Report finale - Traefik

Federico Mazzini
federico.mazzini3@studio.unibo.it

2021

Contents

1 Piano di lavoro

La prima parte del progetto sarà focalizzata sulla comprensione delle funzionalità di Traefik dal punto di vista teorico della funzionalità stessa, con particolare riferimento ad uno scenario applicativo web-based distribuito.

Ci si concentrerà poi su come queste funzionalità sono implementate da Traefik nella pratica con esempi e attraverso il setup e lo sviluppo di real-world scenarios. Per fare ciò si utilizzerà Docker e un orchestratore (Docker Swarm o Kubernetes). Durante il progetto quindi, si cercherà di familiarizzare anche con queste tecnologie e le conseguenti integrazioni con Traefik.

Verrà testata l'affidabilità della tecnologia con riguardo alla scalabilità e alla fault-tolerance, Si cercherà di capire inoltre, cosa differenzia Traefik da altri reverse proxy attualmente in uso, quando convenga utilizzare la tecnologia e quando invece no, riportando i pregi ed i difetti dell'applicativo.

2 Introduzione a Traefik

Traefik non è un singolo servizio ma un insieme di tecnologie, nate semplicemente con la funzione di reverse proxy e poi estesesi fino a comprendere quattro principali servizi: Traefik Proxy, Traefik Pilot, Traefik Mesh e Traefik Enterprise.

- Traefik Proxy è essenzialmente un reverse proxy in ambienti microservice-based, in grado di integrarsi con i principali orchestratori di container, quali **Docker Swarm** e **Kubernetes** al fine di sviluppare web services scalabili e dinamici. Il suo utilizzo come **reverse proxy** consente di esporre solamente un punto dell'applicazione all'esterno, il quale rimarrà in ascolto delle richieste di vari client ed eseguirà routing poi ad uno o più nodi della rete che saranno in grado di soddisfarle, sulla base di specifiche regole di routing. Traefik inoltre si pone come **ingress controller** per Kubernetes e **load balancer** tra i vari nodi della rete o cluster aprendo anche alla possibilità di svolgere **canary** e **blue-green deployment**. Traefik inoltre supporta nativamente i maggiori protocolli di comunicazione e **Let's Encrypt**, una tecnologia che consente di ottenere certificati SSL gratuiti al bisogno.

- Traefik Pilot è invece un interfaccia, prevalentemente grafica, la quale permette la visualizzazione e la gestione di tutte le istanze di Traefik (generalmente containerizzate). Traefik pilot offre quindi un unico punto di accesso per consultare le metriche e monitorare la rete.

- Traefik mesh offre invece un servizio di mesh all'interno di un cluster. Un service mesh è un layer di comunicazione posto all'interno di un cluster, che permette la gestione e la comunicazione tra i diversi servizi che compongono un cluster.

2.1 Reverse proxy e load balancer

Traefik è un reverse proxy open source sviluppato in Go. Di seguito sono riportati i vantaggi di un qualsiasi reverse proxy sul mercato, con riguardo a quello che permette di fare la tecnologia presa in esame. Un **reverse proxy** è essenzialmente un layer posto come unico punto di ingresso di un servizio web, il quale intercetta le richieste da parte di vari client e le reindirizza verso un server di backend. Un reserve proxy si rivela utile in tutte quelle situazioni in cui non si vuole esporre direttamente un servizio, o quando si vuole mantenere un unico punto di accesso a fronte di diverse macchine o servizi di backend. Avere un unico punto di accesso ad un servizio permette di avere la completa gestione delle richieste e poterle sottoporle a specifiche macchine o servizi per bilanciare il carico di lavoro (load balancing).

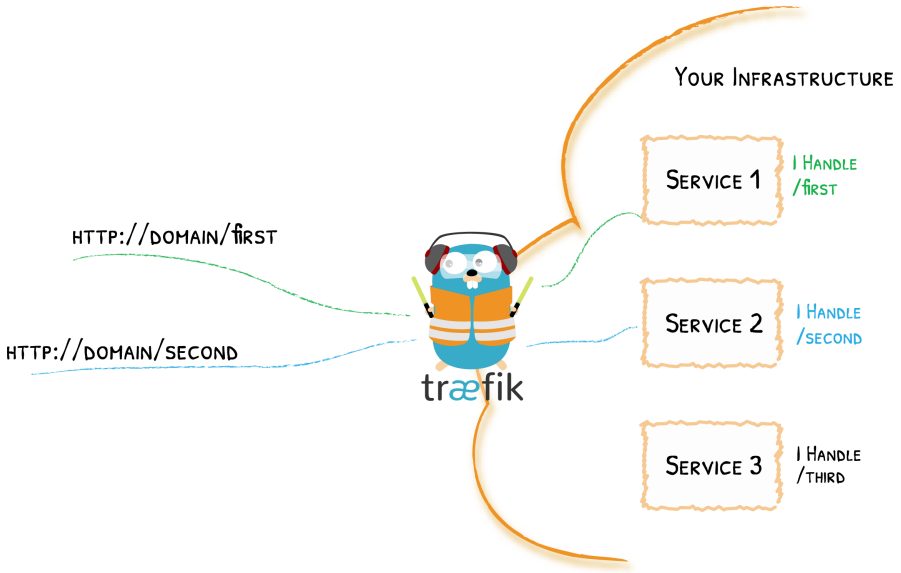


Figure 1: Traefik reverse proxy

Traefik svolge quindi anche il lavoro di **load balancer** garantendo la distribuzione delle richieste dei client tra i vari server, garantisce la disponibilità (availability) del servizio inoltrando le richieste ai soli server disponibili e permette la flessibilità di poter aggiungere o togliere server al bisogno senza bisogno di modificare l'entry point. Il load balancer di Traefik supporta per ora solamente l'algoritmo round robin, ovvero ogni richiesta viene reindirizzata ad un server differente in maniera ciclica e ripetitiva, a rotazione.

Porsi come punto di ingresso fa si che un reverse proxy possa, e in quanto utile allo sviluppo di servizi web debba, garantire sicurezza nella comunicazione client-server. In Traefik sono abilitati a questo proposito i protocolli **HTTPS** e **TLS**. TLS è essenzialmente un certificato rilasciato da terze parti (certification authorities) per garantire l'autenticità e la sicurezza di un particolare servizio sul web. E' nativo anche il supporto a **Let's Encrypt**, attraverso il quale è possibile, se non si possiede un certificato TLS,

averne uno generato al momento della durata di 90 giorni.

Un' ultima caratteristica che un reverse proxy/load balancer permette di avere è la possibilità di avere diverse strategie di deployment, Traefik permette di svolgere blue-green deployment e canary deployment. **Blue-Green deployment** è un tipo di sviluppo in cui si mantiene un ambiente di produzione stabile, a cui arrivano tutte le richieste della rete, e un identico ambiente su cui viene effettuato lo sviluppo. Una volta che il secondo ambiente ha superato tutti i test, avviene lo switch ed il traffico è dirottato totalmente su di esso. **Canary deployment** invece vede la nuova versione del servizio già disponibile, ma solo per una piccola percentuale delle richieste. In questo caso il load balancer è responsabile di reindirizzare una parte delle richieste al backend principale (la maggior parte), mentre il resto delle richieste al backend con le nuove funzionalità.

Ciò che dovrebbe contraddistinguere Traefik da altri reverse proxy, ad esempio nginx o HAproxy, è la maggior flessibilità e dinamicità. La dinamicità di Traefik è dovuta a parte della configurazione del proxy, definita come dinamica, modificabile in run-time senza dover riavviare il servizio. Esso comunica mediante API con uno o più orchestratori a cui è associato e resta in attesa di modifiche, come ad esempio la creazione o l'eliminazione di un container.

2.2 Architettura di Traefik

Di seguito viene descritta l'architettura generale di Traefik e ne vengono riportati esempi di configurazione in YAML per evidenziare meglio i concetti.

1. **Providers**, componenti con cui Traefik dialoga per ricavare automaticamente diverse regole di routing.
2. **Configuration**, si divide in statica e dinamica.
3. **Entrypoints**, essenzialmente il punto di entrata al servizio che si sta sviluppando.
4. **Services**, intermediari verso i servizi backend dell'applicativo.
5. **Routers**, il collegamento tra entrypoint e service.
6. **Middlewares**, feature aggiuntive.

2.2.1 Providers

Un provider in Traefik è un componente di un'infrastruttura esistente, con cui il servizio di Traefik comunica, attraverso API dei provider, al fine di ricavare informazioni riguardo al routing delle richieste. Il fatto di ricavare il routing tramite le API dei singoli provider è ciò che permette a Traefik di rilevare modifiche ai nodi della rete automaticamente e rilevare malfunzionamenti ai server, aggiornando il routing di conseguenza. I provider supportati da Traefik, sono essenzialmente di tre tipi:

- **Orchestratori**
  - Docker

  - Kubernetes
  - Consul Catalog
  - ECS
  - Marathon
  - Rancher
- **Key-value**
  - Consul
  - Etcd
  - ZooKeeper
  - Redis
- **Manuali**
  - File
  - Http

In questa prima parte introduttiva, viene utilizzato il File, per evitare di complicare la teoria con nozioni Docker o Kubernets.

2.2.2 Configuration

La configurazione di Traefik avviene in due step, definiti configurazione statica e configurazione dinamica.

La **configurazione statica** o startup configuration è da specificare necessariamente per avviare il servizio in quanto vengono settati i parametri iniziali, quali i Provider che si desidera utilizzare e gli EntryPoint, essenzialmente le porte di ascolto di Traefik. La specifica della configurazione può avvenire in tre modi diversi, tramite file di configurazione in linguaggio YAML o TOML, linea di comando o variabili d'ambiente.

La **configurazione dinamica** è invece una configurazione modificabile a runtime dai Provider. Essa comprende tutto ciò che riguarda Services, Routers e Middlewares. Modificare uno di questi componenti comporterà la modifica della configurazione dinamica di Traefik ma non quella statica, permettendo al servizio di rimanere disponibile e non essere riavviato.

In figura si può vedere come le componenti di Traefik siano soggette alla specifica o alle modifiche. Infatti a sinistra con la configurazione statica si specificano EntryPoint e Provider, mentre a destra tramite chiamate API ai provider si specificano e modificano Services, Routers e Middleware.

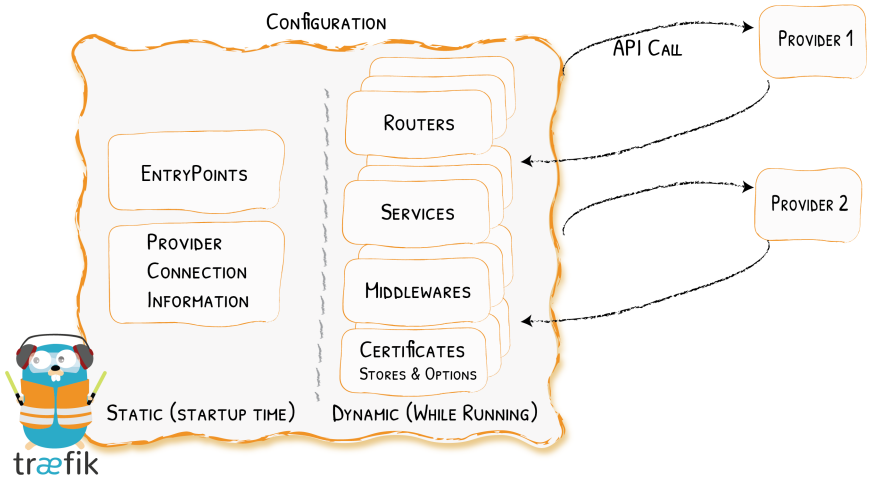


Figure 2: La configurazione statica e dinamica in Traefik

2.2.3 EntryPoints

Un EntryPoint in Traefik è essenzialmente un punto di accesso esterno alla rete, è ciò che viene esposto all'esterno a cui i client possono inviare richieste. Come già detto è necessario specificarli durante la configurazione statica in modo tale che il servizio possa rimanere in ascolto delle richieste, che verranno poi girate attraverso regole di routing ai servizi che si occuperanno di soddisfarle.

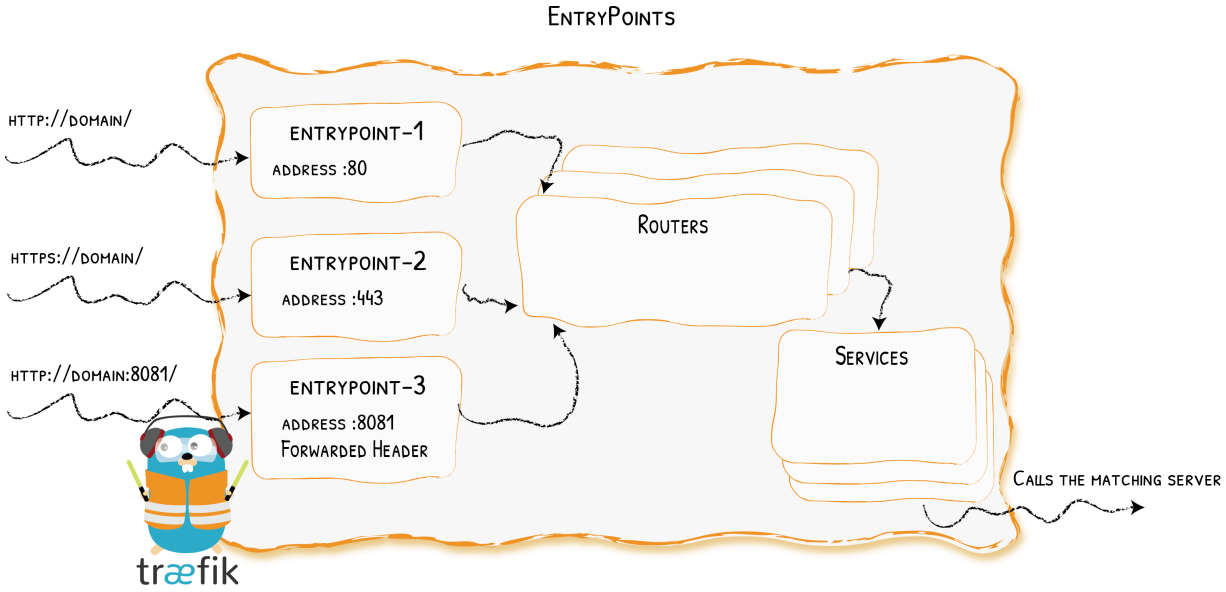


Figure 3: Entrypoint

2.2.4 Services

I services non sono le macchine di backend finali, le quali riescono effettivamente a soddisfare le richieste inoltrate dagli utenti, bensì un ultimo layer prima di giungere ad essi. I services sono una sorte di contenitore per i vari server e sono responsabili di inoltrare le richieste ai server. E' a questo livello il punto in cui viene effettuato il load balancing. Ogni Service può contenere un numero arbitrario di server, quando una richiesta sopraggiunge questa è inoltrata ad uno dei server, la decisione come detto in precedenza, è guidata dall'algoritmo round robin.

E' possibile utilizzare sessioni persistenti (sticky session) attraverso le quali un client viene associato ad un determinato server per l'intera sessione. Le sue richieste perciò verranno indirizzate sempre verso lo stesso server.

Inoltre è possibile monitorare i Services attraverso health check periodici, tramite i quali il servizio controlla se i vari server dietro il load balancer non sono compromessi.

Traefik non è un web server, ovviamente i server sono indipendenti da esso e devono essere avviati in maniera esterna ad esso.

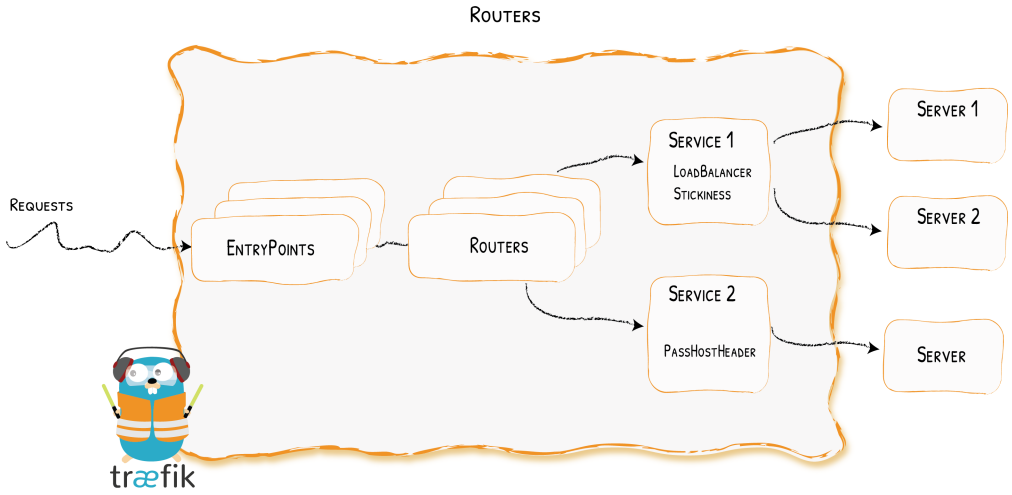


Figure 4: Services

Load balancing in un Service

Si fa il caso ad esempio di 3 web server in ascolto sulle prime porte 8000 tramite protocollo http. Nel file di configurazione dinamica è necessario specificare il protocollo che si sta utilizzando, il nome del servizio e gli indirizzi dei server che lo compongono.

```
#Specifica dei Services
http:
    services:
        my-service-1:
```

```
        loadBalancer:
            servers:
                #I vari server contenuti nel Service
                - url "http://localhost:8000/"
                - url "http://localhost:8001/"
                - url "http://localhost:8002/"
```

Load balancing tra Services

Oltre alle operazioni tra server dello stesso Service, è possibile specificare operazioni tra Service stessi. E' il caso in cui si hanno diversi Service che concorrono a richieste simili. Tra loro è possibile attuare strategie quali weighted round robin e mirroring di richieste, utili per sviluppare soluzioni di sviluppo Canary o Blue-Green. La keyword weight all'interno della configurazione dei vari service nel file YAML è quello che va a determinare l'effettiva distribuzione delle richieste.

In questo caso tutte le richieste verso *service-wrr* (al quale sarà associato un indirizzo attraverso un Router enunciato successivamente) verranno reindirizzate per la maggior parte a *service-1* rispetto che a *service-2*.

```
#Specifica di Service multipli con weighted round robin
http:
    services:
        service-wrr:
            weighted:
                services:
                - name: service-1
                weight: 3
                - name: service-2
                weight: 1
        service-1:
            loadBalancer:
                servers:
                - url: "http://localhost:8001/"
                - url: "http://localhost:8002/"
        service-2:
            loadBalancer:
                servers:
                - url: "http://localhost:8003/"
                - url: "http://localhost:8004/"
```

2.2.5 Routers

Non tutte le richieste possono, o devono, essere eseguite dallo stesso Service. I Routers si occupano di indirizzare le varie richieste dei client, dagli Entrypoint del sistema ad un Service che possa soddisfarle. Esse sono sostanzialmente regole di routing e in quanto tali si basano sull'URI della richiesta, sui metodi GET, POST, PUT, DELETE di essa, sugli Header e sulla base di epressioni regolari . Quando una richiesta fa match con un Router, questa verrà indirizzata al Service corrispondente. In questo semplice caso viene creato un solo Services e di conseguenza un Routing in ascolto sugli Entrypoint.

```
#Dynamic configuration File
http:
    #Specifica dei Routers
    routers:
        my-router-1:
            rule: "Host('localhost')"
            service: my-service-1
```

Di default un Router ascolta su tutti gli entrypoint, ma è un dettaglio modificabile durante la configurazione statica, specificando solo alcuni degli Entrypoints.

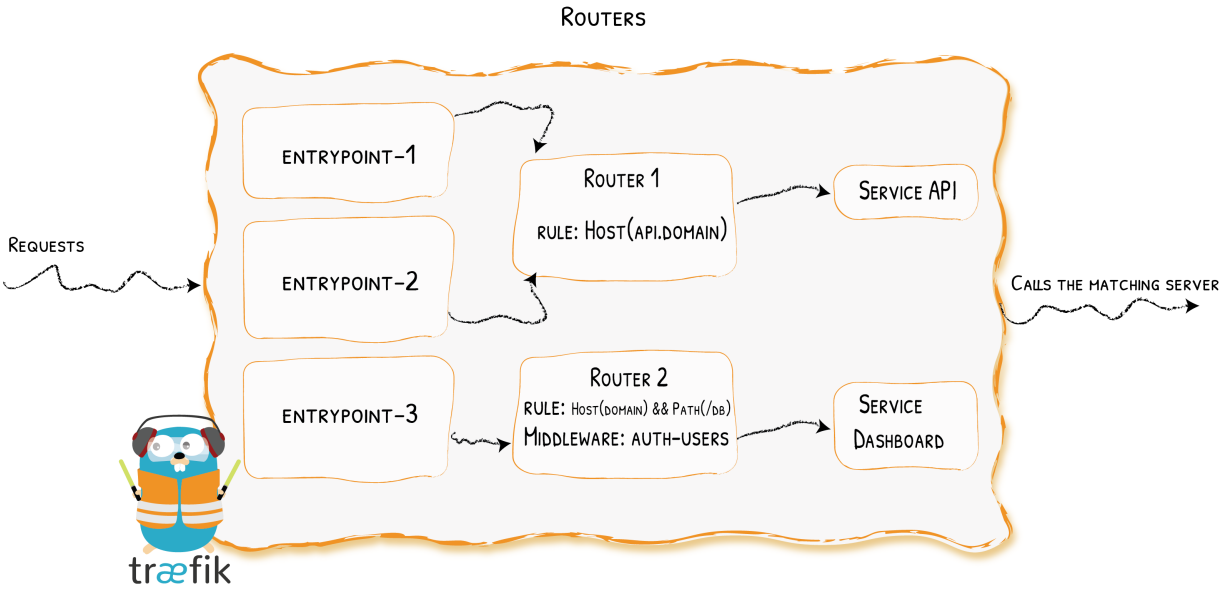


Figure 5: Routers

Un Router può essere integrato da componenti aggiuntivi, definiti Middleware, descritti nella prossima sezione.

2.2.6 Middlewares

Un Middleware in Traefik è un estensione della request del client o della response del Service. Alcuni Middleware quindi si applicano prima che la richiesta venga inoltrata al Service, come ad esempio un'autenticazione esterna, mentre altri vengono applicati successivamente alla risposta. Alcuni tra i Middleware più utili sono quelli per l'autenticazione, in questo caso prima che la richiesta venga indirizzata verso il Service, si passa per un servizio di autenticazione interna con il Middleware *BasicAuth*, mentre per un servizio di autenticazione esterna con *ForwardAuth*. Questi due Middleware sono esempi di una lista esaustiva disponibile all'interno della documentazione di Traefik. In ogni caso, l'estensione Middleware è applicata ad uno o più Router, specificando nella sua dichiarazione il tipo di Middleware che si vuole utilizzare.

```
#Middleware di redirect https
http:
    middleware:
        redirect-https:
            redirectScheme:
                scheme:https
                permanent:true
                port: "443"
```

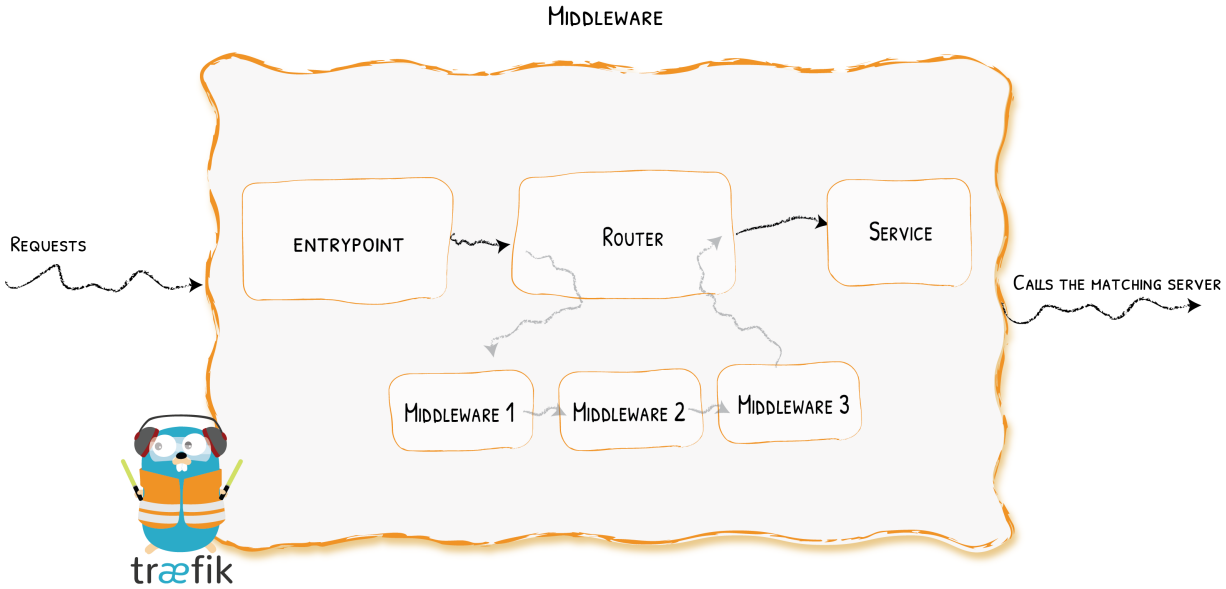


Figure 6: Middlewares

2.2.7 Certificati TLS e Let's Encrypt

TLS (Transport Layer Security) è un protocollo per la trasmissione di dati in maniera criptata e sicura tra due sistemi su internet, successore del protocollo SSL. TLS permette di utilizzare il protocollo HTTPS certificando la sicurezza della connessione e l'autenticità di chi sta fornendo il servizio. E' buona prassi, per qualsiasi servizio web si crei, garantire una connessione client-server mediante protocollo HTTPS. Per farlo è necessario dotarsi di un certificato digitale, o certificato TLS, rilasciato da una certification authority (CA). Un certificato digitale non è altro che un file contenente una chiave pubblica dimostrante l'autorità di chi sta fornendo il servizio e che da inizio ad una sessione di comunicazione criptata attraverso crittografia asimmetrica.

Let's Encrypt Let's encrypt è una Certificate Authorithy che rilascia gratuitamente certificati TLS della durata di 90 giorni. La procedura di richiesta è molto snella e veloce, necessità solo di un indirizzo email valido e un nome di dominio correttamente detenuto. Il certificato è rilasciato al momento della richiesta e viene automaticamente rinnovato dopo 90 giorni.

Traefik supporta nativamente la richiesta di certificati e il rinnovo automatico tramite Let's Encrypt, ma è comunque possibile utilizzare qualsiasi altro certificato TLS ottenuto da altre Certificate Authorities.

2.3 Traefik Pilot

Traefik Pilot è una piattaforma di controllo per una o più istanze di Traefik Proxy. Attraverso la sua interfaccia è possibile visualizzare alcune metriche riguardanti lo stato della rete, le richieste e le connessioni. E' inoltre possibile ricevere alert dal Pilot riguardo a problemi di sicurezza o guasti all'interno del servizio.

Per attivare il Pilot è necessario creare un account apposito al seguente indirizzo. Sarà poi necessario creare un nuovo token per la configurazione tra Proxy e Pilot registrando una nuova istanza. Il token andrà poi inserito nella configurazione statica del Proxy in maniera seguente

```
pilot:
    token: "token-here"
```

Il vantaggio del Pilot è quello di essere accessibile da internet, previa autenticazione, e poter inviare notifiche ad un utente rispetto ad eventi che accadono. Il servizio è esterno e indipendente da una o più istanze di Traefik. Dal momento che un'istanza è registrata al proprio Pilot, questa invierà periodicamente aggiornamenti al servizio, il quale aggiornerà le metriche. Diventa fondamentale quindi che la macchina, o il container su cui è istanziato Traefik, abbia correttamente accesso alla rete.

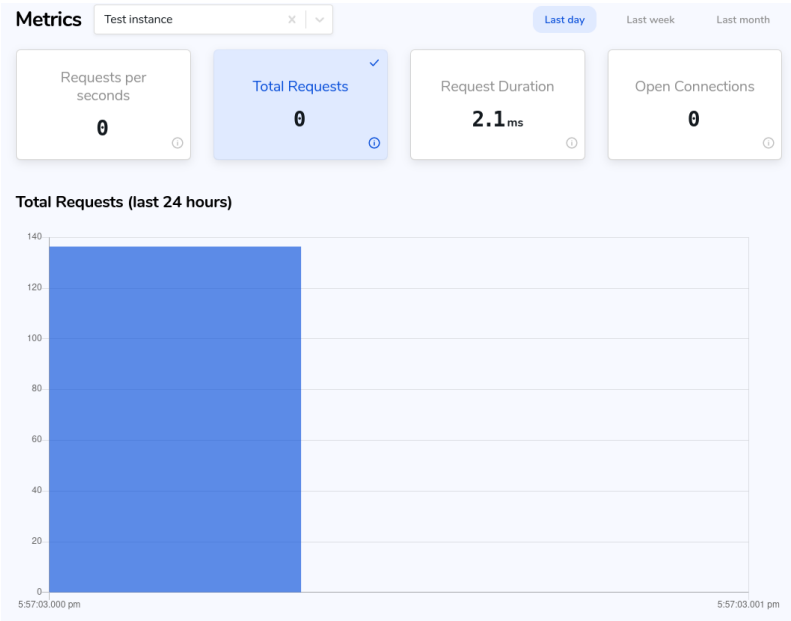


Figure 7: L'interfaccia di Traefik Pilot

L'interfaccia di Traefik Pilot raccoglie informazioni essenziali ma non sufficienti per un monitoraggio approfondito di Traefik. Inoltre si è riscontrata una latenza significativa durante l'utilizzo. I valori ed i grafici spesso differiscono, come nel caso dell'immagine, in cui le richieste totali risultano 0 ma nel grafico vengono riportate (correttamente perchè era stato generato del traffico sul servizio)

2.4 Traefik Mesh

Un servizio di mesh è un network layer configurabile all'interno di un cluster per la gestione delle comunicazioni tra servizi ed applicazioni. Un servizio di mesh aiuta ed automatizza un processo che altrimenti dovrebbe essere gestito dai servizi stessi. Permette quindi di separare la logica dell'applicazione dalle regole di network e la sicurezza della comunicazione.

In particolare Traefik Mesh è un servizio di mesh che permette la gestione e la visibilità delle comunicazioni all'interno di un cluster Kubernetes. Si basa su Traefik Proxy e ne condivide parte delle funzionalità, a partire dalla gestione del traffico e bilanciamento del carico tra i servizi. L'utilizzo di questa mesh all'interno del cluster prevede l'utilizzo di un servizio separato dal resto (pod), che quindi è indipendente dal resto dei servizi già offerti dal cluster. Traefik Mesh dispone inoltre di una dashboard di controllo, la stessa che si ritrova anche nell'utilizzo di Proxy e supporta tool quali Prometheus e Grafana per la consultazione delle metriche.

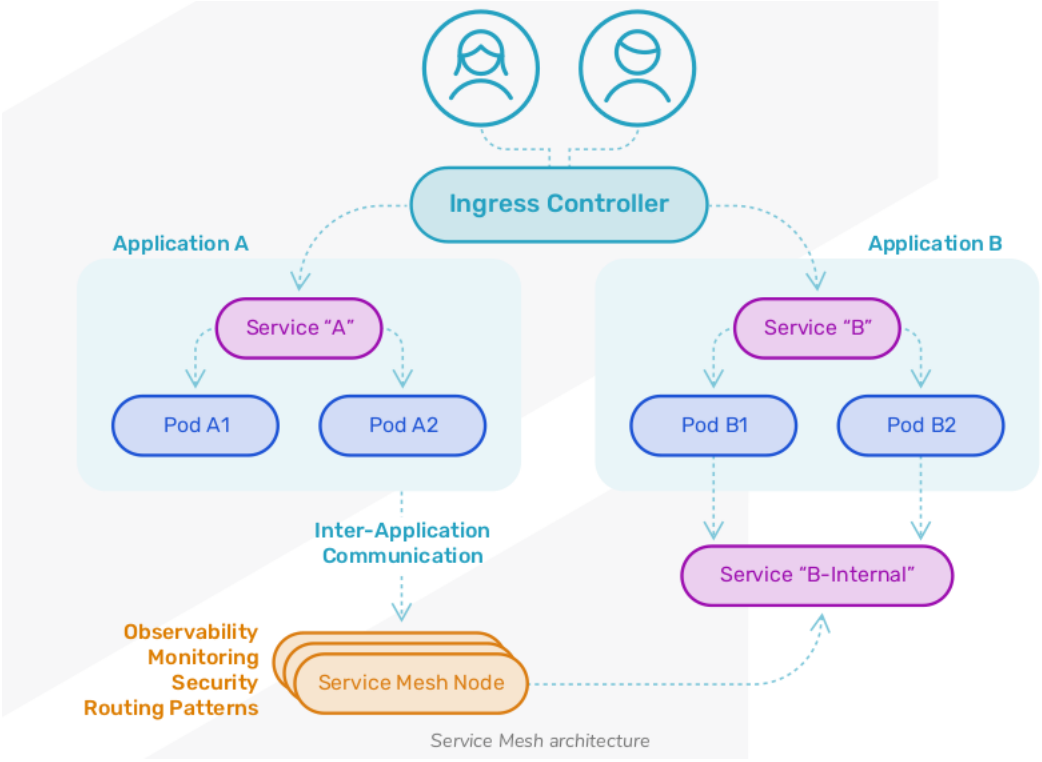


Figure 8: Architettura con un servizio di mesh

Come si può verificare in figura, un service mesh è un componente del cluster, esterno ai servizi che compono la logica dell'applicazione, che ha il compito della sola gestione della comunicazione intra cluster.

3 Introduzione a Docker

Docker è una piattaforma per lo sviluppo, l'esecuzione e il rilascio di applicazioni. Attraverso esso è possibile racchiudere un'applicazione, o un servizio, all'interno di componenti standardizzati definiti container al fine di rendere indipendente il servizio sviluppato dal suo ambiente di esecuzione. Questi container sono la base, l'unità minima, con cui poter definire applicazioni moderne costituite da una moltitudine di essi, definita architettura a microservizi. Docker offre inoltre una serie di tool ed una piattaforma per gestire il ciclo di vita dei container, dalla creazione all'eventuale rimozione.

3.1 Concetti fondamentali

Docker Engine

Il Docker Engine è il layer sul quale Docker viene eseguito. E' un ambiente leggero per il rilascio e la gestione di container, immagini, servizi. Nativamente è eseguito in ambiente Linux ed è costituito da tre parti principali:

- Un Docker Daemon eseguito nella macchina host.
- Un client Docker che comunica con il daemon per eseguire comandi .
- Un'API REST per interagire con il Docker Daemon da remoto.

Docker Client

Il client di Docker è ciò con cui lo sviluppatore, o comunque l'utente che sta utilizzando Docker, si interfaccia. Questo comunica con il demone Docker per l'esecuzione di istruzioni.

Docker Daemon

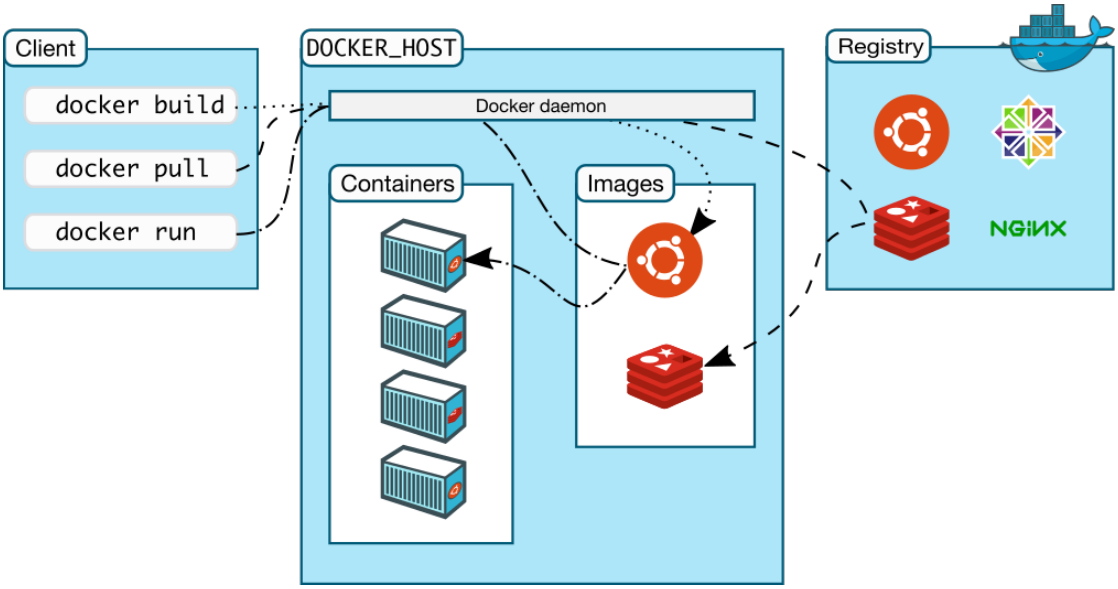
Il Docker Daemon è il servizio che effettivamente esegue i comandi richiesti dal client, come la build o il run di un container. Il demone Docker viene eseguito sulla macchina ospitante, ma non è possibile comunicarvi.

Docker Image

Una image, o immagine, Docker è una sorta di template predefinito di un container e contiene le istruzioni per definirlo all'interno di un Dockerfile. Solitamente un'immagine Docker si basa su un'altra immagine e aggiunge poi dei layer superiori atti a creare la nuova immagine. Questo meccanismo di layering nelle immagini render Docker molto leggero e veloce. Un'immagine può poi essere istanziata dando orgine ad un container.

Docker Container

Un container, come già precedentemente anticipato, è un'istanza di un immagine Docker. La sua creazione, così come l'esecuzione ed il relativo stop viene eseguito tramite API o linea di comando. Un container può essere creato attraverso CLI specificando l'immagine da istanziare, mediante il comando *docker run*. Fondamentale in questo frangente



potrebbe essere specificare altre istruzioni per la creazione come ad esempio il comando -d per specificare di eseguire il container in background nel Docker Engine. Un esempio per la creazione di un semplice web server nginx.

```
sys@admin:~$ docker run -d -p 8000:80 nginx
Unable to find image 'nginx:latest' locally
latest: Pulling from library/nginx
45b42c59be33: Pull complete
...
83500d851118: Pull complete
52333e4bb610191515449a2423f31a67d014a886db3a4ecd68d02e2ff4cf8a32
```

-d viene utilizzato per il motivo discusso sopra, mentre *-p 8000:80* serve per esporre il server esternamente da Docker così da accedervi. Un container infatti possiede un'interfaccia di rete per la comunicazione con l'host ed un proprio indirizzo IP che, di default, è accessibile solamente all'interno del Docker Engine. Per far si che un container sia accessibile anche esternamente dal Docker host, sarà necessario eseguire un mapping tra le porte.

Volumi

I volumi sono il modo in cui i dati all'interno di un container sono resi persistenti. Senza uno o più volumi, qualsiasi dato all'interno di un container attivo che viene rimosso non è persistente. Per far si che i dati all'interno persistano è necessario mappare una directory all'esterno del container con una all'interno, un concetto simile al port mapping per il networking.

3.2 Docker compose

Compose è una procedura Docker che permette di definire ed eseguire applicazioni costituite da più container. La configurazione di Compose avviene tramite file in formato YAML e di default ha nome *docker-compose.yml*. All'interno del file si inseriscono le informazioni utili alla creazione dei vari container che compongono l'applicazione come image, porte, volumi, cosa che avveniva prima a linea di comando. Attraverso Compose quindi è possibile istanziare un ambiente costituito da più container che potenzialmente comunicano tra loro. Un' altro grande vantaggio è il fatto che la configurazione è modificale e, di conseguenza, verranno riavviati solamente i container modificati. Un esempio di struttura per il *docker-compose* file è il seguente

```
version: "3.7"

services:
  example-app:
    image: example-app
    container_name: myapp
    ports:
        -5000:80
  db:
    image: example_db
    #file di configurazione
    #volumi
    volumes:
        - database-data:/var/lib/data
```

L'esempio vuole solo spiegare come può essere definito un file docker-compose. Per eseguire l'ambiente configurato tramite compose è necessario utilizzare il comando

```
sys@admin:~$ docker-compose up -d
```

Non è necessario specificare il nome del file perchè di default Docker cerca un file denominato *docker-compose.yml*, ma è comunque possibile specificare un altro nome.

3.3 Container orchestration

La gestione ed il mantenimento di applicazioni containerizzate, con riguardo a scenari distribuiti con necessità di scalabilità avviene solitamente mediante un orchestratore. La portabilità e la riproducibilità di processi containerizzati danno l'opportunità per muovere e scalare queste applicazioni attraverso cloud e datacenter. I container effettivamente permettono, attraverso la loro separazione dall'ambiente di esecuzione, di girare dovunque. A ragion di ciò, sarebbe molto utile avere, nel caso si avesse la necessità di scalare l'applicazione, qualche processo per automatizzare il mantenimento dell'applicazione in grado di rimpiazzare container difettosi, gestire updates e la riconfigurazione.

I tool per la gestione, il mantenimento e l'automatizzazione di container sono chiamati orchestratori. I più comuni orchestratori sono Docker Swarm e Kubernetes. La container orchestration quindi, consiste in un insieme di tool che aiutano a gestire container in un ambiente di produzione. Tipicamente le soluzioni di orchestrazione contano multipli Docker host, ovvero macchine, virtuali o fisiche, le quali possiedono una propria istanza di Docker Engine.

3.3.1 Docker Swarm

Docker Swarm è la soluzione di Docker per l'orchestrazione. Attraverso Docker Swarm è possibile combinare Docker machine in un singolo cluster, in cui Swarm si occuperà della distribuzione dei servizi e delle istanze in host separati per high availability e per il load balancing.

Nodi

In Swarm, un nodo del cluster è definito manager, mentre gli altri nodi sono definiti workers. I nodi manager sono i responsabili per lo stato del cluster e la schedulazione dei service. I nodi worker invece hanno il compito di eseguire container e servizi come istruzioni dai manager. Per utilizzare swarm è necessario inizializzarlo. Per farlo è necessario utilizzare il comando *docker swarm init*

```
sys@admin:~$ docker swarm init
Swarm initialized: current node (xq49o1ut6wug0duphsjwhm97e) is now a manager.

To add a worker to this swarm, run the following command:
    docker swarm join --token SWMTKN-1-4p3jn9hmhzs0wuiwv8yfrcv40bq
    wl9wwdk8z8vdpwpl6zpxlif-3uvmknwt7zzo8owq2f66hjnqy 192.168.1.14:2377

To add a manager to this swarm, run 'docker swarm join-token manager'
and follow the instructions.
```

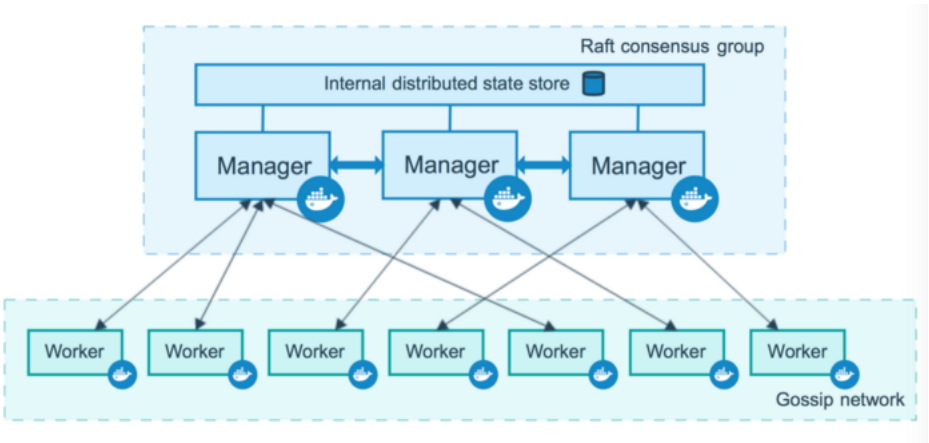
I nodi workers possono unirsi al cluster mediante lo specifico token ricevuto.

Service

Al di fuori di un cluster swarm, si è fatto riferimento ai container come elementi isolati, i quali erano istanziati mediante il comando *docker run image*. In Swarm invece ci si riferisce ai service, i quali sono istanziati, e replicati, attraverso i nodi manager del cluster. Quando un service viene creato all'interno di Swarm viene definito lo "stato" ideale del servizio, numero di replice, risorse, porte e altri dettagli. Docker cercherà di mantenere lo stato desiderato gestendo malfunzionamenti sui nodi, perdita di connessione e cercherà di bilanciare i servizi.

Stack

Docker Stack è un'estensione di Docker-compose e con esso condivide i file *docker-compose.yml* e la relativa struttura, ma si applica nello scenario di uno swarm per



istanziare servizi, gestirli e controllarli. A partire quindi da un file *yaml* uno stack è creato attraverso il comando

```
sys@admin:~$ docker stack deploy -c docker-compose.yml traefik

Creating network traefik_default
Creating service traefik_traefik
Creating service traefik_nginx
```

dove "-c docker-compose.yml" specifica il nome del file e "traefik" il nome dato allo stack. In questo caso è stato creata una custom network e due servizi, che verranno descritti nella sezione successiva rigurdante l'utilizzo di Traefik.

E' possibile verificare i servizi attivi all'interno di uno stack mediante

```
sys@admin:~$ docker stack services traefik
ID             NAME              MODE         REPLICAS   IMAGE
wty4i9d8irqn   traefik_nginx     replicated   1/1        nginxdemos/hello:latest
bjel02kuktmx   traefik_traefik   replicated   1/1        traefik:latest
```

3.3.2 Kubernetes

Kubernetes è il framework per l'orchestrazione di container più utilizzato. Sviluppato inizialmente da Google è stato poi reso un progetto open-source a partire dal 2014. Esso condivide con Swarm diverse funzionalità, soprattutto per quanto riguarda funzionalità di base, ma si differenzia da quest'ultimo nel modo di operare e in una maggiore disponibilità di funzioni avanzate. Attraverso Kubernetes è possibile gestire cluster di nodi, lasciando ad esso il controllo della scalabilità dei servizi, tolleranza ai guasti, gestione del carico.

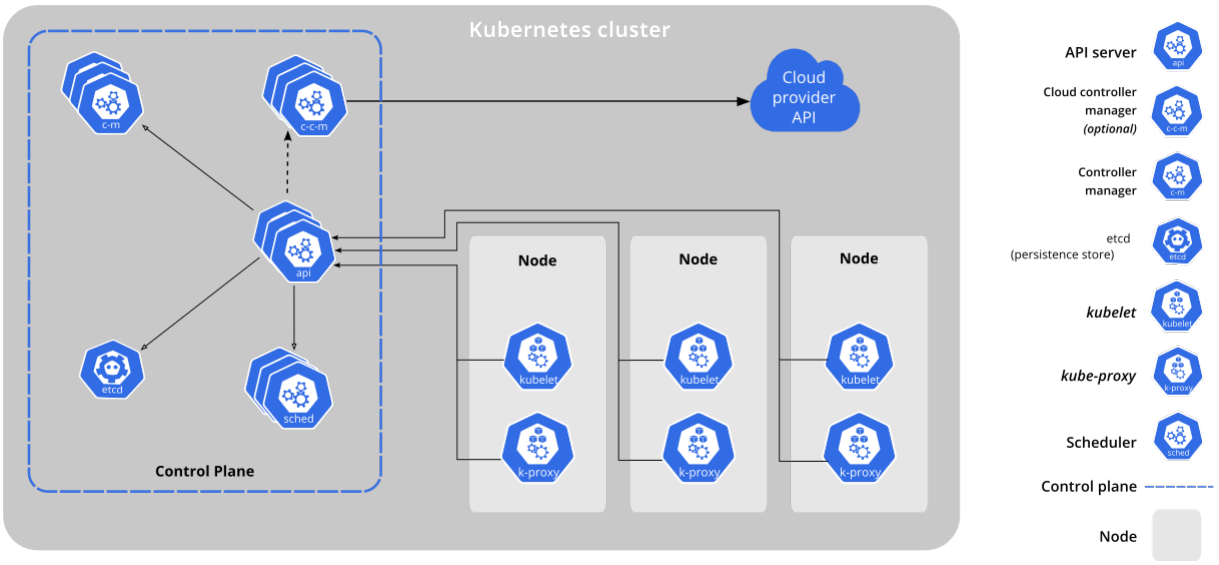
Un cluster Kubernetes è un insieme di macchine fisiche o virtuali denominate nodi. La prima distinzione possibile tra nodi, simile anche a quella effettuata in Swarm, è quella tra worker node e master node. Un **worker node** è un nodo su cui vengono eseguiti

uno o più container a comporre un applicazione o un servizio. I componenti principali di un worker node sono

- Kubelet: un servizio che gestisce i container sul nodo, si assicura che siano in esecuzione ed healthy.
- Kube-proxy: mantiene e gestisce e le regole di networking all'interno del nodo

Un **master node** rappresenta invece il responsabile del controllo all'interno del cluster, i cui componenti principali sono

- Kube-APIserver: espone le API di sistema e un' interfaccia con il panello di controllo.
- Kube-controller manager:monitora e gestisce lo stato dei nodi e dei servizi.
- Kube-scheduler: si occupa di gestire e assegnare pod di cui è stata richiesta la creazione ma che non sono state create e ancora assegnate a nessun nodo.

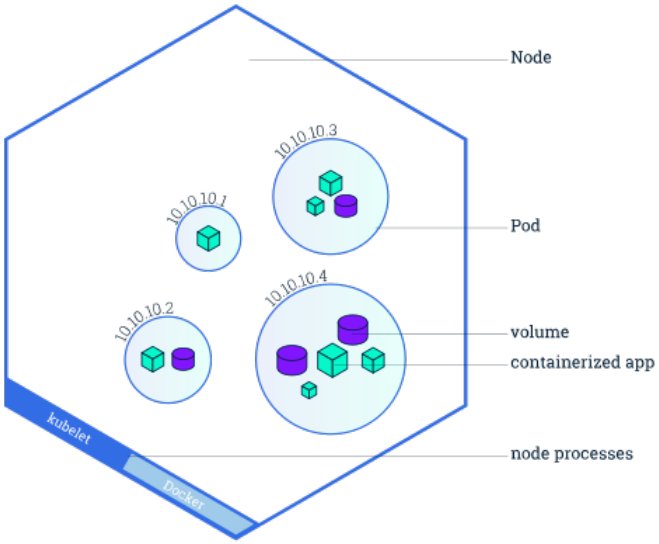


Pods

Una differenza fondamentale con Swarm è la presenza di un componente, denominato pod. Una pod è l'unità più piccola all'interno di Kubernetes e contiene al suo interno uno o più container. Se all'interno di esso sono contenuti più container, questi sono trattati come una singola entità e condividono le risorse del loro contenitore. Questa astrazione, dal container ad una pod, permette a Kubernetes di separarsi dalla tecnologia utilizzata e di non dover dialogare direttamente con un container, bensì con un componente nativo di Kubernetes stesso. Le pod sono usate come unità di replicazione all'interno di

kubernetes. E' questo componente infatti a venire scalato tra i diversi nodi del cluster in caso di un alto volume di richieste o di fallimenti tra nodi o pod stesse.

Le pod sono create e mantenute a partire da una specifica di deployment, la quale gestisce in primo luogo il numero delle repliche che dovrebbero essere mantenute attive e gestisce i casi in cui non dovesse essere così.



Service

Un service è un modo astratto per raggruppare le pod, le quali sono viste come componenti volatili all'interno di un sistema, create e distrutte frequentemente, possono incorrere in fallimenti improvvisi o creazioni al fine di far scalare l'applicazione. Un Service è un'astrazione con cui definire un set logico di pod e la policy di accesso ad esse.

Ingress

Le pod sono isolate dall'esterno, perciò la comunicazione dall'esterno di esse, senza componenti aggiuntivi non è possibile. Se si desidera comunicare con un servizio all'interno di una pod è necessario avere un canale per la comunicazione, chiamato in Kubernetes semplicemente Ingress. Un Ingress quindi è un componente che definisce il routing tra una pod e l'esterno.

Un altro componente, denominato Ingress Controller, è un servizio in ascolto davanti alla rete di un cluster e gestisce le regole di network tra l'esterno ed i vari servizi della rete, attraverso routing HTTP. L'ingress controller è un componente fondamentale all'interno di un cluster Kubernetes, tuttavia è lasciata l'implementazione da proxy di terze parti, tra cui Traefik.

4 Utilizzo di Traefik Proxy

4.1 Ambiente e strumenti

Per effettuare il testing delle funzionalità principali è stato utilizzato Docker e Docker Swarm come orchestratore.Il servizio di Traefik è sempre stato utilizzato come immagine Docker, perciò non è necessario installare l'istanza standalone di Traefik. Questa esiste ma va un po' contro l'essenza cloud-based e container-based della tecnologia in esame, per questo motivo si è scelto di non utilizzarla.

Per riprodurre questi primi esempi di utilizzo è necessario installare solamente Docker, di cui si rimanda l'istallazione al sito ufficiale.

Di seguito le tecnologie utilizzate durante i test e gli esempi di utilizzo:

- Docker-compose
- Docker Swarm
- Traefik
- NGINX
- Dynamic DNS
- Let's Encrypt
- NodeJs
- Express
- Swagger

4.2 Configurazione di Traefik

Tipi di configurazione statica

Come anticipato alla sezione precedente, la configurazione statica è specificabile in tre diversi modi. E' necessario tenere presente che uno esclude gli altri, è possibile quindi specificare la configurazione solo in una maniera alla volta. Comunque qualunque sia il metodo utilizzato, è indifferente, in termini di risultati, utilizzare uno piuttosto che un altro tipo di configurazione.

4.2.1 Configuration file

E' il metodo più utilizzato, è possibile creare file di configurazione in linguaggio YAML o TOML, il cui nome di default è traefik.toml/traefik.yml.

```
#static configuration da file
  api:
    dashboard: true
    insecure: true
```

```
log:
  level: DEBUG
providers:
  docker:
      exposedByDefault: false
entryPoints:
  web:
    address: ":80"
  websecure:
    address: ":443"
```

4.2.2 Command line

E' altresì possibile creare la configurazione statica di Traefik da linea di comando, utile se si utilizza docker-compose per avviare il servizio. Un esempio è proprio in un file docker-compose, all'interno dei "command" viene specificata la configurazione statica.

```
version: '3'
  services:
    traefik:
      image: traefik
      command:
        - "--api.insecure=true"
        - "--providers.docker"
        - "--log.level=INFO"
        - "--entrypoints.web.address=:80"
      ...
```

4.2.3 Environment variables

Utile quando si vuole istanziare Traefik in multiple istanze, perchè specificabili in un solo punto. La specifica avviene sotto la parola chiave "environment"

```
version: '3'
  services:
    traefik:
      image: traefik
      environment:
        - TRAEFIK_API_INSECURE=true
        - TRAEFIK_PROVIDERS_DOCKER=true
        - TRAEFIK_LOG_LEVEL=INFO
        - TRAEFIK_ENTRYPOINTS_WEB_ADDRESS=:80
      ...
```

4.2.4 Configurare i Provider

La specifica del provider è fondamentale nella configurazione di Traefik, in modo tale che esso possa restare in ascolto per eventuali modifiche. E' possibile specificare anche diversi provider, nell'esempio è riportato solamento Docker. Se un container espone una porta, Traefik usa quella porta per comunicare internamente con il container. Se un container invece non espone nessuna o diverse porte,allora è necessario definire la porta da usare all'interno della configurazione dinamica.

```
#Specifica dei provider
providers:
  docker:
    exposedByDefault: false
```

L'ultimo setting *exposedByDefault* non è obbligatorio, ma viene utilizzato per esporre a Traefik i container all'interno del cluster in maniera predefinita o meno. Se settato a *false*, i container che non hanno abilitato Traefik saranno ignorati da esso.

4.2.5 Configurare gli Entrypoint

Per definire un EntryPoint è necessario definire la porta e il protocollo utilizzato tra UDP e TCP secondo il seguente schema.

```
[host]:port[/tcp|/udp]
```

Un primo esempio di configurazione statica attraverso file YAML è il seguente. Si definiscono due Entrypoint, la porta 80 per HTTP e 443 per HTTPS.

```
#Specifica degli entrypoint
entrypoints:
    web:
        address: :80
    websecure:
        address: :443
```

4.3 Primo utilizzo di Traefik

In questo primo esempio di utilizzo di Traefik, verrà definita la configurazione statica di esso mediante file, come visto in precedenza. Ci si avvarrà poi di un file *docker-compose.yml* per la creazione dei servizi veri e propri, i quali saranno composti da un'istanza di Traefik ed un server nginx.

4.3.1 Configurazione statica

Definisco la configurazione statica mediante il file traefik.yml. Il file è l'insieme di tutti gli elementi configurati precedentemente singolarmente, dashboard, log, entrypoints e providers.

```
api:
  dashboard: true
  insecure: true

log:
  level: DEBUG

entryPoints:
  web:
    address: ":80"
  websecure:
    address: ":443"

providers:
  docker:
    exposedByDefault: false
```

4.3.2 Configurazione dinamica

Dopo aver definito in un file la configurazione statica, definisco all'interno di un file docker-compose i servizi che saranno creati, ovvero un istanza di Traefik e un server d'esempio nginx.

```
version: '3'
services:
  #istanza di traefik
  traefik:
    image: traefik
    ports:
      - "80:80"
      - "443:443"
      #porta per la dashboard
      - "8080:8080"
    volumes:
      #monto file socket di docker così che possa utilizzare le api del provider
      - /var/run/docker.sock:/var/run/docker.sock
      #monto il file di configurazione statica
      - ./traefik.yml:/etc/traefik/traefik.yml
#server nginx
  nginx:
     image: nginxdemos/hello
     labels:
      - "traefik.enable=true"
      - "traefik.http.services.myservice.loadbalancer.server.port=80"
```

```
- "traefik.http.routers.myroute.service=myservice"
- "traefik.http.routers.myroute.rule=Host(`demo.localhost`)"
```

4.3.3 Analisi delle labels

Per esporre a Traefik i servizi definiti nel file *docker-compose* all'interno del cluster si utilizzano le labels, di seguito analizzate.

Nella prima si associa nginx ad un Service, definito con il nome *myservice*, il quale possiede un load balancer interno utile nel caso ci fossero più istanze dello stesso container. Inoltre viene esplicitata la porta di accesso per le richieste da indirizzare al server. In questo caso, essendo nginx un server web, è stata esposta la porta 80.

```
"traefik.http.services.myservice.loadbalancer.server.port=80"
```

Si associa poi un Router al Service appena creato e si esplicita il path per le richieste che dovranno essere soddisfatte da esso, in questo caso http://demo.localhost

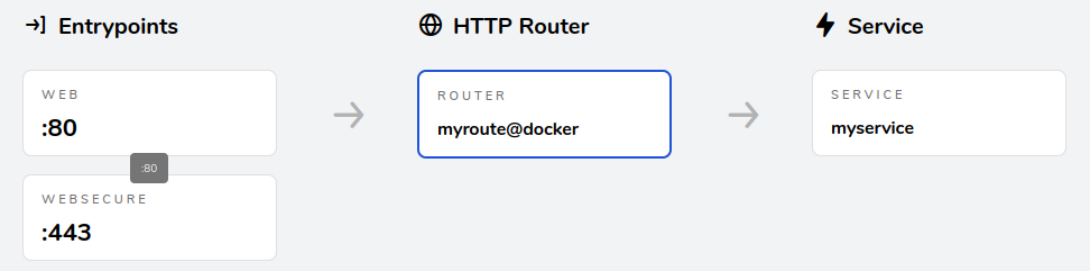
```
"traefik.http.routers.myroute.service=myservice"
"traefik.http.routers.myroute.rule=Host(`demo.localhost`)"
```

4.3.4 Avvio

A questo punto per avviare l'esempio si può procedere in due modi, utilizzando il costrutto docker-compose, o all'interno di un cluster docker swarm. Di seguito si procederà con il primo. Dalla directory in cui sono posizionati i due file vengono creati i due container.

```
$ docker-compose up -d
Starting ex1_traefik_1 ... done
Starting ex1_nginx_1   ... done
```

All'indirizzo http://localhost:8080 sarà presente la dashboard di Traefik. Da qui si nota come sia stato creato un Router *myroute@docker* (dove docker è in questo caso il provider) che collega entrypoints e service.



Anche il Service *myservice* è stato definito da docker-compose e sempre dalla dashboard è possibile verificare che al suo interno è presente un server, lo stesso server che risponde alla request su *http://demo.localhost*

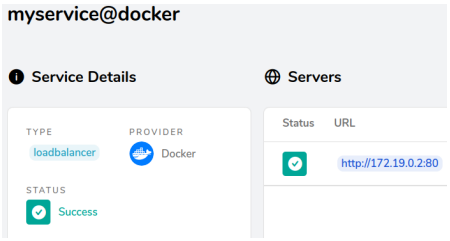


Figure 9: Il service

Figure 10: Il server nginx

4.4 HTTPS, TLS e Let's Encrypt

Di seguito viene mostrato come poter dotare il proprio server web di un certificato TLS in modo tale da poter abilitare la comunicazione tramite protocollo HTTPS ed assicurare un livello di sicurezza più alto rispetto alla comunicazione non criptata fornita da HTTP. Vi sono diversi metodi per dotarsi di un certificato TLS valido e di seguito sono riportate alcune alternative valide all'interno di Traefik.

Traefik self-signed certificate

Traefik di default se non trova certificati TLS genera automaticamente un certificato self-signed. Questo non è valido come un certificato rilasciato da una CA ed i browser navigando tramite https verso un arbitrario Service di Traefik segnalano che la connessione non è sicura.

Certificato rilasciato da una CA

Viene supportato il classico certificato rilasciato da una Certification Authority, è necessario in questo caso specificare il percorso di esso e gli Entrypoint di riferimento. Traefik esegue automaticamente il match certificato-dominio. E' possibile specificare il certificato sia nella configurazione statica, sia in quella dinamica tramite il provider File. La seconda ha il vantaggio di poter effettuare modifiche senza dover riavviare l'istanza di Traefik, trattandosi di configurazione dinamica.

Let's Encrypt

L'ultimo metodo per avere un certificato TLS valido è tramite Let's Encrypt. Attraverso esso Traefik è in grado di generare un certificato valido su richiesta. Il rilascio del certificato da parte di Let's Encrypt avviene dopo aver eseguito una delle "challenge", in cui la CA verifica l'effettivo possesso del dominio. Le challenge disponibili sono HTTP, DNS e TLS.

4.4.1 Utilizzo di Let's encrypt e HTTP challenge

La HTTP challenge segue il seguente schema.

All'interno della configurazione statica viene definito il tipo di challenge e Traefik, all'avvio, richiede un certificato TLS valido. Let's Encrypt, alla richiesta del certificato,

specifica all'istanza di Traefik un token e rimane in attesa. Traefik predispone un file all'indirizzo http://[dominio]/well-known/acme-challenge/[token] contenente il token e avvisa Let's Encrypt di verificare. Se la verifica va a buon fine, il certificato TLS viene rilasciato.

4.4.2 Richiesta di un certificato in pratica

Per poter effettivamente testare la challenge ed il relativo rilascio del certificato, ci si è dotati di un nome di dominio di terzo livello *sd2021traefik.ddnsfree.com* fornito da un provider DynamicDNS. Attraverso un DDNS è possibile associare l'indirizzo IP pubblico della propria macchina ad uno stesso hostname, anche se l'IP cambia nel tempo. Si ha quindi un indirizzo a cui poter accedere dalla rete e a cui anche Let's Encrypt può accedere per verificare l'effettiva proprietà del dominio.

Di seguito si riportano le modifiche alla configurazione statica e dinamica apportate all'esempio precedente.

```
#configurazione statica
certificatesResolvers:
  myresolver:
    acme:
      email: email-here
      storage: acme.json
      httpChallenge:
        #l'entrypoint è sulla porta 80
        entryPoint: web

#file docker-compose e configurazione dinamica
services:
    traefik:
        #....
        volumes:
        #directory dove sarà contenuto il certificato
        - ./letsencrypt:/letsencrypt

    nginx:
        #come esempio precedente...
        #associo il resolver della configurazione statica al Router
        - "traefik.http.routers.nginx.tls.certresolver=myresolver"
```

Oltre alla specifica definita sopra per la configurazione statica e dinamica, è necessario predisporre una directory ed un file *acme.json*, per il momento vuoto. Questo servirà per contenere il certificato rilasciato da Let's Encrypt. Necessario in questa fase dotare la directory dei giusti permessi, in ambiente UNIX questo è possibile tramite il comando *chmod 600*.

Mantenendo il certificato in memoria, Traefik non lo richiederà nuovamente alla CA, bensì utilizzerà quello precedentemente rilasciato evitando così il rischio di ottenere da Let's Encrypt l'errore "Too many certificates already issued for exact set of domains".

Ecco come si presenta la pagina dopo aver ottenuto il certificato.

Figure 11: Certificato TLS

4.5 Utilizzo di Middlewares

Viene di seguito analizzato l'uso di alcuni middleware all'interno di un'istanza di esempio di Traefik. I middleware sono componenti aggiuntivi che estendono i Routers e come tali vanno configurati all'interno della configurazione dinamica. Essi possono agire prima che la richiesta venga passata al Service, come ad esempio un redirect, o dopo che è stata eseguita la response, come ad esempio un compressione della response del service.

Per creare un Middleware all'interno di Traefik sono necessari due passi: la definizione vera e propria del middleware e il collegamento di esso ad uno o più router.

Di seguito è riportato un esempio di middleware visibile dalla dashboard, la quale esplica lo schema di connessione.

4.5.1 Redirect https

Il redirect https permette di reindirizzare le richieste eseguite tramite protocollo http, in https. La definizione, nel caso di Traefik definito in un file docker-compose utilizzando le label, è la seguente.

```
- "traefik.http.middlewares.redirecthttps.redirectscheme.scheme=https"
```

Il collegamento al Router avviene sempre tramite label, supponendo esista un Router chiamato *myroute*.

```
- "traefik.http.routers.myroute.middlewares=redirecthttps"
```

4.5.2 Authentication

Un' altro middleware utile è la basic authentication. Il quale permette di poter accedere ad un determinato Service solo previa autenticazione. La definizione ed il collegamento ad un router sono molto simili alla definizione del middleware di redirect, con la differenza che per *basicAuth* è necessario specificare anche le credenziali con cui si prevede di accedere.

```
#definizione di basicAuth
- "traefik.http.middlewares.authenticator.basicauth.users=
   admin:$$apr1$$62f2.Zjd$$3Uo.SJxB.CVx1Xa4ZIYhb."
```

La specifica dell'username e della password avviene previa crittografia, per generare una stringa compatibile con la label è possibile utilizzare il comando htpasswd (in ambiente Linux). Ad esempio, per la creazione di user: "admin", password: "sd2021" è stato eseguito

```
$ echo $(htpasswd -nb admin sd2021) | sed -e s/\\$/\\$\\$/g

 admin:$$apr1$$lacuSvMT$$IOBNQXDlEHOTSe5t9Z75J.
```

La seconda parte del comando è necessaria per sostuire ogni simbolo "$" con 2 di essi perchè, per Traefik, "$" sta ad indicare un variabile e potrebbero generarsi conflitti.

```
#associazione router-middleware
- "traefik.http.routers.myroute-secure.middlewares=authenticator"
```

Nel caso si vogliano associare più middleware allo stesso Router, sarà necessario farlo in una singola label separando i middleware da virgole, altrimenti verrà effettivamente associata solo l'ultima dichiarata.

4.6 Monitoraggio

Logs

E' possibile impostare nella configurazione statica l'utilizzo dei log ed il conseguente livello di dettaglio tra DEBUG, PANIC, FATAL, ERROR, WARN ed INFO. I log sono accessibili, se si utilizza il servizio in container Docker, visualizzando i log del container stesso.

Particolari log, da abilitare a parte, sempre nella configurazione statica, sono gli access log. Questi riportano informazioni riguardanti le richieste del servizio da parte dei client, tra cui informazioni sul richiedente, il path della richiesta e lo status code della richiesta. Per abilitare gli access log è necessario inserire la seguente riga nel file di configurazione statica.

```
accesslog:{}
```

In questo modo, nei log del servizio sono raccolti tutti gli access log. Sempre nella configurazione statica questi sono filtrabili in modo tale da visualizzare solo le richieste che hanno generato un solo status code o hanno avuto un tempo di risposta maggiori di un certo valore.

Nel corso del testing e di questi primi utilizzi i log si sono rivelati molto utili nel capire mano a mano gli errori compiuti. Traefik per ogni service, router, middleware, certificato tls aggiunto lo riporta nei log ed è stato semplice capire cosa di volta in volta non andasse a buon fine.

I log sono visualizzabili a partire dai log del container stesso. Supponendo il nome *service-1*, mediante il comando

```
$ docker service logs service-1
```

5 Utilizzo di Traefik Mesh

In un architettura a microservizi, il classico monolite si spezza in diverse parti, definite servizi, con compiti ben precisi e definiti, ognuno infatti ha la propria business logic. La piattaforma più importante e usata per lo sviluppo di architetture a microservizi è Kubernetes.

Il disaccoppiamento funzionale in vari servizi che compongono l'applicativo, porta al dover far comunicare i servizi tra loro. Ciò che una rete mesh fa è proprio questo, permette la comunicazione intra-service e aggiunge funzionalità quali retry logic e analisi di metriche. Per *retry logic* si intende uno scenario in cui un servizio sia irraggiungibile e si cerca di ricomunicare con esso o una sua replica all'interno del cluster.

Un' altra feature significativa che una rete mesh offre è lo split del traffico, descritto come load balancing nella sezione dedicata a Traefik Proxy. In questo modo è possibile, per i microservizi, specificare la comunicazione con un altro servizio disinteressandosi del servizio che effettivamente gestirà la richiesta e apre la possibilità a scenari già precedentemente descritti di canary e blue green deployment.

5.1 Ambiente e strumenti

Per effettuare gli esempi d'utilizzo riguardanti Traefik Mesh in ambiente Kubernetes è stato utilizzato **Minikube**. Minikube è un tool per il setup di cluster Kubernetes locali, tramite il quale è possibile creare un ambiente Kubernetes in maniera rapida e veloce.

5.2 Primo utilizzo di Traefik Mesh

5.2.1 I servizi

Per questo primo esempio di utilizzo verrà installato Traefik Mesh su un cluster locale Kubernetes e verrà utilizzato, a partire da un servizio *client*, per comunicare con un servizio *server*.

Il servizio *client* è un'istanza di una Docker Image basata su Alpine, la quale offre strumenti di base, tra cui quello utilizzato per questo primo esempio *curl*.

Il servizio *server* invece, il quale sarà presente in più repliche, è un'istanza di una Docker Image fornita da Traefik rappresentante un semplice web server sviluppato in Go. Esso verrà utilizzato per rispondere alle richieste del client.

La configurazione dei due servizi avviene tramite file *yaml* di seguito riportati.

Client

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: client
  namespace: app
  labels:
```

```
    app: client
spec:
  replicas: 1
  selector:
    matchLabels:
      app: client
  template:
    metadata:
      labels:
        app: client
    spec:
      containers:
        - name: client
          image: giantswarm/tiny-tools:3.9
          imagePullPolicy: IfNotPresent
          command:
            - "sleep"
            - "infinity"
```

Server

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: server
  namespace: test
  labels:
    app: server
spec:
  replicas: 2
  selector:
    matchLabels:
      app: server
  template:
    metadata:
      labels:
        app: server
    spec:
      containers:
        - name: server
          image: traefik/whoami:v1.6.0
          ports:
            - containerPort: 80
---
```

```
kind: Service
apiVersion: v1
metadata:
  name: server
  namespace: test
spec:
  selector:
    app: server
  ports:
    - name: web
      protocol: TCP
      port: 80
      targetPort: 80
```

5.2.2 Creazione cluster

Al fine di creare il cluster locale Kubernetes, è necessario installare Minikube, di cui si rimanda l'installazione alla documentazione e kubectl, il quale è il punto di ingresso per la comunicazione e la gestione di un cluster kubernetes, anche per esso si rimanda l'installazione alla documentazione. Una volta installati entrambi, è possibile avviare il cluster tramite il comando

```
$ minikube start
```

Al fine di interagire con esso ci si avvarrà, come precedentemente specificato, di *kubectl*.

5.2.3 Installazione di Traefik Mesh

Una volta dispiegato il cluster, è necessario installare Traefik Mesh al fine di utilizzarlo. E' possibile farlo tramite *helm*, package manager specifico per Kubernetes, tramite i seguenti comandi

```
$ helm repo add traefik-mesh https://helm.traefik.io/mesh

$ helm repo update

$ helm install traefik-mesh traefik-mesh/traefik-mesh
```

5.2.4 Esempio d'uso

Per prima cosa viene definito un *namespace* Kubernetes, un cluster virtuale sviluppato sopra al cluster Kubernetes creato. Kubernetes supporta multipli namespace per lo stesso cluster, utile per un'ulteriore suddivisione della logica.

```
$ kubectl create namespace app
namespace/app created
```

Dopodichè viene fatto il loading delle configurazioni per i servizi client e server.

```
$ kubectl apply -f server.yaml
deployment.apps/server created
service/server created

$ kubectl apply -f client.yaml
deployment.apps/client created
```

Vengono individuate le pod create e i servizi, in modo tale da poter verificare che tutto sia in esecuzione.

```
$ kubectl get all -n app
NAME                          READY   STATUS    RESTARTS   AGE
pod/client-7d99c5976-zpjw2    1/1     Running   0          15s
pod/server-cf9fbc6cb-4576l    1/1     Running   0          40s
pod/server-cf9fbc6cb-gxxbz    1/1     Running   0          40s

NAME             TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
service/server   ClusterIP   10.104.116.219   <none>        80/TCP     40s

NAME                     READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/client   1/1     1            1           15s
deployment.apps/server   2/2     2            2           40s

NAME                                DESIRED   CURRENT   READY   AGE
replicaset.apps/client-7d99c5976    1         1         1       15s
replicaset.apps/server-cf9fbc6cb    2         2         2       40s
```

Avendo individuato il nome della pod rappresentante il client, viene aperta una sessione con esso, in modo tale da simulare la comunicazione tra i due servizi *client* e *server*.

```
$ kubectl -n app exec -ti pod/client-7d99c5976-zpjw2 -- ash
/ #
```

Attraverso la seguente richiesta, non viene utilizzato il servizio di Mesh dato da Traefik ma bensì solamente il servizio DNS fornito da Kubernetes

```
/ # curl server.app.svc.cluster.local

Hostname: server-cf9fbc6cb-gxxbz
IP: 127.0.0.1
IP: 10.244.0.16
RemoteAddr: 10.244.0.17:42764
GET / HTTP/1.1
```

```
Host: server.app.svc.cluster.local
User-Agent: curl/7.64.0
Accept: */*
```

ciò che Traefik permette è il reindirizzamento delle richieste tramite il proprio servizio di Mesh, aprendo agli scenari e alle possibilità descritte in precedenza.

```
/ # curl server.app.traefik.mesh

Hostname: server-cf9fbc6cb-4576l
IP: 127.0.0.1
IP: 10.244.0.15
RemoteAddr: 10.244.0.2:42666
Host: server.app.traefik.mesh
User-Agent: curl/7.64.0
X-Forwarded-For: 10.244.0.17
X-Forwarded-Host: server.app.traefik.mesh
X-Forwarded-Port: 80
X-Forwarded-Proto: http
X-Forwarded-Server: traefik-mesh-proxy-x4xl2
X-Real-Ip: 10.244.0.17
```

Per chiudere la sessione con il client è necessario utilizzare il comando *exit*.

5.3 Traffic split

In questo secondo esempio, verrà analizzata la funzione di traffic split offerta da Traefik Mesh, tramite la quale il traffico rivolto al server viene direzionato verso servizi diversi in base a determinati pesi dati ai servizi stessi.

L'esempio parte dalla base della sezione precedente, con l'aggiunta di una versione 2 del server con cui simulare un canary deployment e un server d'appoggio posizionato davanti all'infrastruttura.

```
$ kubectl create namespace app
namespace/app created
$ kubectl apply -f server.yaml
service/server created
$ kubectl apply -f server-v1.yaml
deployment.apps/server-v1 created
service/server-v1 created
$ kubectl apply -f server-v2.yaml
deployment.apps/server-v2 created
service/server-v2 created
$ kubectl apply -f client.yaml
deployment.apps/client created
```

Di seguito la configurazione necessaria a Traefik per svolgere la funzione, conforme allo standard SMI per le reti mesh all'interno di un cluster Kubernetes.

```
apiVersion: split.smi-spec.io/v1alpha3
kind: TrafficSplit
metadata:
  name: server-split
  namespace: app
spec:
  service: server
  backends:
    - service: server-v1
      weight: 80
    - service: server-v2
      weight: 20
```

Loading della configurazione per il traffic split, definito in un file yaml conforme a SMI.

```
$ kubectl apply -f split-configuration.yaml
trafficsplit.split.smi-spec.io/server-split created
```

A questo punto è possibile notare come il traffico sia splittato prevalentemente sulle pod del service *server-v1* rispetto a quelle del service *server-v2*

```
$ kubectl -n app exec -ti client-7d99c5976-n76qn -- ash
/ # curl server.app.traefik.mesh

Hostname: server-v1-647586c97d-x4vwv
IP: 127.0.0.1
IP: 10.244.0.20
RemoteAddr: 10.244.0.2:37304
Host: server-v1.app.traefik.mesh:80
User-Agent: curl/7.64.0
X-Forwarded-For: 10.244.0.24, 10.244.0.1
X-Forwarded-Host: server.app.traefik.mesh
X-Forwarded-Port: 80
X-Forwarded-Proto: http
X-Forwarded-Server: traefik-mesh-proxy-x4xl2
X-Real-Ip: 10.244.0.24
```

6 Testing

6.1 Service auto discovery

In questa sezione, a partire dalle configurazioni di Traefik e un servizio web discussi sopra, si testerà l'auto discovery dei servizi, ovvero se Traefik riesce a individuare nuove repliche sulla rete e a eliminare automaticamente routing a servizi non più attivi o guasti.

Inizialmente vengono istanziati i due container utilizzando docker swarm all'interno di uno stack denominato *simple app*

```
$ docker stack deploy -c docker-compose.yml simple_app
Creating network simple_app_default
Creating service simple_app_traefik
Creating service simple_app_nginx
```

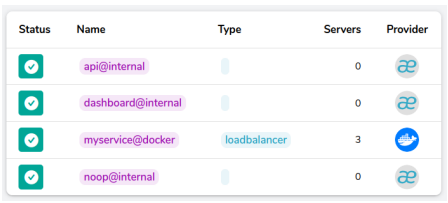
Verifico che siano stati effettivamente avviati i due servizi, uno per Traefik ed uno per il server web nginx. Inoltre verifico che sia la dashboard, sia il server rispondano correttamente ai loro indirizzi.

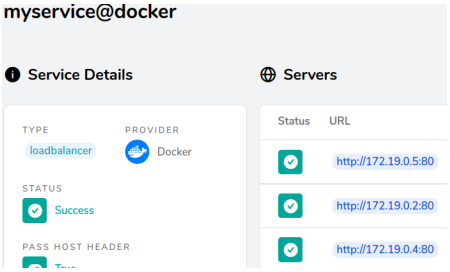
```
$ docker service ls
ID             NAME                MODE         REPLICAS   IMAGE
jm87e3r9xm9b   simple_app_nginx    replicated   1/1        nginxdemos/hello:latest
a142gxdi7hsr   simple_app_traefik  replicated   1/1        traefik:latest
```

A questo punto, attraverso swarm, è possibile replicare il servizio di nginx. Replicando il servizio, verrà istanziato un numero di container pari al numero specificato.

```
$ docker service scale simple_app_nginx=3
simple_app_nginx scaled to 3
overall progress: 3 out of 3 tasks
1/3: running
2/3: running
3/3: running
verify: Service converged
```

Avendo replicato il servizio Traefik dovrebbe riconoscere automaticamente i due nuovi servizi e aggiungerli al load balancer già precedentemente creato per la prima istanza.





Aggiornando la dashboard è possibile verificare come automaticamente Traefik abbia riconosciuto i due container appena creati e li abbia associati allo stesso Service.

Visitando poi il servizio si nota come le informazioni riguardanti l'indirizzo ed il nome del server cambino ogni volta che si esegue la richiesta. Traefik bilancia correttamente le richieste ai 3 vari server.

6.1.1 Guasto ad un container

E' possibile inoltre simulare il guasto ad un container, il quale causa il suo stop e la conseguente eliminazione dai container attivi. Swarm istanzierà un nuovo container per mantenere il numero di repliche costante. Traefik dovrebbe riconoscere eliminazione e ricreazione di un nuovo container.

Mediante il comando *docker container ls* ottengo l'id di un container nginx, questo verrà stoppato ed eliminato. Verifico poi la creazione di una nuova istanza creata automaticamente da swarm (osservando il numero di repliche).

```
$ docker stop ea8e5cb4c0ac
ea8e5cb4c0ac
$ docker rm ea8e5cb4c0ac
ea8e5cb4c0ac
$ docker service ls
ID             NAME                  MODE         REPLICAS   IMAGE
jm87e3r9xm9b   simple_app_nginx      replicated   2/3        nginxdemos/hello:latest
a142gxdi7hsr   simple_app_traefik    replicated   1/1        traefik:latest

$ docker service ls
ID             NAME                  MODE         REPLICAS   IMAGE
jm87e3r9xm9b   simple_app_nginx      replicated   3/3        nginxdemos/hello:latest
a142gxdi7hsr   simple_app_traefik    replicated   1/1        traefik:latest
```

Osservo attraverso la dashboard che il numero di repliche venga ridotto e poi riportato stabilmente a 3 (come ho fatto da command line). Traefik in questo caso ha riconosciuto eliminazione ed aggiunta, riuscendo a mantenere il servizio web disponibile in ogni caso, senza più reindirizzare il traffico al container ormai down, cosa che avrebbe causato errori nel soddisfacimento della richiesta.

6.2 Weighted Round Robin e Canary

Nello sviluppo di applicazioni e servizi web può essere utile rilasciare aggiornamenti in maniera graduale, rilasciando le nuove funzionalità solo a una ristretta cerchia di utenti, questa pratica prende il nome di Canary deployment.

Ciò che rende possibile questo scenario è un particolare load-balancer che agisce mediante algoritmo weighted round-robin. Il bilanciamento è effettuato sui Service, non sui servizi finali disponibile all'utente.

A partire dagli esempi precedenti, dove era già stato definito un Service responsabile per il backend NGiNX, si è aggiunto quindi un altro Service responsabile per il nuovo servizio. Traefik al momento supporta l'algoritmo WRR solamente su alcuni provider, tra cui non configura Docker Swarm. Perciò oltre a Docker, in questo esempio, si è aggiunto il provider File all'intero della configurazione dinamica.

```
#configurazione statica
providers:
  file:
    #directory contenente la configurazione dinamica per il provider file
    directory: /fileconfig
    watch: true
```

All'interno di un file YAML si è poi aggiunta la configurazione dinamica per il provider File.

```
#file canary.yml
http:
  services:
    #service che si occuperà di gestire il wrr tra i vari services
    canary:
      weighted:
        sticky:
          cookie: {}
        services:
          #service principale
          - name: myservice_old@docker
            weight: 8
          #nuovo service
          - name: myservice_new@docker
            weight: 2
```

La configurazione specifica un Service denominato *canary* e, mediante la keyword *weighted*, vengono specificati i Service a cui *canary* dovrà fare riferimento. In particolare, ogni Service avrà uno specifico peso rappresentate la percentuale di richieste reindirizzate all'uno e all'altro Service.

Mediante la keyword sticky invece, viene specificato di rendere la connessione tra il client e il servizio persistente, un client che richiederà il servizio verrà servito sempre dallo stesso Service.

Si definisce quindi la configurazione dinamica per il Service *canary* e i due sotto servizi ad esso collegati *myservice-old* e *myservice-new*. Mediante *@docker* si specifica il provider dei Service.

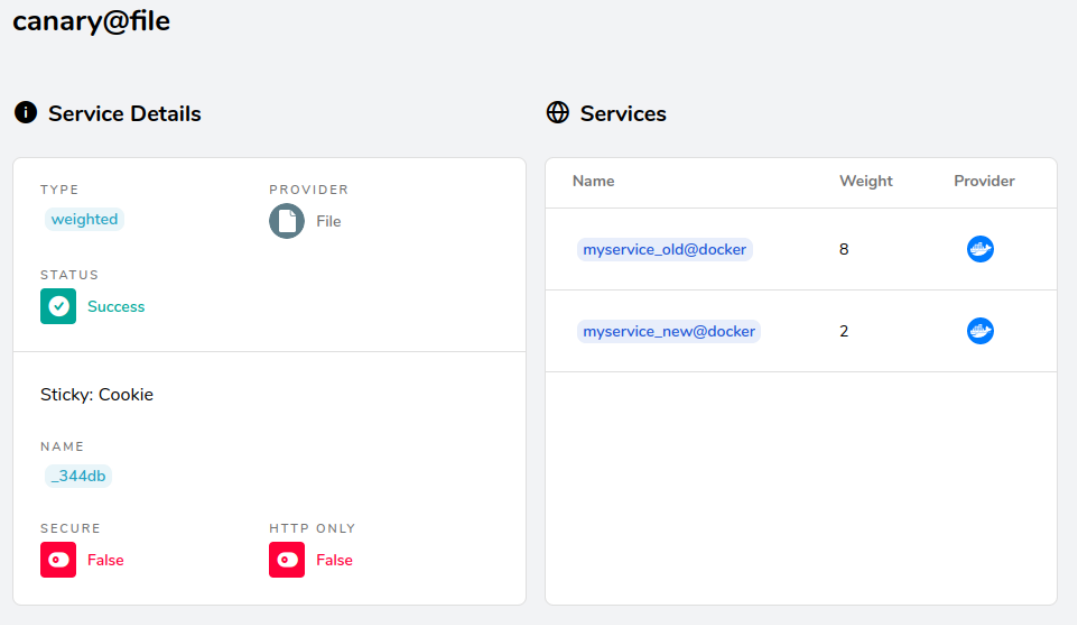


Figure 12: I service sotto weighted round robin

All'interno del file *docker-compose* discusso negli esempi precedenti si effettua il binding tra servizio docker e Service traefik.

```
services:
  #istanza di traefik
  traefik:
    # specifica configurazione dinamica provider File
    configs:
      - source: canary-config
        target: /fileconfig/canary.yml
    labels:
      # altre labels riguardanti dashboard, middleware ecc...
      # associo la route del servizio al Service canary specificato nel file canary.yml
      - "traefik.http.routers.myroute.rule=Host(`sd2021traefik.ddnsfree.com`)"
      - "traefik.http.routers.myroute.tls.certresolver=myresolver"
      - "traefik.http.routers.myroute.service=canary@file"

  #server nginx
  nginx-v1:
    labels:
    # il nome di questo service deve essere il medesimo
    # specificato all'interno del file canary
    - "traefik.http.services.myservice_old.loadbalancer.server.port=80"
```

```
#new version server nginx
nginx-v2:
  labels:
  # il nome di questo service deve essere il medesimo
  # specificato all'interno del file canary
  - "traefik.http.services.myservice_new.loadbalancer.server.port=80"
```

Come si può notare, è stata definita una singola root comune verso il Service *canary*. La logica per cui da questo service si venga reindirizzati verso uno degli altri due è contenuta all'interno del file *canary.yml* di configurazione dinamica.

La seguente configurazione è solamente un semplice test volto a mirare l'effettivo bilanciamento delle varie richieste ai vari Service nelle modalità richieste, astrae pertanto dalle possibili implementazioni di applicazioni sviluppabili all'interno dei container. Si è scelto di utilizzare come "applicazione" due semplici istanze di server nginx.

7 Un esempio di applicazione

Negli esempi precedenti si è cercato di prendere familiarità con la configurazione e le funzionalità di Traefik. A tal fine si è fatto uso di server nginx, il quale supporta anch'esso funzioni di reverse proxy e load balancing. Sebbene non siano stati sfruttati, si è cercato in questo esempio di staccarsi definitivamente dal tool nginx e sviluppare due semplici server nodeJS containerizzati. Nella seguente sezione viene quindi dispiegata un'applicazione contenente diversi servizi, in particolare due semplici REST API, raggiungibili allo stesso indirizzo ma con path diversi mediante Traefik. Oltre a testare quindi il load balancing, ad esempio con la creazione di diverse istanze per questi servizi, attraverso il reverse proxy nativo di Traefik è stato possibile mantenere un indirizzo comune, come se il servizio fosse uno solo, nascondendo così, davanti a un unico punto d'accesso, l'architettura interna.

7.1 I servizi

I servizi in questione sono due semplici server sviluppati in NodeJS e Express, i quali gestiscono diverse richieste riguardanti uno scenario di ristoranti e ordini. Entrambi sono disponibili su Docker Hub ai seguenti indirizzi

- Restaurants
- Orders

Traefik gestirà la parte di reindirizzamento al microservizio giusto in base a determinate regole di routing e bilancerà il carico tra essi se dovessero essere presenti più istanze di questi. In particolare, l'indirizzo mantenuto comune a entrambi i servizi è quello utilizzato nei test precedenti, con l'aggiunta di specifiche regole di routing per ogni servizio.

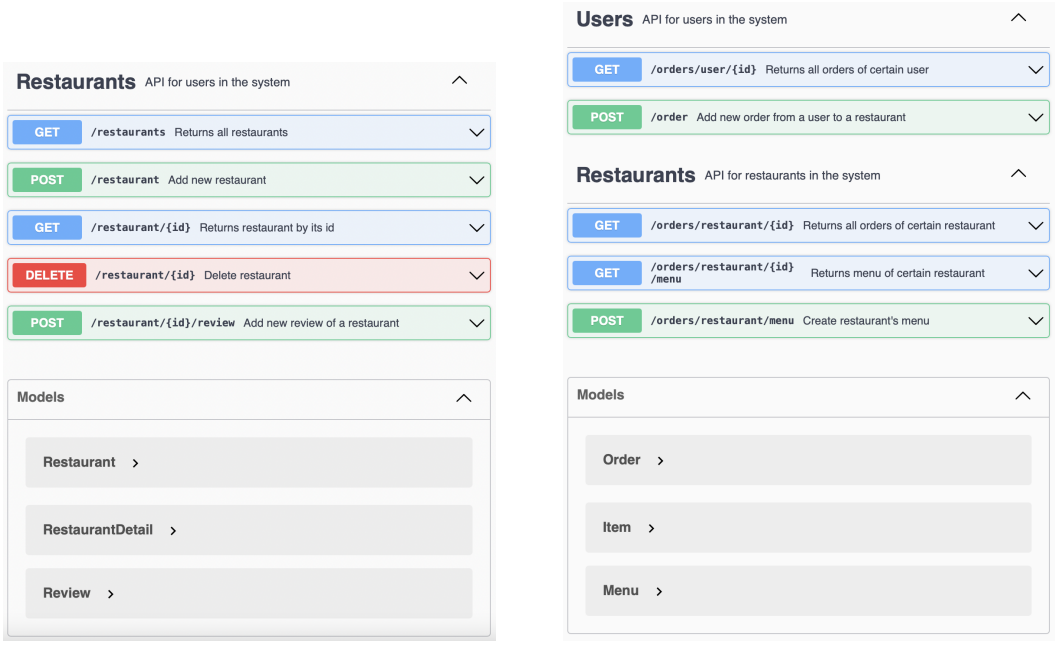
```
- "traefik.http.routers.restaurants-route-secure.rule=
    (Host(`sd2021traefik.ddnsfree.com`) &&
    (PathPrefix(`/restaurants`) || PathPrefix(`/restaurant`)))"

- "traefik.http.routers.orders-route-secure.rule=
    (Host(`sd2021traefik.ddnsfree.com`) &&
    (PathPrefix(`/orders`) || PathPrefix(`/order`)))"
```

7.1.1 Swagger

Swagger è un tool per lo sviluppo di documentazione di API, attraverso esso è stato possibile documentare le due API REST create e dotare ogni servizio di una dedicata alla visualizzazione della doc. Swagger mette a disposizione un package npm specifico alla realizzazione di documentazione specifica per applicazioni express. E' possibile, dato un progetto inizializzato con npm, scaricare il pacchetto tramite

```
$ npm install swagger-ui-express
```

La documentazione viene specificata tramite un file Json, seguendo quanto riportato alla documentazione ufficiale di Swagger. Di seguito si riporta uno spaccato della documentazione per i due servizi, visibile agli URL specifici */restaurants/api-docs* e */orders/api-docs*.



(a) Restaurants (b) Orders

Figure 13: Documentazione API tramite Swagger

7.2 Funzionamento

I servizi, oltre a offrire metodi get e post, permettono di visualizzare l'indirizzo del servizio che sta gestendo la richiesta, in questo modo è possibile verificare, qual'ora fossero presenti più istanze dello stesso servizio l'effettivo load balancing tra esse.

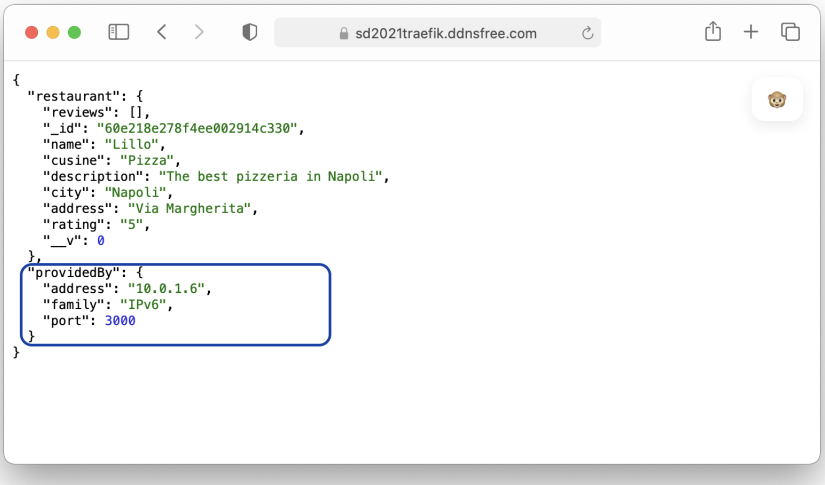


Figure 14: Verifica dell'indirizzo del servizio

Questo semplice esempio di applicazione, attraverso l'utilizzo di Traefik e l'adozione di un'architettura a microservizi permette un disaccoppiamento forte tra gli elementi responsabili della logica del servizio, ma garantisce un entrypoint comune all'applicativo, consentendo così di separare alcune logiche di sviluppo mantenendo però una connessione funzionale.

8 Conclusioni

8.1 Performance della tecnologia

Nelle prove effettuate Traefik si è rivelato convincente sotto molti punti di vista. La possibilità di usare questa tecnologia in uno scenario containerizzato in maniera relativamente semplice è un suo punto di forza. Gestendo in maniera ottimale e trasparente all'utente il **load balancing** tra i servizi, la gestione dei **fault** e la possibilità di questi di scalare al bisogno.

Un aspetto da considerare è il fatto che, così come un normale container o processo, anche Traefik stesso può incorrere in fallimenti improvvisi o crash. Con **high availability** si intende la garanzia che un servizio sia online, stabilmente, senza perdite di connessione. Nel caso comune, Traefik, ponendosi come unico punto di ingresso rappresenta un single-point of failure nel sistema, non garantendo high availability. Se l'unica istanza di Traefik dovesse fallire, si perderebbe completamente la connessione al sistema o al cluster sottostante. Gli altri servizi e i container dietro Traefik continuerebbero a essere disponibili, ma mancherebbe il punto di accesso al sistema.

Per garantire high availability in un cluster avente traefik come edge router e reverse proxy, è necessario creare un cluster Traefik, di cui preferibilmente fare il deploy in host separati. Per quanto riguarda gli ip di queste istanze, dovrebbe essere assegnato un virtual IP davanti al cluster rappresentante le varie istanze di traefik, in modo tale da poterle raggiungere indifferentemente dalla singola istanza.

8.2 Competitor

Tra i principali competitor di Traefik come reverse proxy e load balancer si trovano diversi tool, tra cui i più diffusi sono **nginx** e **HAproxy**. Essi sono di fatto i leader in questo settore, sia per un motivo temporale (sono da molto sul mercato), sia per l'efficienza e le molte funzionalità che offrono.

Entrambi hanno una però configurazione piuttosto statica, la quale in determinati casi può essere modificata a runtime manualmente, ma non supportano una configurazione dinamica tramite API così come Traefik, almeno nelle loro versioni open-source.

Il grande vantaggio di Traefik e grande punto di differenziazione è quindi l'assoluta integrazione con ambienti containerizzati e il conseguente supporto e dinamicità nella creazione e nel mantenimento della configurazione. Offre meno opzioni, meno algoritmi di load balancing, ma pensato come tool out-of-the-box richiede poco tempo in termini di configurazione per funzionare correttamente e permette di architettare soluzioni scalabili e dinamiche.

Il punto debole riscontrato durante l'utilizzo è stata la documentazione, scarna dal punto di vista della visione generale dell'architettura del tool e mancante di esempi significativi, punto nel quale, con questo studio, si spera di aver sopperito. Questo ha richiesto un discreto tempo per la configurazione iniziale, ma una volta entrati nei meccanismi e nel dialogo con Docker, Traefik si è rivelato piuttosto trasparente allo sviluppo dell'architettura.

References

[1] Docker. *Docker Compose Documentation*. URL: https://docs.docker.com/compose/.

[2] Docker. *Docker Documentation*. URL: https://docs.docker.com.

[3] Docker. *Docker Swarm Documentation*. URL: https://docs.docker.com/engine/swarm/.

[4] Let's Encrypt. *Let's Encrypt Documentation*. URL: https://letsencrypt.org/docs/.

[5] Kubernetes. *Kubernetes Documentation*. URL: https://kubernetes.io/docs/home/.

[6] Kubernetes. *Minikube Documentation*. URL: https://minikube.sigs.k8s.io/docs/.

[7] NodeJS. *Dockerizing a Node.js web app*. URL: https://nodejs.org/en/docs/guides/nodejs-docker-webapp/.

[8] NodeJS. *NodeJS Documentation*. URL: https://nodejs.org/en/docs/.

[9] *Reverse Proxy*. URL: https://en.wikipedia.org/wiki/Reverse_proxy.

[10] Jason Skowronski. *Intro to deployment strategies: blue-green, canary, and more*. URL: https://dev.to/mostlyjason/intro-to-deployment-strategies-blue-green-canary-and-more-3a3.

[11] *SMI Service Mesh Interface*. URL: https://smi-spec.io.

[12] Swagger. *Swagger Documentation*. URL: https://swagger.io/docs/.

[13] Traefik. *Traefik Documentation*. URL: https://doc.traefik.io/traefik/.