**Vitaletti**

**Docker**

Docker è una tecnologia che automatizza istallazione e configurazione, la maggior parte delle app sono Cloud,cioè mi garantisce elasticità di servizio, portabilità tra ambienti di sviluppo (per garantire portabilità devi avere omologazione certi standard – non mi porto dietro il SO), lock-in

Divido l’applicazione in microservizi, in modo tale da rendere il tutto scalabile rispetto ad applicazione monolitica. Vantaggio sta nel non dover riscrivere tutto un codice (o anche ricompilarlo) ogni volta ma modificare unicamente il microservizio specifico, inoltre in questo modo non ho un unico punto di rottura.

Nel mondo moderno ci sono quindi 3 approcci indipendenti delle infrastrutture:

1) Tradizionale

* *Costi molto alti*: perché ogni volta devo comprare un nuovo oggetto da far girare sulla macchina
* *Lento lo sviluppo:* perché dopo aver finito l’istallazione e dopo aver sviluppato tutto,se voglio installarlo su un altra macchina devo vedere se risulta essere compatibile. Se cosi non fosse devo modificarlo e riprovare finché non risulti essere compatibile.

2) Tradizionale Moderno – Virtualizzazione

* Benefici
  + Server fisico può contenere applicazioni multiple
  + Ogni app gira su virtual machine
  + Una fisica VM viene divisa in VM
  + Maggiore scalabilità rispetto all’approccio tradizionale
* Svantaggi
  + Un intero Guest OS che genera spreco di risorse
  + Più macchine VM girano, più risorse ne hanno bisogno
  + Ogni VM richiede allocazione CPU
  + Portabilità delle applicazioni non è garantita

3) Container

* Virtualizzando solamente il sistema operativo e le componenti necessarie all’esecuzione delle applicazioni (librerie e file binari), invece che l’intera macchina, i tempi di avvio si riducono notevolmente rispetto quelli di una VM
* Container consentono agli sviluppatori di suddividere ulteriormente le risorse in microservizi, garantendo maggior controllo sull’eseguibilità delle applicazioni e miglioramento delle performance generali.
* Container non devono attivare SO
* Vantaggi
  + Portabilità
  + Efficienza
  + Velocità

Se nella VM ho un’infrastruttura fisica, un Hypervisor e vari guest; in Docker utilizzo “container” che hanno solo le librerie per far girare app.

***DOCKER BASIC***

Immagine: Classe

Container: Istanza di una classe

Volume: Astrazione dello storage

Docker Compose: descrive più istanze di un container

DockerFile: descrive sola istanza del container

***DOCKER NETWORK***

La rete di Docker è costituita da Drivers:

* *Bridge*  è la rete di default che viene creata appena avvii Docker e i container avviati si ollegano ad essa se non diversamente specificato. Bridge consente ai container, connessi ad essa, di comunicare tra di loro (per esempio NodeJS, CouchDB e Nginx che si trovano in container isolati).
* *Overlay* è equivalente al bridge ma i container non sono posti su un unica macchina ma sparsi nel mondo di internet
* *Host*
* *Macvlan* consentono di assegnare un indirizzo MAC a un contenitore, facendolo apparire come un dispositivo fisico sulla rete. Questo implica che vi è una virtualizzazione anche dell’indirizzo MAC

**Riepilogo**

* **User-define bridge networks** sono le migliori quando sono necessari più contenitori per comunicare sullo stesso host Docker.
* **Host networks** sono le migliori quando lo stack di rete non deve essere isolato dall'host Docker, ma si desidera isolare altri aspetti del contenitore.
* **Overlay networks** sono le migliori quando hai bisogno di contenitori in esecuzione su diversi host Docker per comunicare o quando più applicazioni lavorano insieme utilizzando i servizi di swarm.
* **Macvlan newtworks** sono le migliori quando si **esegue la** migrazione da un'installazione di VM o si richiede che i contenitori **sembrino** host fisici sulla rete, ognuno con un indirizzo MAC univoco.

***DOCKER SWARM***

Uno swarm è costuito da più Docker host che lavorano in modalità swarm e fungono sia da manger che da worker.

Quando si va a creare un servizio si va a definire lo stato ottimale (numero di repliche, risorse di rete e di archiviazione disponibili, porte che il servizio espone al mondo esterno e altro).

Docker funziona per mantenere lo stato desiderato. Ad esempio, se un nodo di lavoro diventa non disponibile, Docker pianifica le attività del nodo su altri nodi.

***Nodi***

Un **nodo** è un'istanza del Docker engine che partecipa allo swarm. È possibile eseguire uno o più nodi su un singolo computer fisico o su un server cloud, ma in genere le distribuzioni di swarm di produzione includono nodi Docker distribuiti su più macchine fisiche e cloud.

Per distribuire l'applicazione su uno swarm, si invia una specifica di servizio a un **nodo magager.** Il nodo manager invia unità di lavoro, chiamate task, ai nodi worker.

**I nodi di lavoro** ricevono ed eseguono attività inviate dai nodi manager. Il nodo worker notifica al nodo manager lo stato corrente delle attività assegnate in modo che il manager possa mantenere lo stato desiderato di ciascun lavoratore.

***Servizi***

Il servizio è l’attivita da eseguire sul manager o sui nodi worker. Quando si crea un servizio, si specifica l'immagine del container da utilizzare e quali comandi eseguire all'interno dei container in esecuzione.

Nei **servizi replicati**, il swarm manager distribuisce un numero specifico di attività di replica tra i nodi in base alla scala impostata nello stato desiderato.

Per **servizi globali**, lo swarm esegue un'attività per il servizio su ogni nodo disponibile nel cluster.

Un task trasporta una docker container e i comandi da runnare all’interno del container. I nodi Manager assegnano le attività ai nodi worker in base al numero di repliche impostate nella scala del servizio. Una volta che un'attività è assegnata a un nodo, non può spostarsi su un altro nodo. Può essere eseguito solo sul nodo assegnato.

**Socket e HTTP**

Un socket è un endpoint di un collegamento di comunicazione bidirezionale tra due programmi in esecuzione sulla rete.

Un socket è associato a un numero di porta in modo che il livello TCP possa identificare l'applicazione a cui i dati sono destinati a essere inviati. Un endpoint è una combinazione di un indirizzo IP e un numero di porta.

Ogni connessione TCP può essere identificata in modo univoco dai suoi due endpoint. In questo modo puoi avere più connessioni tra l'host e il server.

Nginx – Web Server (F)

Nodejs – Application Server (S)

**REST – CRUD**

Rest è uno stile architetturale e non uno standard. RESTful viene in genere utilizzato per fare riferimento a servizi Web che implementano l’architettura REST.

Rest riguarda come rappresentare le risorse (identificate attraverso URI – come accedo, dove,con cosa,e a cosa sono interessato ad accedere), serve per manipolare le risorse, quindi un modo facile e flessibile di scrivere un web service.

Principi Implementazione REST

1) *Architettura Client/Server*: che rende i vari servizi implementati indipendenti l’uno dall’altro.

2) *Stateless:* vuol dire che non conserva lo stato/memoria del processo. Utilizziamo quindi, come in HTTP usiamo i Cookie (info che tracciano il comportamento dell’utente sul web)

3) *Cache*: utilizzo del caching, cioè se il contenuto che ho,è ancora valido, è inutile che vado alla sorgente

4) *Layered* System: trasparenza nell’interagire con server che a sua volta ha molti layer

5) *Code on demand:* una risorsa può essere codice

Rest è stateless, non conserva lo stato del processo,ma io voglio accedere allo stato (prendendolo o cambiandolo)

* Rappresentazione della risorsa con JSON
* Gestire stato risorsa (per esempio con HTTP)

Tramite i metodi REST: POST,GET,PUT,DELETE

posso andare a prendere le risorse “in giro” per il Web (Google,Facebook..) e li conpongo tra loro nell’Application Server.

In conclusione, REST ci consente,non solo di prendere/ottenere risorse, ma anche di manipolarle, quindi posso cambiarne lo stato.

**REST vs SOAP**

Soap utilizzato per lo scambio di messaggi tra componenti software secondo le regole di sintassi di XML. Soap è un protocollo, Rest è un framework.

La differenza principale tra SOAP e REST è il grado di accoppiamento tra implementazioni client e server. Un client SOAP funziona come un'applicazione desktop personalizzata, strettamente associata al server. C'è un contratto rigido tra il client e il server e non funziona più nulla se uno di questi cambia.

Un client REST è più simile a un browser. È un client generico che utilizza una interfaccia con metodi standardizzati che restituisce dati in formati più o meno standard.

REST è indipendente dal protocollo. Non è accoppiato necessariamente con HTTP. E' possibile seguire, ad esempio un link ftp da un sito web. Un'applicazione REST può utilizzare qualsiasi protocollo per il quale esiste uno schema standardizzato di richiesta URI.

Soap mette in risalto il concetto di servizio, invece il REST mette in risalto il concetto di risorsa. Un web service basato sul soap espone un insieme di metodi richiamabili da remoto da parte di un client, mentre il REST è custode di un insieme di risorse sulle quali in client può chiedere le operazioni canoniche del protocollo HTTP.

*Esempio Soap è più educato di REST.*

REST : GET Coffee → diretto ma potrebbe non arrivarti il caffè che ti aspetti

SOAP: quello che voglio chiederti appartiene al Web Server “Coffee” (specfiche coffee)

XML vs JSON

**CouchDB – Database - No SQL vs SQL**

CouchDB è un database non relazionale (NOSQL). A differenza dei database relazionali **CouchDB non possiede il concetto di schema, ma il collezioni di documenti.**

Ogni documento in ogni database può essere strutturato in modo diverso dagli altri, cosa che invece non è possibile nei database relazionali (SQL) dove tutto ruota intorno al concetto di tabella (formata da righe e colonne) e che tra le varie tabelle di un database possono esistere alcune relazioni.

*Esempio di relazione* : Una riga di una tabella A può fare riferimento ad un’altra riga di un’altra tabella B, e ciò può essere espresso inserendo la chiave primaria (indice che permette di riconoscere univocamente quella riga rispetto a tutte le altre) della riga di B tra i dati di quella di B.

CouchDB utilizza il protocollo HTTP e il formato dati JSON.

Le uniche chiavi obbligatorie sono *\_id* (serve per identificare univocamente il documento) e *\_rev* (utilizzato per la gestione di revisioni/modifiche), \_rev viene aggiornato dopo ogni modifica.

Vantaggi database NOSQL

* *Elevata velocità computazionale,* anche al crescere del volume dei dati, legata alla mancanza di operazioni di aggregazione dei dati (no join)
* *Riduzione significativa dei tempi di sviluppo,* grazie alla definizione di logiche di lettura dati molto più semplici rispetto a quelle da scrivere con database relazionali
* *Supporto per la scalabilità orizzontale,* garantire tempi di risposta al crescere del carico con l’aggiunta di nuovi server
* *Elevato livello di disponibilità del servizio*
* *Schemaless,* non vi è necessità di una definizione formale e rigida del suo contenuto.

Oauth (C)

WebSocket vs SocketIO (F)

Message Queue: AMQP e MQTT (C)

Promises