: possono essere compilati a piacimento, facendo attenzione al contenuto che vi si inserisce per evitare conflitti.

3. Parametri privati: qui si può inserire quello che si vuole poiché la struttura JSON garantisce piena flessibilità.

Un esempio di payload di un token JWT è presente in figura 20.

```
{
  "iss": "NOME_APP",
  "name": "Mario Rossi",
  "iat": 1540890704,
  "exp": 1540918800,
  "user": {
     "profile": "editor"
  }
}
```

Figura 20

- Signature: per generare il token si codifica in base 64 l'header e il payload. Dopo aver unito i due risultati separandoli con un punto, si applica l'algoritmo di crittografia indicato nell'header utilizzando una chiave segreta. Utilizzando l'algoritmo HMAC SHA256 il token si potrebbe ottenere come mostrato nella figura 21.

```
HMACSHA256(
    base64UrlEncode(header) + "." +
    base64UrlEncode(payload),
    secret)
```

Figura 21

Dati il payload e l'header mostrati rispettivamente in figura 19 e 20 il token JWT ottenuto sarà quello mostrato in figura 22.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJOT01FX0FQUCIsIm5hbWUiO
iJNYXJpbyBSb3NzaSIsImlhdCI6MTU0MDg5MDcwNCwiZXhwIjoxNTQwOTE4ODAwLCJ1c2V
yIjp7InByb2ZpbGUiOiJlZGl0b3IifX0.ZUlQ_W8U5oW4bRTEuUEwcSNcvsazitYoXrrDj
tP6wkY
```

Figura 22

OAuth2

Abbiamo visto che per via di esplicite richieste da parte di client dell'azienda è stato deciso di implementare sul microservizio una autenticazione basata su OAuth2; andiamo a vedere ora in cosa consiste questo protocollo.

OAuth2 permette ad applicazioni di ottenere un accesso limitato agli account utente su un servizio HTTP. Esso delega l'autenticazione dell'utente ad un servizio che possiede un account per quell'utente; tale servizio autorizza un'applicazione di terze parti ad avere accesso a quell'account. Il protocollo descrive quattro ruoli [21]:

- Resource Owner: è lo user che autorizza un'applicazione ad avere accesso al suo account. L'accesso dell'applicazione all'account dello user è limitata dallo scopo dichiarato per tale autorizzazione (ad esempio accesso per sola scrittura o per sola lettura).

- Client: è l'applicazione che desidera accedere all'account utente. Prima che l'accesso sia permesso deve essere autorizzata dallo user e validata dalla API.

- Resource Server: è il server che ospita l'account utente.

- Authorization Server: esso verifica l'identità dell'utente e rilascia il token all'applicazione.

OAuth2 fornisce diversi tipi di flow adatti a diversi tipi di client API [22].

- Authorization code flow: questo flow è quello più comunemente usato lato server e per applicazioni web mobile. È simile a quello utilizzato per accedere agli account Facebook o Google.

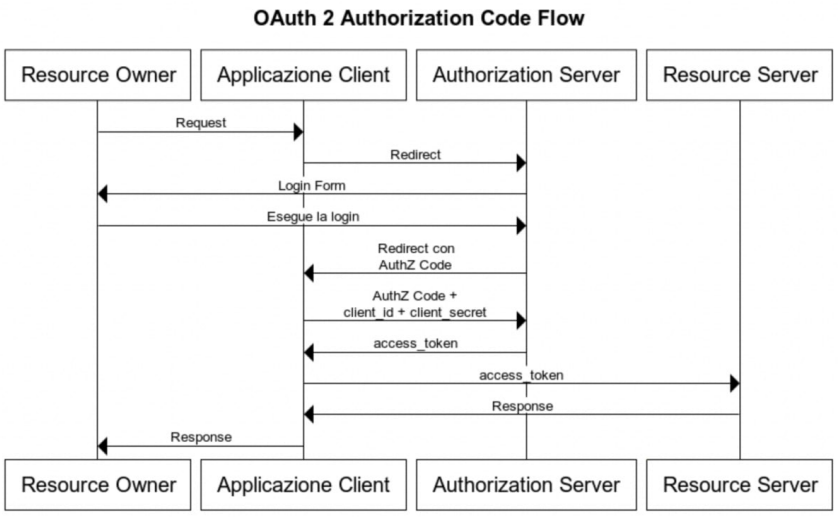


Figura 23

- Implicit flow: questo flow richiede che il client ricavi direttamente il proprio access token. Viene utilizzato quando non è possibile salvare le credenziali utente sul client poiché questo potrebbe essere accessibile anche da terze parti. È indicato per tutte quelle applicazioni web, desktop o mobile che non includono alcun componente server.

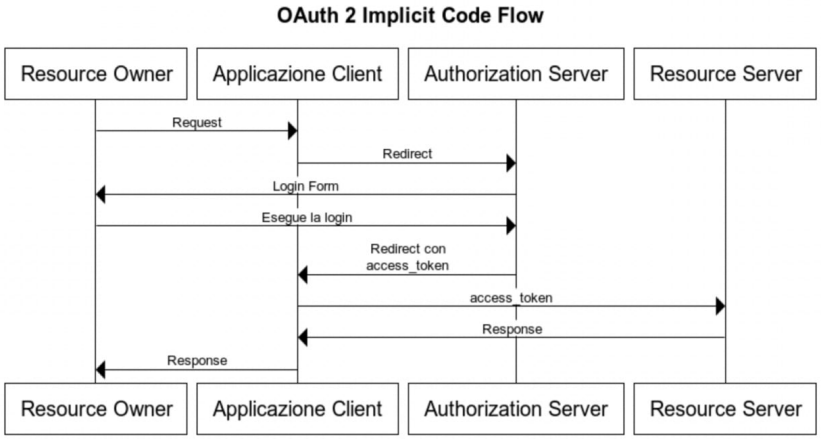


Figura 24

- Resource owner password credentials flow: richiede l'inserimento di username e password, dato che con questo flow le credenziali saranno parte integrante della richiesta è consigliato solo quando applicato a client fidati.

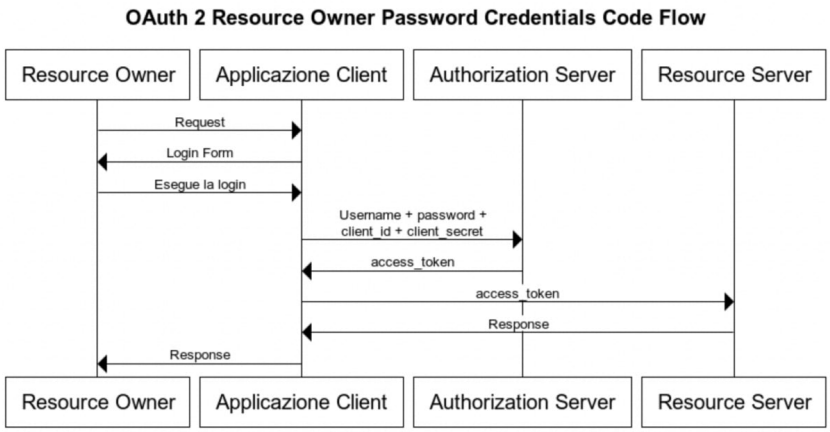


Figura 25

- Client credentials flow: utilizzato per autenticazione server-to-server. In questo caso il client agisce per proprio conto e non per conto di un individuo. In molti casi questo flow permette agli utenti di specificare le proprie credenziali sull'applicazione del client in modo da poter accedere alle risorse che sono sotto il controllo del client stesso.

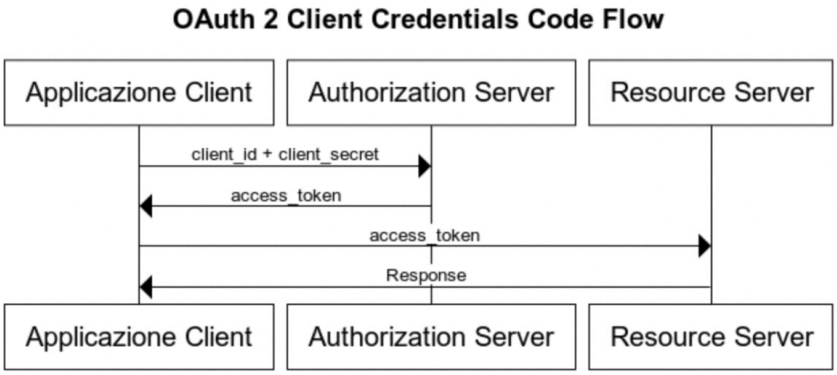


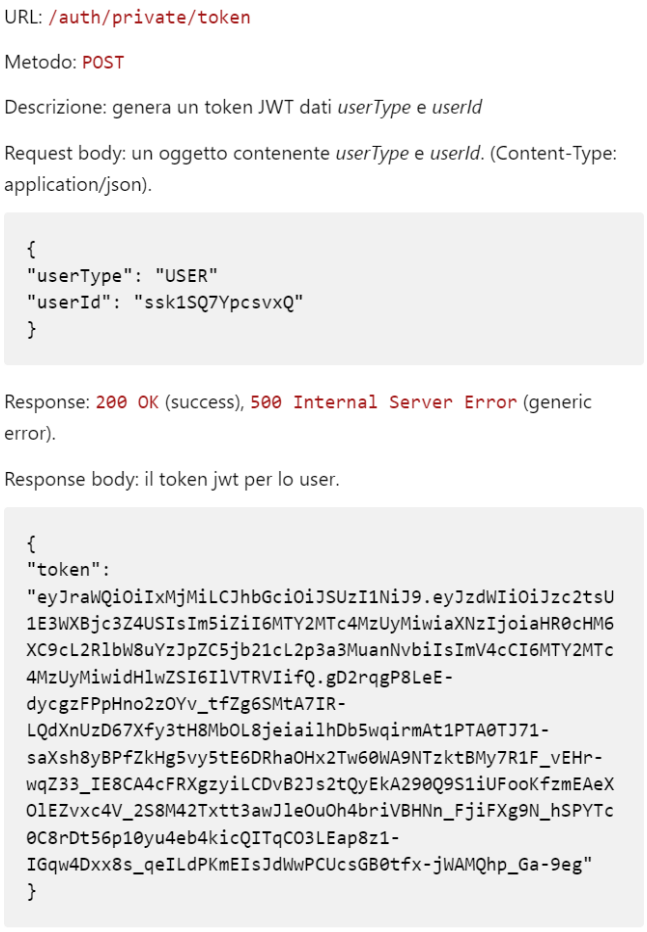
Figura 26

3.2 Creazione di un microservizio che implementa autenticazione tramite JWT token

Finito lo studio preliminare sugli argomenti necessari alla scrittura del microservizio, è iniziata la creazione del microservizio stesso.

Per prima cosa si è implementato il filtro per il solo token JWT, salvo poi integrare il filtro per il pattern OAuth2 in un secondo momento. A questo punto il microservizio contava due API (una terza API sarebbe stata aggiunta in seguito in concomitanza con l'integrazione del pattern OAuth2). Le due API REST in questione sono una POST ed una GET:

- POST generateToken: prende come parametri il tipo di utente ed il suo id, passati nel body della richiesta, e restituisce il token generato per quell'utente.


URL: `/auth/private/token`

Metodo: `POST`

Descrizione: genera un token JWT dati *userType* e *userId*

Request body: un oggetto contenente *userType* e *userId*. (Content-Type: application/json).

```
{
"userType": "USER"
"userId": "ssk1SQ7YpcsvxQ"
}
```

Response: `200 OK` (success), `500 Internal Server Error` (generic error).

Response body: il token jwt per lo user.

```
{
"token":
"eyJraWQiOiIxMjMiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJzc2tsU
1E3WXBjc3Z4USIsIm5iZiI6MTY2MTc4MzUyMiwiaXNzIjoiaHR0cHM6
XC9cL2RlbW8uYzJpZC5jb21cL2p3a3MuanNvbiIsImV4cCI6MTY2MTc
4MzUyMiwidHlwZSI6IlVTRVIifQ.gD2rqgP8LeE-
dycgzFPpHno2zOYv_tfZg6SMtA7IR-
LQdXnUzD67Xfy3tH8MbOL8jeiailhDb5wqirmAt1PTA0TJ71-
saXsh8yBPfZkHg5vy5tE6DRhaOHx2Tw60WA9NTzktBMy7R1F_vEHr-
wqZ33_IE8CA4cFRXgzyiLCDvB2Js2tQyEkA290Q9S1iUFooKfzmEAeX
OlEZvxc4V_2S8M42Txtt3awJleOuOh4briVBHNn_FjiFXg9N_hSPYTc
0C8rDt56p10yu4eb4kicQITqCO3LEap8z1-
IGqw4Dxx8s_qeILdPKmEIsJdWwPCUcsGB0tfx-jWAMQhp_Ga-9eg"
}
```

Figura 27

- GET checkToken: accetta un token nella richiesta http; se questo passa attraverso il filtro restituisce il tipo e l'id dell'utente relativo a quel token.

URL: `/auth/private/check`

Method: `GET`

Descrizione: verifica che il toke sia valid e in caso affermativo restituisce *userType* e *userId*.

Header: `Authorizzazion: Bearer {token}` token = eyJraWQiOiIxMjMiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJzc2tsU1E3WXBjc3Z4USIsIm5iZiI6MTY2MTc4MzUyMiwiaXNzIjoiaHR0cHM6XC9cL2RlbW8uYzJpZC5jb21cL2p3a3MuanNvbiIsImV4cCI6MTY2MTc4MzUyMiwidHlwZSI6IlVTRVIifQ.gD2rqgP8LeE-dycgzFPpHno2zOYv_tfZg6SMtA7IR-LQdXnUzD67Xfy3tH8MbOL8jeiailhDb5wqirmAt1PTA0TJ71-saXsh8yBPfZkHg5vy5tE6DRhaOHx2Tw60WA9NTzktBMy7R1F_vEHr-wqZ33_IE8CA4cFRXgzyiLCDvB2Js2tQyEkA290Q9S1iUFooKfzmEAeXOlEZvxc4V_2S8M42Txtt3awJleOuOh4briVBHNn_FjiFXg9N_hSPYTc0C8rDt56p10yu4eb4kicQITqCO3LEap8z1-IGqw4Dxx8s_qeILdPKmEIsJdWwPCUcsGB0tfx-jWAMQhp_Ga-9eg.

Response: `200 OK` (success), `403 Forbiden` (expired or invalid jwt), `500 Internal Server Error` (generic error).

Response body: un oggetto contenente *userType* e *userId*

```
{
"userType": "USER",
"userId": "ssk1SQ7YpcsvxQ"
}
```

Figura 28

Ecco come si presenta nel microservizio l'implementazione delle due API REST:

```
@RestController
@RequestMapping(AUTHPRIVATE)
class AuthPrivateController @Autowired constructor(
    private val jwtUtils: JwtUtils,
) {

    fun TokenUserDetails.toWeb() = TokenContent().also {
        it.userId = id
        it.userType = type.toWeb()
    }

    fun TokenUserDetails.Type.toWeb() = UserType.valueOf(name)
    fun UserType.toModel() = TokenUserDetails.Type.valueOf(value)

    @PostMapping(TOKEN)
    fun generateToken(@RequestBody tokenRequest: TokenRequest): TokenResponse {
        val token = jwtUtils.generate(principalId = tokenRequest.userId, type
        tokenRequest.userType.toModel())
        val response = TokenResponse()
        response.token = token

        return response
    }

    @GetMapping(CHECK)
    fun checkToken(): TokenContent {
        val user = SecurityContextHolder.getContext().authentication.principal as
        TokenUserDetails
        return user.toWeb()
    }
}
```

Come è possibile vedere dal codice, sia per la POST generateToken che per la GET checkToken le richieste e le risposte sono rappresentante da classi opportunamente create. In particolare le classi TokenRequest, TokenResponse e TokenContent sono classi java automaticamente generate grazie all'utilizzo di un plugin Maven a partire dalla descrizione all'interno di un file YAML seguendo lo standard OpenAPI. Tale standard permette una comprensione delle API descritte, sia da parte del programmatore che da parte del programma stesso, senza il bisogno di dover consultare necessariamente il codice o la documentazione. Tutte e tre le classi contengono quindi tutti gli argomenti necessari e specificati all'interno del file auth.yaml, oltre ai metodi setter e getter per la lettura e la scrittura dei valori all'interno di tali argomenti. Vediamo di seguito la struttura e l'implementazione del file auth.yaml che descrive le tre classi sopra citate, oltre che la descrizione delle due API.

```yaml
openapi: 3.0.0
info:
  version: "7.9.0-SNAPSHOT"
  title: E4T-AUTH
  description: e4t authentication api

paths:
  "/auth/token":
    post:
      tags:
        - auth
      operationId: generateToken
      requestBody:
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/TokenRequest"
      responses:
        "200":
          description: OK
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/TokenResponse"
        "4XX":
          $ref: "#/components/responses/Error"
        "5XX":
          $ref: "#/components/responses/Error"

  "/auth/check":
    get:
      tags:
        - auth
      operationId: checkToken
      responses:
        "200":
          description: OK
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/TokenContent"
        "4XX":
          $ref: "#/components/responses/Error"
        "5XX":
          $ref: "#/components/responses/Error"
```

```yaml
components:
  responses:
    OK:
      description: OK

    Error:
      description: error
      content:
        application/json:
          schema:
            $ref: "#/components/schemas/ErrorResponse"

  schemas:
    TokenRequest:
      properties:
        userType:
          $ref: "#/components/schemas/UserType"
        userId:
          $ref: "#/components/schemas/Id"
      required:
        - userType
        - userId

    TokenResponse:
      properties:
        token:
          type: string
      required:
        - token

    TokenContent:
      properties:
        userType:
          $ref: "#/components/schemas/UserType"
        userId:
          $ref: "#/components/schemas/Id"
      required:
        - userType
        - userId

    Id:
      type: string
      description: system autogenerated id
      pattern: "[a-zA-Z0-9]{1,30}"

    UserType:
      type: string
      enum:
        - USER
        - GUEST
```

Dopo aver definito ed implementato le API necessarie al funzionamento del microservizio, c'è stato bisogno di configurare Spring Security. È stato infatti necessario fare in modo che si passasse attraverso il filtro per l'autenticazione solo nel caso in cui l'API chiamata fosse quella per il check del token JWT; nel caso in cui venga richiamata la POST per la creazione del token, non è necessaria l'autenticazione da parte dell'utente. Per fare ciò è stata utilizzata la classe JWTSecurityConfig, che estende WebSecurityConfigurerAdapter ed è decorata dall'annotazione @Configuration. All'interno della classe viene aggiunto il filtro (rappresentato dalla classe JwtAuthenticationFilter) alla catena di filtri di Spring Security in modo che, quando viene fatta una richiesta al microservizio, questa passi attraverso il filtro specificato. Per far sì che il filtro non venga attraversato quando le richieste sono fatte verso la GET checkToken, il path corrispondente a tale API è stato specificato tra quelli da ignorare.

Ecco di seguito il codice per la configurazione del filtro:

```
@Configuration
class JWTSecurityConfig @Autowired constructor(
    private val jwtUtils: JwtUtils,
    private val eftExceptionHandler: AuthExceptionHandler
) : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        klog.info { "Configuring jwt authentication" }


        addCommonCfg(http, "auth/**")


       http.exceptionHandling().authenticationEntryPoint(eftExceptionHandler)


        val jwtAuthenticationFilter =
            JwtAuthenticationFilter(jwtUtils, eftExceptionHandler)


        http.addFilterAfter(
            jwtAuthenticationFilter,
            AbstractPreAuthenticatedProcessingFilter::class.java
        )

    }

    override fun configure(web: WebSecurity) {
        klog.info { "Configuring noAuth endpoints" }
        web.ignoring()
            .antMatchers("/auth/private/token")
    }
}
```

L'implementazione del filtro avviene all'interno della classe JwtAuthenticationFilter, che estende OncePerRequestFilter e sovrascrive il metodo doFilterInternal.

```
class JwtAuthenticationFilter(
    private val jwtUtils: JwtUtils,
    private val authenticationEntryPoint: AuthenticationEntryPoint
) : OncePerRequestFilter() {


    override fun doFilterInternal(
        request: HttpServletRequest,
        response: HttpServletResponse,
        filterChain: FilterChain
    ) {
        try {
            val jwt = getToken(request)
            val jwtContent = jwtUtils.getContent(jwt)

            val tokenUserDetails = TokenUserDetails(
                id = jwtContent.principalId, type = jwtContent.type
            )

            SecurityContextHolder.getContext().authentication =
UsernamePasswordAuthenticationToken(
                tokenUserDetails,
                null,
                emptyList()
            )
        } catch (e: TokenAuthenticationException) {
            authenticationEntryPoint.commence(request, response, e)
            return
        }

        filterChain.doFilter(request, response)

    }

}
```

Il filtro prende il token dalla richiesta HTTP e tramite il metodo getContent ricava userType e userId dal token stesso. Internamente la getContent verifica che l'utente che ha fatto richiesta sia autorizzato, in caso contrario un'eccezione vene lanciata (vedremo in seguito come

sono gestite le eccezioni). Il controllo sul token viene effettuato come mostrato nel codice riportato di seguito.

```
if(jwsObject.verify(verifier)){
    val jwtProcessor: ConfigurableJWTProcessor<SecurityContext> =
DefaultJWTProcessor()
    val keySource: JWKSource<SecurityContext> = ImmutableJWKSet(JWKSet(publicJWK))
    val expectedJWSAlg: JWSAlgorithm = JWSAlgorithm.RS256
    val keySelector: JWSKeySelector<SecurityContext> =
JWSVerificationKeySelector(expectedJWSAlg, keySource)
    jwtProcessor.jwsKeySelector = keySelector
    jwtProcessor.jwtClaimsSetVerifier = DefaultJWTClaimsVerifier(
        JWTClaimsSet.Builder().issuer(issuer).build(),
        setOf("sub", "exp", "type", "iss")
    )
    val claims = jwtProcessor.process(token, null)
    return JwtContent(
        type = TokenUserDetails.Type.valueOf(claims.getStringClaim(claimType)),
        principalId = claims.getStringClaim(claimPrincipal)
    )
} else {
    throw InvalidRSAKey()
}
```

Per tale controllo, come per tutte le operazioni che comprendono la gestione del token, è stata utilizzata la libreria Nimbus. Grazie all'utilizzo di tale libreria è stato possibile fare diversi controlli sul token come mostrato nel codice. Per prima cosa viene verificato che il token sia stato firmato con la coppia di chiavi RSA pubblica/privata desiderata; se questo requisito è soddisfatto si verificano i parametri interni del token, quali ad esempio issuer ed expiration date. Tutti i parametri necessari al controllo del token, come ad esempio chiavi RSA ed issuer (ma anche la durata del tempo di validità del token a partire dalla sua creazione o altri parametri), sono inseriti all'interno di un file di configurazione in formato YALM (application.yaml).

```
jwt:
  expiration: 86400000
  issuer: https://demo.c2id.com/jwks.json
  key:
    public: "{chiave pubblica}"
    private: "{chiave private}"
    id: 123
```

Gestire in questo modo i parametri di configurazione è, come abbiamo visto, buona prassi secondo la terza legge delle 12-factor application rules.

3.3 Aggiunta al microservizio del protocollo OAuth2 per l'autenticazione

Creato il microservizio con le API REST sopra citate e con il filtro per il token JWT, si è passati all'integrazione al suo interno del protocollo OAuth2.

Per prima cosa è stato necessario implementare una nuova API che fornisse informazioni utili all'autenticazione tramite OAuth2, quali issuer, clientId, e scope. Come per i parametri necessari alla creazione e al check del token JWT, i valori corrispondenti ai parametri sopra indicati sono stati inseriti all'interno del file application.yaml.

```
security:
  oauth2:
    issuer: https://sts.windows.net/d09accb6-ced1-4861-9ac9-3bf7b058c349/
    clientId: 65a72bc-59130-4c0c8-b8dbaa-ec040f0318
    scope: https://graph.microsoft.com/User.Read https://graph.microsoft.com/of-
fline_access
    username:
      claim: unique_name
```

La nuova API aggiunta è la GET getInfo, la quale non fa altro che ritornare i parametri necessari per la costruzione di un token in grado di superare i controlli del filtro OAuth2, che sarebbe stato successivamente aggiunto al microservizio.

URL: `/auth/public/info`

Method: `GET`

Descrizione: ritorna i valori dei parametri *issuer*, *clientId* e *scope*.

Response: `200 OK` (success), `500 Internal Server Error` (generic error).

Response body: un oggetto contenente *issuer*, *clientId* e *scope*.

```
{
"issuer": "https://sts.windows.net/d09accb6-ced1-4861-
9ac9-3bf7b058c349/",
"clientId": "65a72bc-59130-4c0c8-b8dbaa-ec040f0318",
"scope": " https://graph.microsoft.com/User.Read
https://graph.microsoft.com/offline_access",
}
```

Figura 29

Vediamo come si presenta il codice per l'implementazione di tale API.

```
@RestController
@RequestMapping(AUTHPUBLIC)
class AuthPublicController @Autowired constructor(
    private val oAuth2Properties: Oauth2Properties
) {

    fun Oauth2Properties.toWeb() = OAuth2InfoContent().also{
        it.issuer = issuer
        it.clientId = clientId
        it.scope = scope
    }

    @GetMapping(INFO)
    fun getInfo(): OAuth2InfoContent {
        return oAuth2Properties.toWeb()
    }
}
```

Come possiamo vedere dal codice, anche in questo caso, come nella definizione delle precedenti API, è stata usata una classe auto generata. La classe in questione è OAuth2InfoContent, e come nei casi precedenti è stata descritta seguendo lo standard OpenAPI all'interno del file auth.yaml (nel file è presente anche la descrizione della GET getInfo).

```yaml
openapi: 3.0.0
info:
  version: "7.9.0-SNAPSHOT"
  title: E4T-AUTH
  description: e4t authentication api

  "/auth/info":
    get:
      tags:
        - auth
      operationId: get oauth2 info
      responses:
        "200":
          description: OK
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/OAuth2InfoContent"
        "4XX":
          $ref: "#/components/responses/Error"
        "5XX":
          $ref: "#/components/responses/Error"

components:
    OAuth2InfoContent:
      properties:
        issuer:
          type: string
        clientId:
          $ref: "#/components/schemas/ClientId"
        scope:
          type: string
      required:
        - issuer
        - clientId

    ClientId:
      type: string
```

In seguito alla definizione della nuova API si è passati all'aggiunta del nuovo filtro che implementasse il pattern Oauth2. È stato quindi necessario modificare la classe JWTSecurityConfig in modo da inserire il nuovo filtro prima del filtro precedente. In questo modo una richiesta fatta verso un URI protetto sarebbe passato prima attraverso il filtro Oauth2 e poi attraverso il vecchio filtro con il controllo sul token JWT.

```
@Configuration
class JWTSecurityConfig @Autowired constructor(
    private val jwtUtils: JwtUtils,
    private val oAuth2TokenUtil: OAuth2TokenUtil,
    private val eftExceptionHandler: AuthExceptionHandler
) : WebSecurityConfigurerAdapter() {

    override fun configure(http: HttpSecurity) {
        klog.info { "Configuring jwt authentication" }


        addCommonCfg(http, "auth/**")


       http.exceptionHandling().authenticationEntryPoint(eftExceptionHandler)


        val oAuth2AuthenticationFilter =
            OAuth2AuthenticationFilter(oAuth2TokenUtil, eftExceptionHandler)

        val jwtAuthenticationFilter =
            JwtAuthenticationFilter(jwtUtils, eftExceptionHandler)


        http.addFilterAfter(
            oAuth2AuthenticationFilter,
            AbstractPreAuthenticatedProcessingFilter::class.java )
        .addFilterAfter(
            jwtAuthenticationFilter,
            AbstractPreAuthenticatedProcessingFilter::class.java
        )

    }

    override fun configure(web: WebSecurity) {
        klog.info { "Configuring noAuth endpoints" }
        web.ignoring()
            .antMatchers("/auth/private/token")
            .antMatchers("/auth/public/info")
    }
}
```

Vediamo dal codice come sia stato creato un nuovo oggetto rappresentante il nuovo filtro per l'autenticazione tramite OAuth2 (classe Oauth2AuthenticationFilter). Il nuovo filtro e quello precedente vengono aggiunti alla catena di filtri di Spring Security uno di seguito all'altro. Oltre a ciò è stato aggiunto il path della getInfo all'elenco di path da ignorare. La nuova API non ha bisogno infatti di autenticazione quando questa viene richiamata.

Ricapitolando, a questo punto dello sviluppo, tutte le richieste fatte verso il path auth/private/check sarebbero passate prima per il filtro con autenticazione tramite OAuth2 e poi attraverso il nuovo filtro con autenticazione sul token JWT. Questo comportamento non corrispondeva a quello desiderato; c'era quindi bisogno di un modo per distinguere le richieste che richiedevano autenticazione tramite token JWT dalle richieste che richiedevano autenticazione tramite OAuth2. Per risolvere tale problema si è deciso di distinguere una richiesta di autenticazione dall'altra utilizzando l'issuer contenuto all'interno del token stesso. Entrambi i tipi di autenticazione prevedono infatti l'invio nell'header della richiesta di un Bearer token. I token inviati per un tipo di autenticazione o per l'altro risultano quindi totalmente indistinguibili se non vengono aperti e non si leggono i parametri contenuti al loro interno. La soluzione ha quindi richiesto i seguenti passaggi:

- Viene inoltrata una richiesta al path auth/private/check con un Bearer token inserito nell'header.

- La richiesta passa attraverso il primo filtro inserito nella catena, quello cioè che implementa OAuth2.

- Prima di procedere con il filtro, il token viene aperto e si verifica se l'issuer contenuto al suo interno sia uguale a quello per il pattern OAuth2.

- Se l'issuer corrisponde la richiesta passa per il filtro OAuth2. Una volta passato per il filtro e passati tutti i controlli, la richiesta proverà a passare anche per il filtro JWT, ma qui verrà bloccato avendo già svolto i controlli nel filtro precedente.

- Se l'issuer non corrisponde a quello per OAuth2 il primo filtro viene ignorato e la richiesta passa direttamente al secondo filtro. Il secondo filtro svolgerà tutti i controlli dal momento che la richiesta non è stata processata dal filtro precedente.

Vediamo quindi come è stato implementato il filtro per OAuth2.

```
class OAuth2AuthenticationFilter(
    private val oAuth2TokenUtil: OAuth2TokenUtil,
    private val authenticationEntryPoint: AuthenticationEntryPoint
) :
    OncePerRequestFilter() {

    companion object {
        private const val claimIssuer = "iss"
    }

    override fun doFilterInternal(
        request: HttpServletRequest,
        response: HttpServletResponse,
        filterChain: FilterChain
    ) {
        try {
            val token = getToken(request)

            val tokenUserDetails = TokenUserDetails(
                id = oAuth2TokenUtil.getUsernameFromToken(token), type =
                TokenUserDetails.Type.USER
            )

            SecurityContextHolder.getContext().authentication =
            UsernamePasswordAuthenticationToken(
                tokenUserDetails,
                null,
                emptyList()
            )
        } catch (e: TokenAuthenticationException) {
            authenticationEntryPoint.commence(request, response, e)
            return
        }

        filterChain.doFilter(request, response)
    }
```

Esattamente come per il filtro precedente, la classe OAuth2AuthenticationFilter estende OncePerRequestFilter e sovrascrive il metodo doFilterInternal. Per fare in modo che il filtro venisse eseguito in base alla presenza o meno del giusto valore nel campo issuer, è stato necessario sovrascrivere anche il metodo shouldNotFilter.

```
override fun shouldNotFilter(request: HttpServletRequest): Boolean {

    try {
        val token = getToken(request)
        val claims = JWSObject.parse(token).payload

        val json = claims.toJSONObject()
        val issuer = json.getValue(claimIssuer)

        if (issuer == oAuth2TokenUtil.oauth2IssuerUrl) {
            request.setAttribute("executeJwtFilter", false)
            return false
        }

        return true
    } catch (e: TokenAuthenticationException) {
        authenticationEntryPoint.commence(request, null, e)
        return true
    }
}
```

All'interno di questo metodo, come possiamo vedere, viene fatto il parse del token e viene estratto il valore del campo issuer. Se tale valore corrisponde a quello desiderato per il pattern OAuth2, allora il metodo ritorna false (quindi il filtro viene eseguito) e si comunica attraverso l'inserimento di un attributo nella richiesta HTTP che il filtro successivo non deve essere eseguito. In caso contrario il metodo torna true, quindi il filtro attuale non viene eseguito e si passa al filtro successivo. È stato inoltre necessario implementare tale metodo anche all'interno del JwtAuthenticationFilter, come mostrato di seguito.

```
override fun shouldNotFilter(request: HttpServletRequest): Boolean {

      if (request.getAttribute("executeJwtFilter") != null &&
          request.getAttribute("executeJwtFilter") == false) {
        return true
    }
    return false
}
```

Qui si verifica che l'attributo executeJwtFilter sia presente e uguale a false; in caso affermativo il metodo ritorna true e quindi il filtro non viene eseguito, poiché i controlli sono stati già fatti nel filtro precedente. In caso contrario viene ritornato false e si procede quindi con i controlli previsti dal filtro JWT sul token.

Anche all'interno del filtro per OAuth2, come per il filtro precedente, si è fatto uso della libreria Nimbus per il controllo dei campi del token.

```
val jwtProcessor: ConfigurableJWTProcessor<SecurityContext> = DefaultJWTProcessor()
val keySource: JWKSource<SecurityContext> = RemoteJWKSet(jwksURI)
val expectedJWSAlg: JWSAlgorithm = JWSAlgorithm.RS256
val keySelector: JWSKeySelector<SecurityContext> =
      JWSVerificationKeySelector(expectedJWSAlg, keySource)
jwtProcessor.jwsKeySelector = keySelector
jwtProcessor.jwtClaimsSetVerifier = DefaultJWTClaimsVerifier(
      JWTClaimsSet.Builder().issuer(oauth2IssuerUrl).build(),
      setOf("sub", "iat", "exp", "scp", usernameClaim, "iss")
      )
return jwtProcessor.process(token, null)
```

Ad essere controllati sono i campi scope, subject, issued at, expiration, issuer e il campo selezionato dal file di configurazione per indicare lo username dell'utente. Con l'implementazione dei due filtri e della logica per fare in modo che una richiesta venga processata da uno solo dei due, è terminato lo sviluppo delle funzionalità base del microservizio.

La struttura del progetto Maven è quella presentata in figura 30.

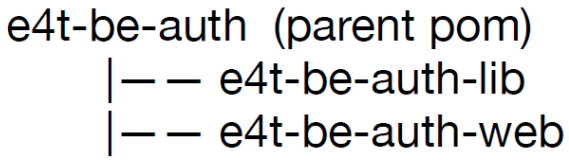


Figura 30

Il microservizio è infatti costituito da due moduli, e4t-be-auth-lib e e4t-be-auth-web. Il modulo lib contiene l'implementazione di un REST template service per l'esposizione del servizio all'esterno. Il modulo web invece contiene la logica presentata fin ora. I due moduli (ognuno con un proprio file pom.xml) sono collegati ad un pom padre (e4t-be-auth). Il pom padre è a sua

volta figlio di un pom Nexus (repository manager usato per il processo di CI/CD) e4t-be-parent; la struttura totale risultante è quindi quella in figura 31.

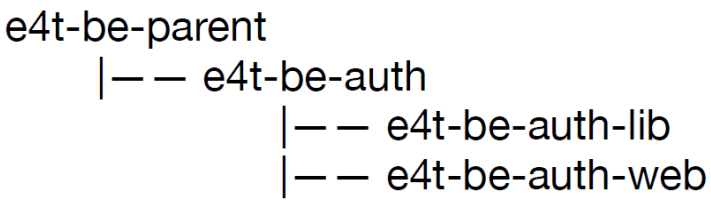


Figura 31

3.4 Exception handler e REST template

Una volta sviluppato il microservizio nelle sue funzionalità principali è stato necessario gestire le eccezioni lanciate e costruire un'interfaccia tramite REST template all'interno del modulo lib, per fare in modo che il servizio sia raggiungibile dall'esterno.

Exception handler

È stato necessario gestire il caso in cui un token risultasse scaduto o invalido, o ci fossero problemi di altro tipo in seguito alla chiamata ad una API. Per tale scopo si è implementata la classe AuthExceptionHandler. All'interno di tale classe vengono catturate le eccezioni lanciate nel codice, nello specifico quelle lanciate dalla libreria Nimbus per la gestione dei token, e processate in modo da ritornare al chiamante dell'API il formato di errore desiderato. Dopo essere passati attraverso la AuthExceptionHandler, le eccezioni lanciate saranno presentate nel formato descritto dalla data class riportata di seguito.

```
data class AuthErrorResponse(
    val code: String?,
    val message: String?,
    val logCorrelationId: String,
    val i18n: String?
)
```

All'interno della AuthExceptionHandler viene fatto un override del metodo commence, il quale rappresenta il punto di ingresso all'intera classe.

```
override fun commence(
        request: HttpServletRequest,
        response: HttpServletResponse?,
        authenticationException: AuthenticationException
    ) {
        val jwtAuthenticationException = authenticationException as
TokenAuthenticationException

        val tokenException = jwtAuthenticationException.cause as ExecutionException

        val responseEntity = manageAuthenticationException(request, tokenException)
        response?.status = responseEntity.statusCodeValue

        response?.contentType = "application/json"
        val body = objectMapper.writeValueAsString(responseEntity.body)

        response?.writer?.write(body)

        klog.debug(tokenException) { "An exception was caught" }
    }
```

La funzione manageAuthenticationException, chiamata all'interno della commence, contiene al suo interno un when basato sul tipo di eccezione, e ritorna un oggetto del tipo newResponseEntity. A partire da tale oggetto ritornato, nella commence si procede a modificare la risposta HTTP di conseguenza. Di seguito viene mostrata la funzione manageAuthenticationException.

```
fun manageAuthenticationException(description: String, exception: Exception):
ResponseEntity<Any> {
    return when (exception) {
        is ExpiredJwtException -> {
            newResponseEntity(
                code = ErrorCode.JWT_TOKEN_EXPIRED.code,
                requestDescription = description,
                cause = exception,
                status = FORBIDDEN,
                i18nMessage = I18nMessage.TOKEN_EXPIRED.of()
            )
        }
        is InvalidJwtException, is MissingHeaderException, is MissingJwtException,
is InvalidRSAKey -> {
            newResponseEntity(
                code = ErrorCode.JWT_TOKEN_INVALID.code,
                requestDescription = description,
                cause = exception,
                status = UNAUTHORIZED,
                i18nMessage = I18nMessage.TOKEN_INVALID.of()
            )
        }

        else -> newResponseEntity(
            requestDescription = description,
            cause = exception,
            status = UNAUTHORIZED
        )
    }
}
```

Come è possibile vedere dal codice, i valori dei parametri code, status e i18nMessage sono scelti in base al tipo di eccezione lanciata. In questo modo si ha una comprensione migliore dell'errore quando questo si presenta; ciò può essere molto utile in fase di debug.

REST template service

Per permettere la comunicazione con servizi esterni è stato utilizzato RestTemplate (il client REST di Spring), che fornisce funzioni in grado di esporre le API all'esterno del microservizio. Tali funzioni sono state implementate all'interno della classe RESTTemplateService e sono: generateToken, checkToken e getInfo, ognuna delle quali espone una delle tre API di cui è costituito il microservizio.

```
class RESTTemplateService(

    restTemplateBuilder: RestTemplateBuilder,
    private val endpoint: String

    ) {

    companion object {
        const val AUTHPRIVATE = "auth/private"
        const val AUTHPUBLIC = "auth/public"
        const val TOKEN = "token"
        const val CHECK = "check"
        const val INFO = "info"
    }
    private val restTemplate: RestTemplate

    init {
        restTemplate = restTemplateBuilder
            .errorHandler(AuthErrorHandler()).build()
    }

    fun generateToken(tokenRequest: TokenRequest) : TokenResponse? {
        return restTemplate.postForObject("${endpoint}/${AUTHPRIVATE}/${TOKEN}",
tokenRequest,  TokenResponse::class.java)
    }

    fun checkToken(bearerToken: String) : TokenContent? {
        val headers : MultiValueMap<String, String> = LinkedMultiValueMap()
        headers.add("Content-Type", "application/json")
        headers.add("Authorization", bearerToken)
        return restTemplate.exchange(
            "${endpoint}/$AUTHPRIVATE/$CHECK", HttpMethod.GET,
             HttpEntity<Any>(headers),
             TokenContent::class.java
             ).body
    }

    fun getInfo(): OAuth2InfoContent? {
        return restTemplate.getForObject(
            "${endpoint}/${AUTHPUBLIC}/${INFO}", OAuth2InfoContent::class.java
             )
    }

}
```

Per far sì che gli errori ritornati siano nello stesso formato desiderato, e prima descritto, si è utilizzata la classe AuthErrorHandler. Tale classe estende ResponseErrorHandler e sovrascrive i metodi hasError e handleError.

```
class AuthErrorHandler : ResponseErrorHandler {
    private val mapper = ObjectMapper()

    @Throws(Exception::class)
    override fun hasError(httpResponse: ClientHttpResponse): Boolean {
        return (httpResponse.statusCode.is4xxClientError
                || httpResponse.statusCode.is5xxServerError)
    }


    override fun handleError(httpResponse: ClientHttpResponse) {
        val responseStatus: HttpStatus =
            HttpStatus.valueOf(httpResponse.rawStatusCode)

        if (httpResponse.statusCode.is4xxClientError) {
            val errorResponse: ErrorResponse =
                mapper.readValue(httpResponse.body.bufferedReader(),
                   ErrorResponse::class.java)
            throw TokenException(errorResponse.message, responseStatus,
                   errorResponse.i18n, errorResponse.code)
        } else {
            throw GenericAuthException()
        }
    }
}
```

Gli errori catturati da questa classe sono quindi ritornati nel formato costituito da code, message, logCorrelationId e i18n.

Conclusioni

Abbiamo fatto quindi una panoramica delle tecnologie principali e del funzionamento del cloud computing, soffermandoci a considerare quali vantaggi e quali svantaggi esso comporta. Abbiamo investigato sul significato di cloud native application e sui principi base per la costruzione di un'applicazione di questo tipo. Siamo andati più nel dettaglio nello studio di un'architettura a microservizi considerando, anche in questo caso, i vantaggi e gli svantaggi; abbiamo visto i benefici che porta la service mesh all'interno di un'architettura di questo tipo.

Per concludere, è stato presentato un microservizio sviluppato per HeadApp, e sono stati illustrati i passaggi che hanno portato alla sua implementazione. Questo è solo l'inizio di un processo di transizione che porterà Eye4Task a passare da un'architettura monolitica ad un'architettura a microservizi. I prossimi passi di tale processo saranno: l'implementazione di numerosi altri microservizi e l'integrazione di questi all'interno del cloud con il supporto di una service mesh. Dallo studio compiuto abbiamo visto come tale cambiamento porterà vantaggi soprattutto in termini di gestione del traffico e tolleranza ai guasti, oltre che una semplificazione ed una maggiore flessibilità nello sviluppo.

Bibliografia

[1] G. P. Robert e P. J. Gerald, «Formal requirements for virtualizable third generation architectures,» *Commun. ACM,* vol. 17, n. 7, pp. 412-421, Luglio 1974.

[2] R. Morabito, «Power consumption of visualization technologies: an empirical investigation,» in *In Proceedings of the 8th International Conference on Utility and Cloud Computing (UCC '15)*, 2015.

[3] J. Gupta, «What are the pros and cons of cloud computing?,» 8 luglio 2016. [Online]. Available: https://www.znetlive.com/blog/pros-and-cons-of-cloud-computing/. [Consultato il giorno 14 luglio 2022].

[4] N. Kratzke e P.-C. Quint, «Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study,» *Journal of Systems and Software,* vol. 126, pp. 1-16, 2017.

[5] M. Yousif, «Cloud-Native Applications—The Journey Continues,» *IEEE Cloud Computing,* vol. 4, n. 5, pp. 4-5, Settebre/Ottobre 2017.

[6] A. Wiggins, «The Twelve-Factor App,» 2017. [Online]. Available: https://12factor.net/it/. [Consultato il giorno 17 luglio 2022].

[7] X. Larrucea, I. Santamaria, R. Colomo-Palacios e C. Ebert, «Microservices,» *IEEE Software,* vol. 35, n. 3, pp. 96-100, maggio/giugno 2018.

[8] J. Lewis e M. Fowler, «Microservices,» 25 marzo 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html. [Consultato il giorno 5 agosto 2022].

[9] «Cos'è il serverless computing?,» RedHat, 31 ottobre 2017. [Online]. Available: https://www.redhat.com/it/topics/cloud-native-apps/what-is-serverless. [Consultato il giorno 5 agosto 2022].

[10] A. Pereira-Vale, E. B. Fernandez, R. Monge, H. Astudillo e G. Márquez, «Security in microservice-based systems: A Multivocal literature review,» *Comput. Secur.,* vol. 103, p. 102200, 2021.

[11] S. Wee e H. Liu, «Client-Side Load Balancer Using Cloud,» in *In Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10)*, New York, 2010.

[12] S. Kumar, «Fault-tolerant patterns for Microservice,» 17 settembre 2020. [Online]. Available: https://sksonudas.medium.com/fault-tolerant-patterns-for-microservice-8d0c40f4f514. [Consultato il giorno 9 agosto 2022].

[13] N. Pubudu, «Design Patterns for Microservices — Circuit Breaker Pattern,» 13 giugno 2021. [Online]. Available: https://medium.com/nerd-for-tech/design-patterns-for-microservices-circuit-breaker-pattern-ba402a45aac2. [Consultato il giorno 9 agosto 2022].

[14] «Cos'è una service mesh?,» RedHat, 29 giugno 2018. [Online]. Available: https://www.redhat.com/it/topics/microservices/what-is-a-service-mesh. [Consultato il giorno 12 agosto 2022].

[15] «Cos'è Istio?,» RedHat, 8 gennaio 2019. [Online]. Available: https://www.redhat.com/it/topics/microservices/what-is-istio. [Consultato il giorno 12 agosto 2022].

[16] «Che cos'è Java Spring Boot?,» Azure, 2022. [Online]. Available: https://azure.microsoft.com/it-it/resources/cloud-computing-dictionary/what-is-java-spring-boot/. [Consultato il giorno 22 agosto 2022].

[17] «Spring Security Architecture,» Spring, 2022. [Online]. Available: https://spring.io/guides/topicals/spring-security-architecture. [Consultato il giorno 22 agosto 2022].

[18] «Cos’è Maven e come si usa,» Nextre engineering, 11 settembre 2020. [Online]. Available: https://www.nextre.it/cose-maven-si-usa/. [Consultato il giorno 22 agosto 2022].

[19] «Cos'è un'API REST?,» RedHat, 8 maggio 2020. [Online]. Available: https://www.redhat.com/it/topics/api/what-is-a-rest-api. [Consultato il giorno 22 agosto 2022].

[20] F. Golfarelli, «JWT — Incrementiamo la sicurezza con i JSON Web Tokens,» 30 ottobre 2018. [Online]. Available: https://medium.com/@fabiogolfarelli/jwt-incrementiamo-la-sicurezza-con-i-json-web-tokens-cd0f2f9880da. [Consultato il giorno 22 agosto 2022].

[21] M. Anicas, «An Introduction to OAuth 2,» 21 luglio 2021. [Online]. Available: https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2. [Consultato il giorno 22 agosto 2022].

[22] «OAuth 2.0,» Swagger, 2022. [Online]. Available: https://swagger.io/docs/specification/authentication/oauth2/#:~:text=OAuth%202.0%20is%20an%20authorization,Facebook%20APIs%20notably%20use%20it.. [Consultato il giorno 22 agosto 2022].