Docker, docker-compose e virtualizzazione
Sistemi innovativi per la distribuzione del software

Ing. F. Mereu

Seminario per il Corso: Big Data Architecture (Prof. P. Nesi), 2024-25

Distributed Data Intelligence and Technologies Lab
Distributed Systems and Internet Technologies Lab

docker

UNIVERSITÀ DEGLI STUDI FIRENZE

DINFO
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DISIT
DISTRIBUTED SYSTEMS AND INTERNET TECHNOLOGIES LAB

Indice

- Introduzione
- Storia
- Motivazioni
- Strumenti utilizzati
- Creare immagini e pubblicarle
- Docker-compose: tanti componenti insieme
- Docker-compose in azione
- Kubernetes
- Riferimenti

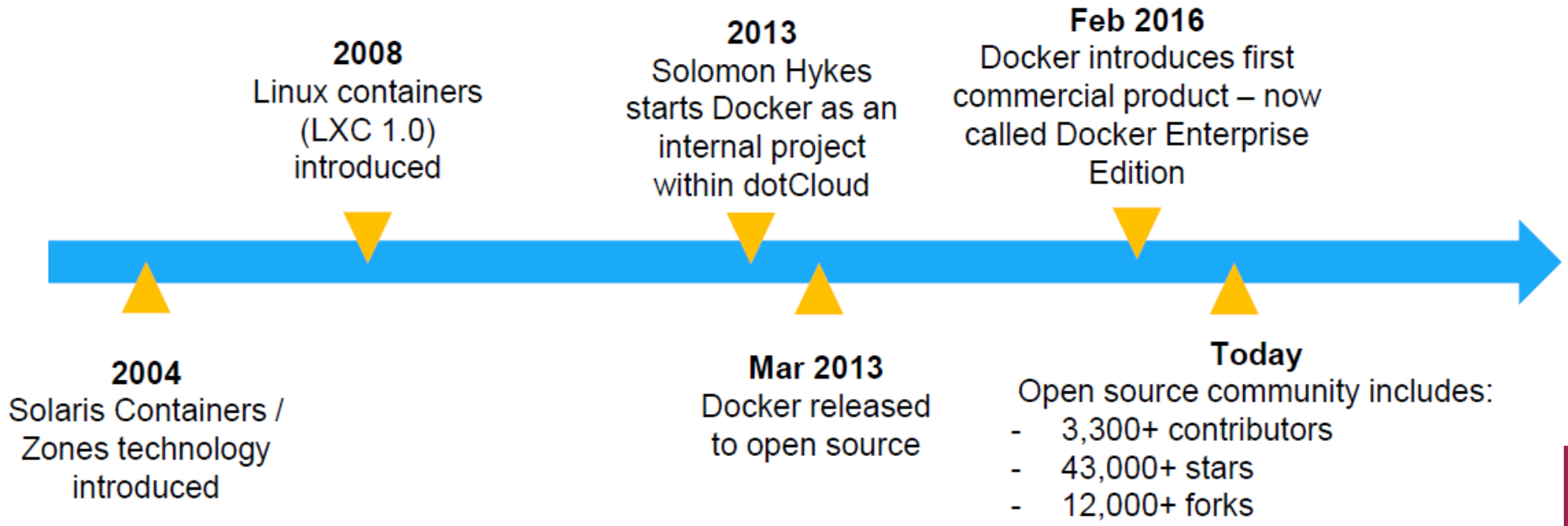
Introduzione

Docker è un insieme di prodotti platform as a service (PaaS) che usa la virtualizzazione a livello di sistema operativo (Linux) per eseguire il deploy di applicazioni in pacchetti chiamati in gergo **Container**.

Questi container hanno un comportamento simile alle macchine virtuali, ma si differenziano per alcune particolarità che li rendono adatti ad un deploy multiplo su singole macchine (anche dei Raspberry, ad esempio) o server dedicati.

Storia

History of Docker



The evolution of containers

Container technology has come a long way from its chroots, starting with Google's exploration into cgroups and working up into widespread organizational adoption.

2004-07
Google uses containers (cgroups) at scale

MARCH 2013
Docker hits the scene: Containerization platform based on Linux; integrates natively with other OSes: Windows, macOS

2013 RESEARCH
30 containerization vendors vie for users (451 RESEARCH)

NOVEMBER 2014
Canonical launches LXD: Built on LXC with new features and functionality

DECEMBER 2014
CoreOS introduces rkt containers as Docker alternative

APRIL 2015
CoreOS first to provide commercial Kubernetes offering

JULY 2015
Kubernetes 1.0 released: Google gives Kubernetes to Cloud Native Computing Foundation (CNCF) for development

JANUARY 2008
cgroups merge into Linux kernel

AUGUST 2008
LXC released: Linux containers, based on cgroups, isolate resources without a VM

OCTOBER 2013
CoreOS Linux (now Container Linux by CoreOS) released

JUNE 2014
Docker 1.0 is released

JUNE 2014
Kubernetes project announced: A now widely popular open source container orchestrator and scheduler created by Google

JUNE 2015
Open Container Project created AKA Open Container Initiative (OCI): Establishes common container standards

MARCH 2017
Pivotal integrates with Kubernetes and Cloud Foundry, dubbed Kubo; Docker donates containerd container runtime to CNCF

APRIL 2017
Portworx opens door for big data with PX-Enterprise update; Microsoft enables orgs to run Linux containers on Windows

JULY 2017
Microsoft Azure Container Instances provides container management for Linux

SEPTEMBER 2017
Rancher adopts Kubernetes for container orchestration

$1B

2017 RESEARCH
Container market tops $1 billion, with 125 application container vendors (451 RESEARCH)

APRIL 2018
Kubernetes adds API Aggregation, improves support for Windows nodes and Linux, audits API stability in 1.10 release

2018 RESEARCH
90% of infrastructure and operations groups report testing or deploying containers (GARTNER)

JULY 2019
IBM acquires Red Hat for Red Hat OpenShift Kubernetes integration

NOV 2019
Mirantis acquires Docker Enterprise from Docker, Inc.

2022 PREDICTION
More than 50% of containerized workloads will span across hybrid environments, up from less than 20% in 2020 (GARTNER)

JUNE 2017
Kubernetes adds stateful support: Stateless apps forget session information; stateful apps hold onto it

2017 RESEARCH
Less than 20% enterprise container adoption (GARTNER)

OCTOBER 2017
Cloud Foundry Container Runtime launches

OCTOBER 2017
Docker adds native Kubernetes support alongside swarm mode

NOV 2018
VMware purchases Heptio for Kubernetes integration

JUNE 2019
Amazon EKS and Microsoft AKS become generally available

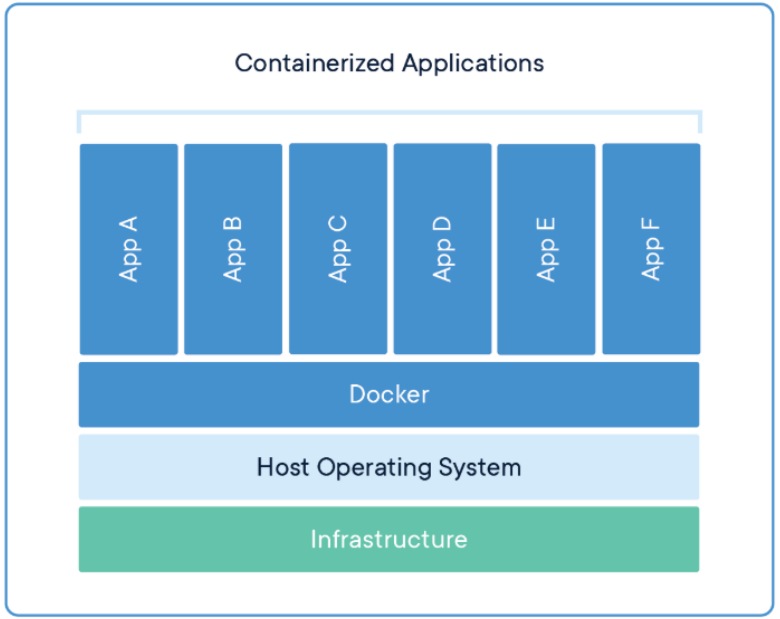
2019 RESEARCH
95% of new applications use containers (451 RESEARCH)
In addition, containers run in production at about 30% of organizations—a number expected to grow to more than 75% by 2022 (GARTNER)

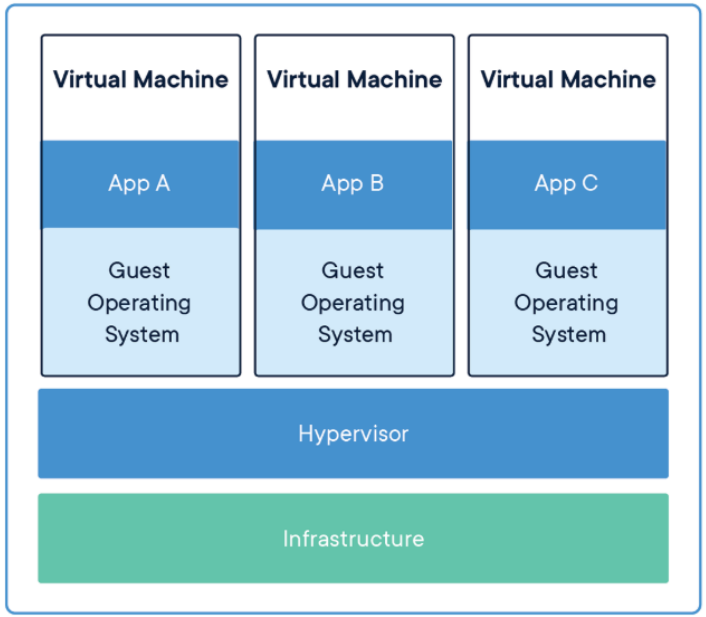
2023 PREDICTION
Application container software market to top $5.5 billion (451 RESEARCH)

2025 PREDICTION
Container market value to top $8 billion (GRAND VIEW RESEARCH)

$8B

SOURCE: EMILY MELL FOR TECHTARGET; DESIGN: LINDA KOURY; LAPTOP ART: LIGHTCOME/GETTY IMAGES





CONTAINERS

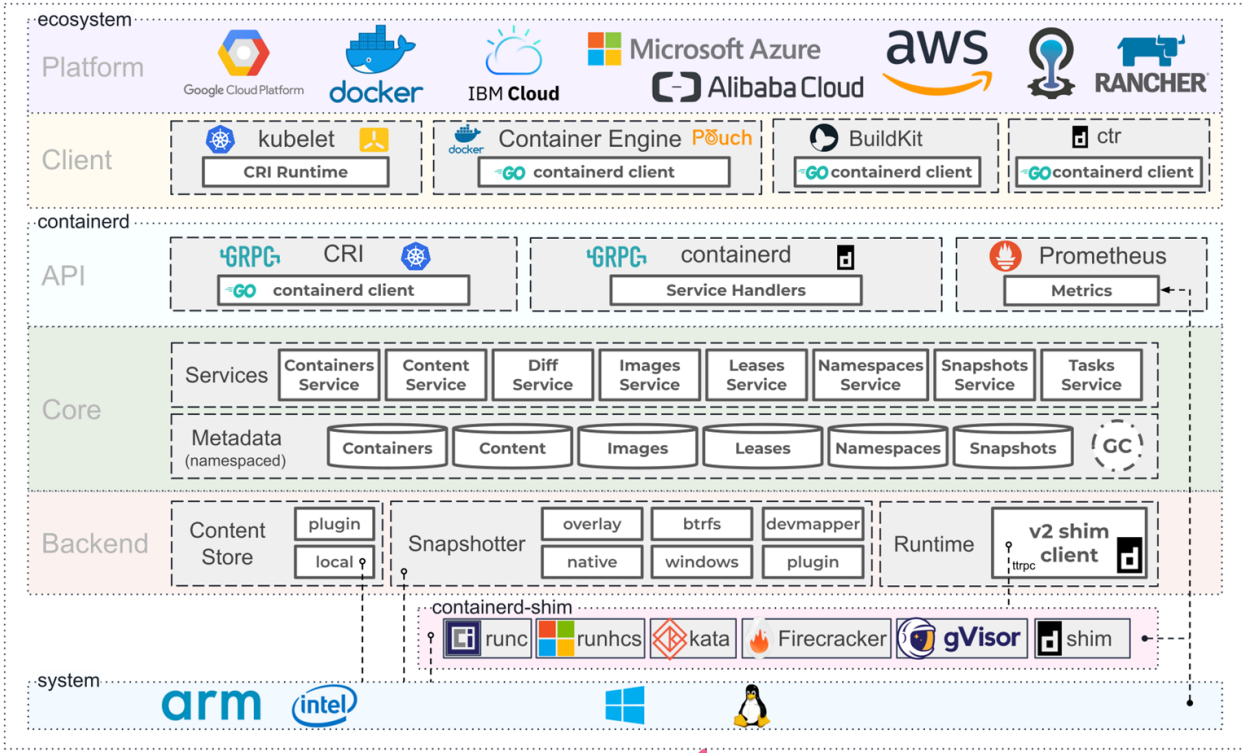
Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), can handle more applications and require fewer VMs and Operating systems.

VIRTUAL MACHINES

Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, the application, necessary binaries and libraries – taking up tens of GBs. VMs can also be slow to boot.

containerd - An industry-standard container runtime with an emphasis on simplicity, robustness and portability

containerd *is available as a daemon for Linux and Windows. It manages the complete container lifecycle of its host system, from image transfer and storage to container execution and supervision to low-level storage to network attachments and beyond.*



Motivazioni

- Nel mondo odierno, la maggior parte dei processi sono spostati sul cloud, o comunque sono decentralizzati.
- Spesso abbiamo bisogno di microservizi piuttosto che macchine intere, perciò un'intera macchina, virtuale o meno, tende ad essere eccessivamente costosa, in termini sia di risorse utilizzate sia di risorse economiche; non c'è bisogno di avere disponibile tutte le funzionalità di un computer quando per esempio abbiamo solo bisogno di un servizio web.
- Questi microservizi tendono inoltre ad essere usa e getta e debolmente legati ad altri eventuali microservizi.

Motivazioni - cont.d

- Per arginare il più possibile l'assenza di uniformità delle macchine che eseguono il software scritto dagli sviluppatori, si rende necessario creare un sistema che riesca ad eseguire i nostri software con risultati identici su ogni hardware.
- Vogliamo inoltre abbandonare il mondo delle applicazioni monolitiche.
- L'hardware è generalmente diverso per ogni macchina, così come l'ambiente dove il software viene eseguito, e quindi risolvere il problema non è banale.
- In altre parole, umoristicamente…

Strumenti utilizzati

- Docker (Desktop)
- Linux
- Console (probabilmente bash)
- Qualunque cosa stiate comunque utilizzando per sviluppare il vostro software
- Un account su hub.docker.com (anche gratuito)
- Una connessione ad internet

Conoscenze gradite

- Ciò che dovreste comunque conoscere per sviluppare le vostre app/servizi/etc.
- Uso di porte, indirizzi ip, risoluzione di nomi, etc
- GitHub (per buona prassi, non è strettamente legato a docker)
- Bash (le immagini possono girare anche su un'immagine di Windows ma queste poi sarebbero utilizzabili solo su Windows)

Uso della CLI di Docker

Dopo aver installato docker, possiamo interagire con il suddetto sistema tramite CLI, usando il comando **docker**. Docker viene trattato come un daemon, ergo è un processo in background, ergo può essere spento o acceso. Da spento il comando docker ovviamente non funziona; si può quindi utilizzare il comando **dockerd** per attivarlo; o anche altri sistemi più generici per abilitare un servizio/daemon.

Si può fare tutto quello che mostreremo da interfaccia grafica, ma questa potrebbe non essere disponibile se lavoriamo su un server dedicato.

Lanciare container singoli

Sintassi

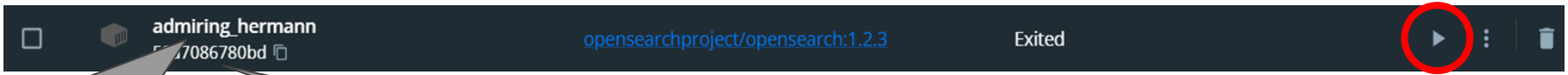
```
docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]
```

Esempio

detached

```
>docker run --name website -d ubuntu/apache2
```

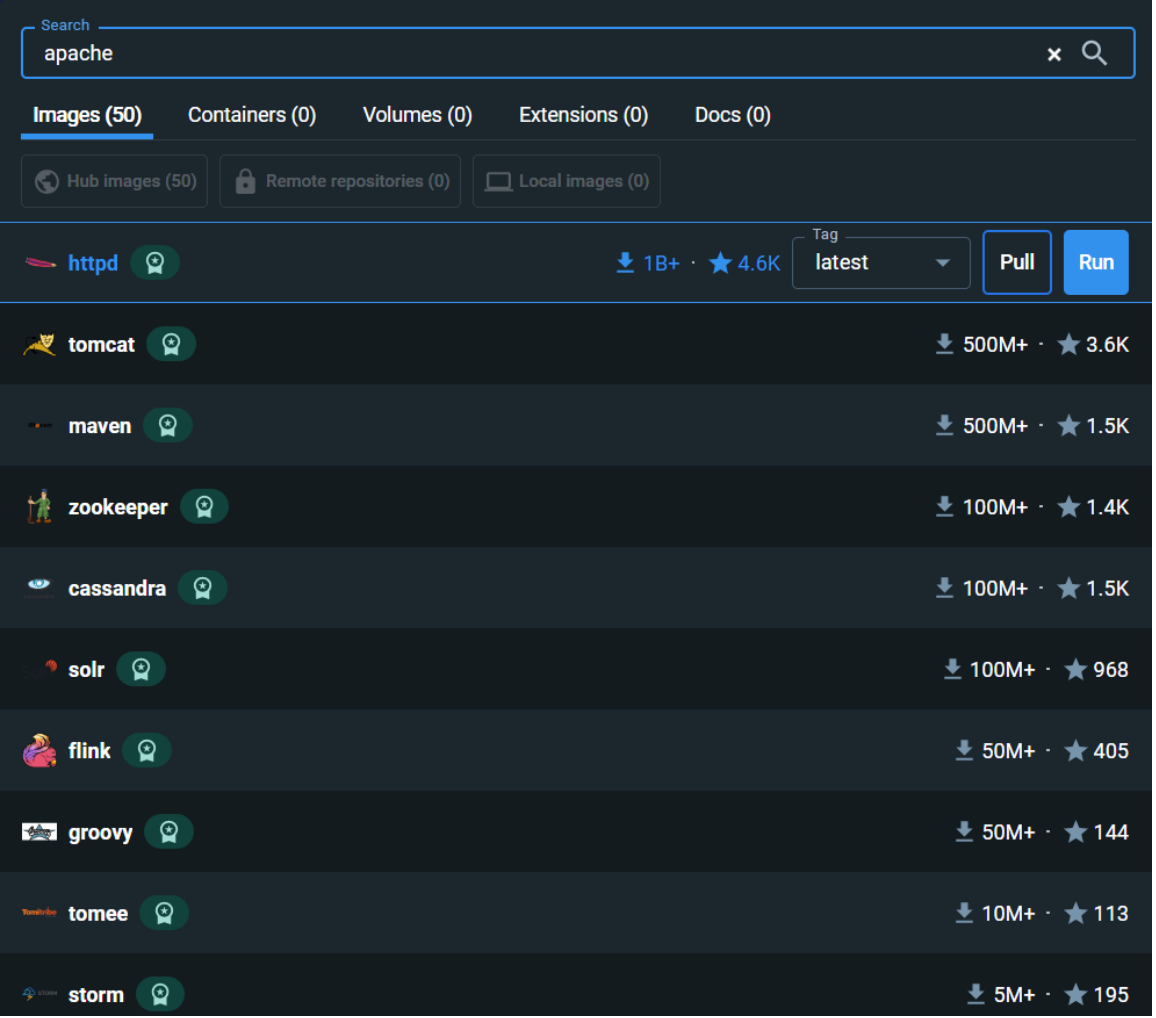
Si può anche fare tramite interfaccia grafica in Docker Desktop (nell'immagine ne avviamo una già esistente)

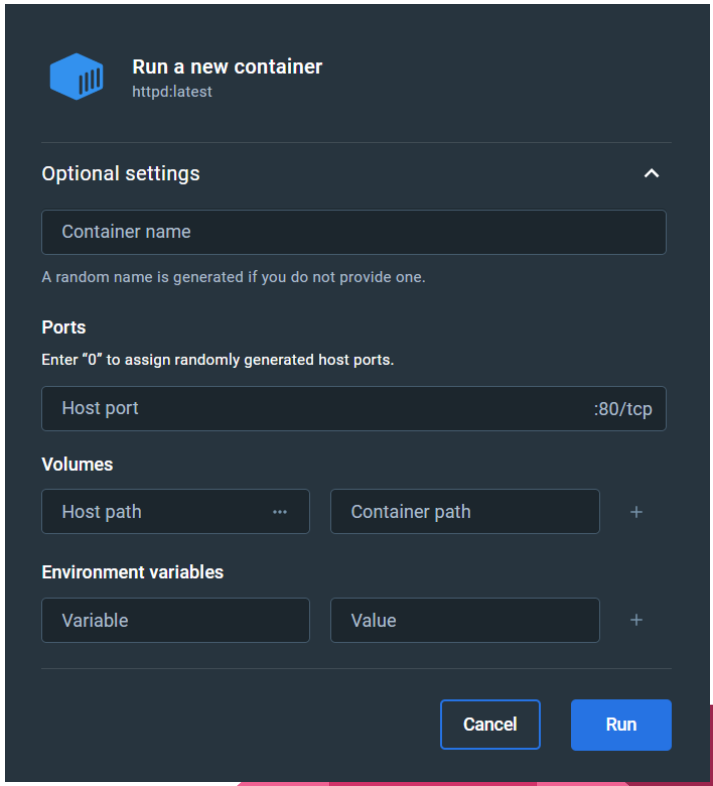


Se non assegnamo un nome, ne sarà generato uno casuale

Ogni container ha comunque un suo id univoco

Lancio di container da GUI





UNIVERSITÀ DEGLI STUDI FIRENZE
DINFO DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
DISIT DISTRIBUTED SYSTEMS AND INTERNET TECHNOLOGIES LAB

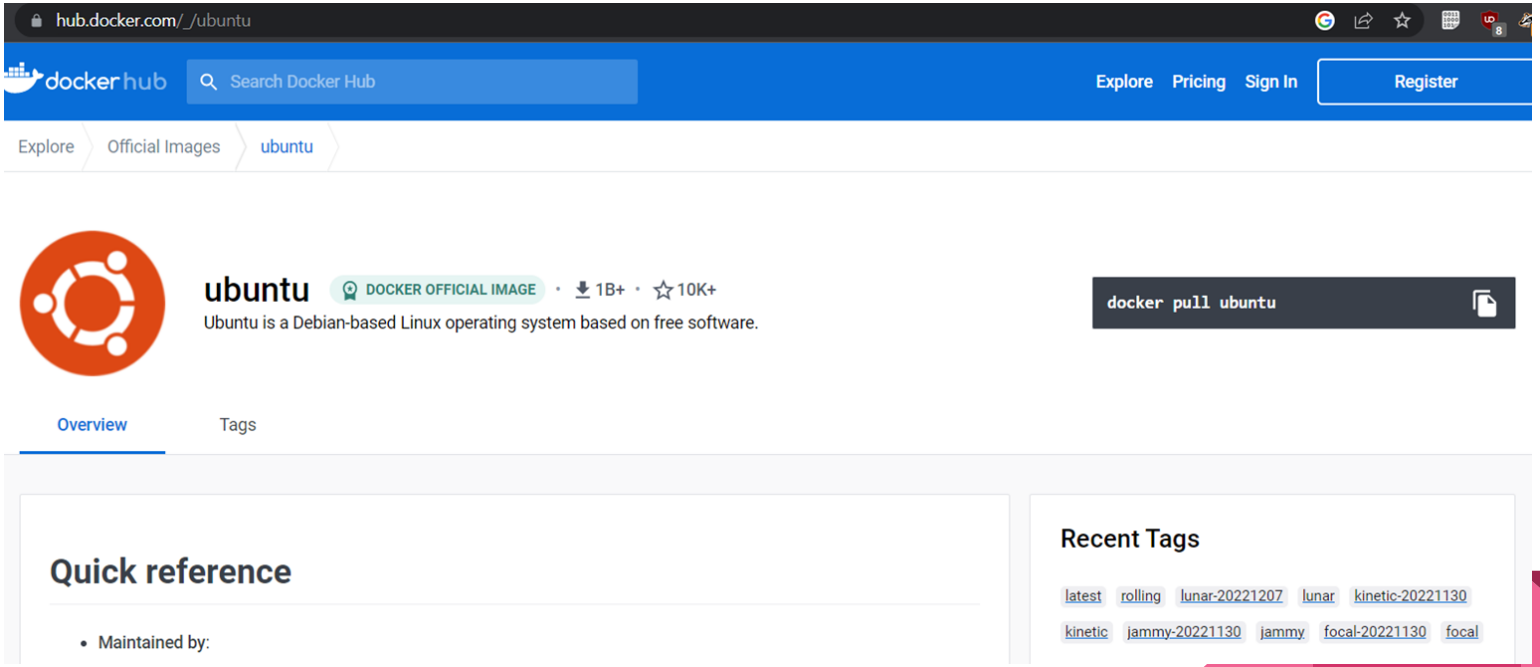
Presenza locale delle immagini

Un’immagine può non essere localmente accessibile; potrebbe non essere stata sviluppata in locale, potrebbe non essere stata ancora scaricata dalla repository o potrebbe essere privata. Docker proverà a scaricare l’immagine per il suo utilizzo (fallendo, nel terzo caso).

Nel comando precedente non ci siamo curati di verificare la presenza dell’immagine localmente; Docker scarica autonomamente l’immagine, se esiste, da docker hub, ammesso che non sia già presente.

Esempio per il recupero di un’immagine

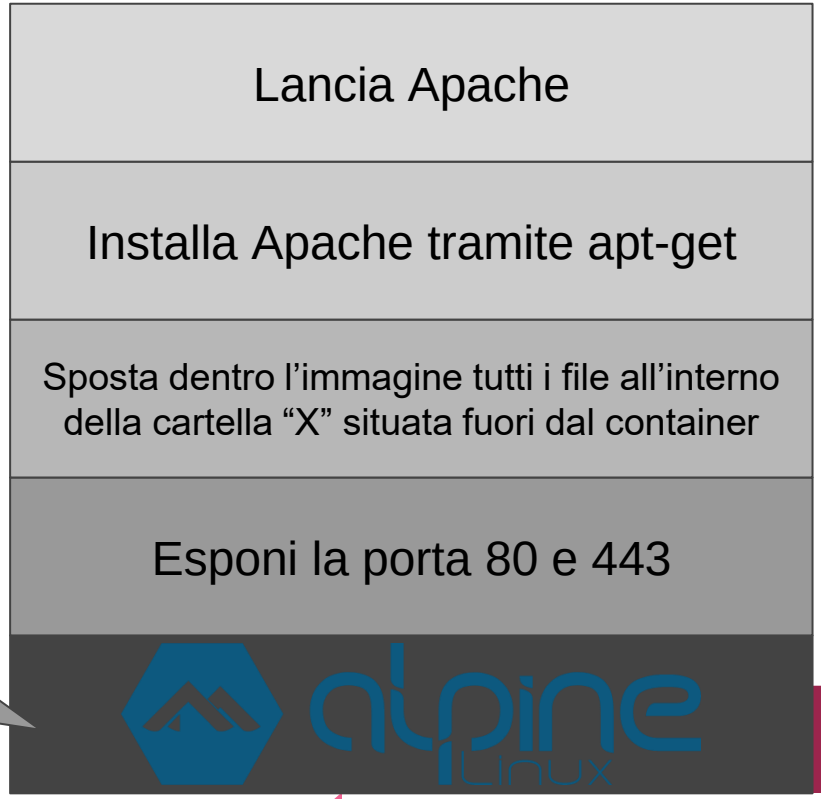
```
docker pull ubuntu
```



Struttura di un’immagine

Le immagini vengono generate a partire da una base, sulla quale vengono applicati dei layer (strati) dove eseguiamo setup aggiuntivi fino ad arrivare ad un prodotto con specifiche e operazioni desiderate.

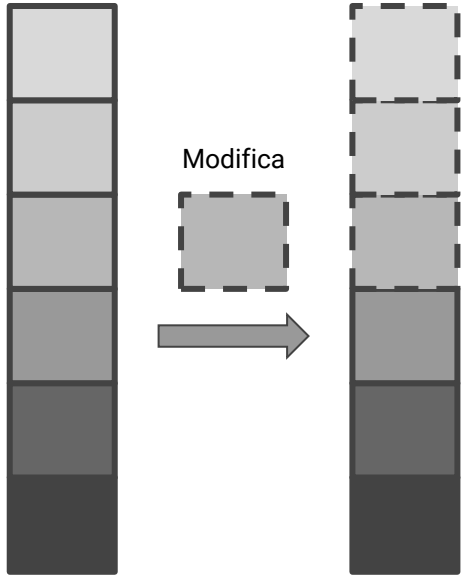
A destra vediamo un esempio:



Distribuzione di Linux molto leggera

Layer delle immagini

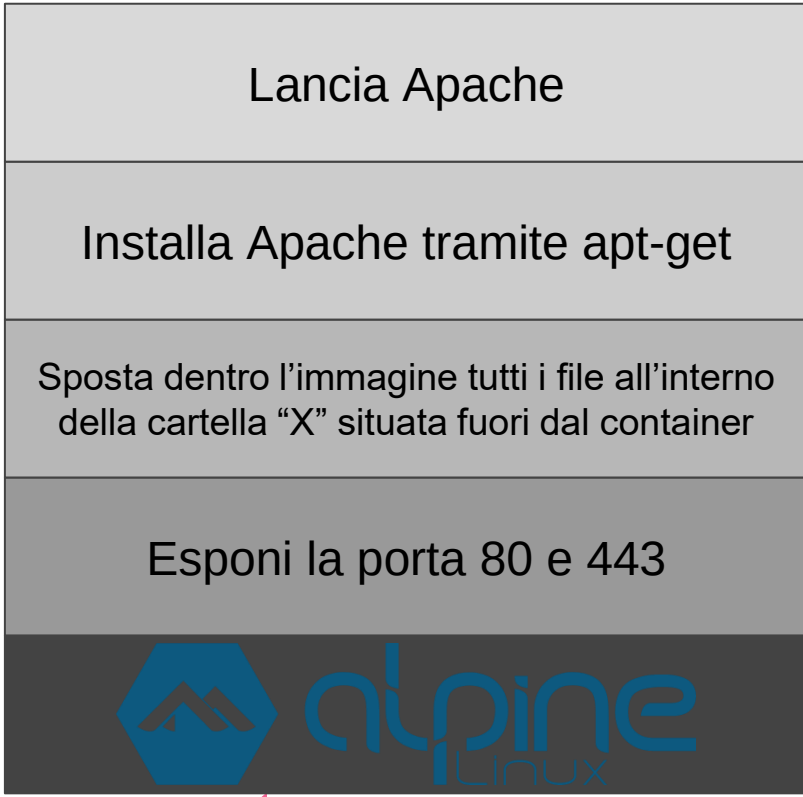
- Quando apportiamo delle modifiche alla nostra immagine, il generatore delle immagini non ricostruirà tutto da capo.
- Utilizzerà invece quanti più layers possibili già esistenti, e creerà soltanto quelli che includono la modifica e i successivi.
- Questo sistema riduce sia il tempo che lo spazio su disco occupato per la generazione delle immagini.
- Il risultato può essere graficamente interpretato a destra.



Dockerfile (senza estensione)

```
# con il cancelletto indichiamo i commenti
FROM alpine:latest

# tcp/udp se necessario, default è tcp
EXPOSE 80, 443

# spostiamo il contenuto della nostra
# posizione dentro la cartella UserFiles
COPY . /UserFiles

# Esegui il comando (o il programma)
RUN apt-get install apache2

# Esegui da console con i seguenti
# comandi e parametri
CMD [“apache2ctl”, “-D”, “FOREGROUND”]
```

Immagine



Attenzione ai flag!

Vedremo, usando la CLI, alcuni flag che permettono di lanciare dei comandi con parametri e comportamenti particolari.

Quelli non chiari verranno spiegati, quelli auto esplicativi saranno dati per scontati, ma alcuni sono dei “false friends”, specie quelli da pochi caratteri.

Se non avete esperienza con i comandi che dovete utilizzare, si raccomanda di utilizzare la versione estesa di ogni flag, che tende ad essere più chiara, e di consultare il testo di aiuto raggiungibile aggiungendo “--help” ad un qualsiasi comando valido; vedrete la pagina di aiuto e il resto del comando verrà ignorato.

Visualizzare le risorse (immagini e container)

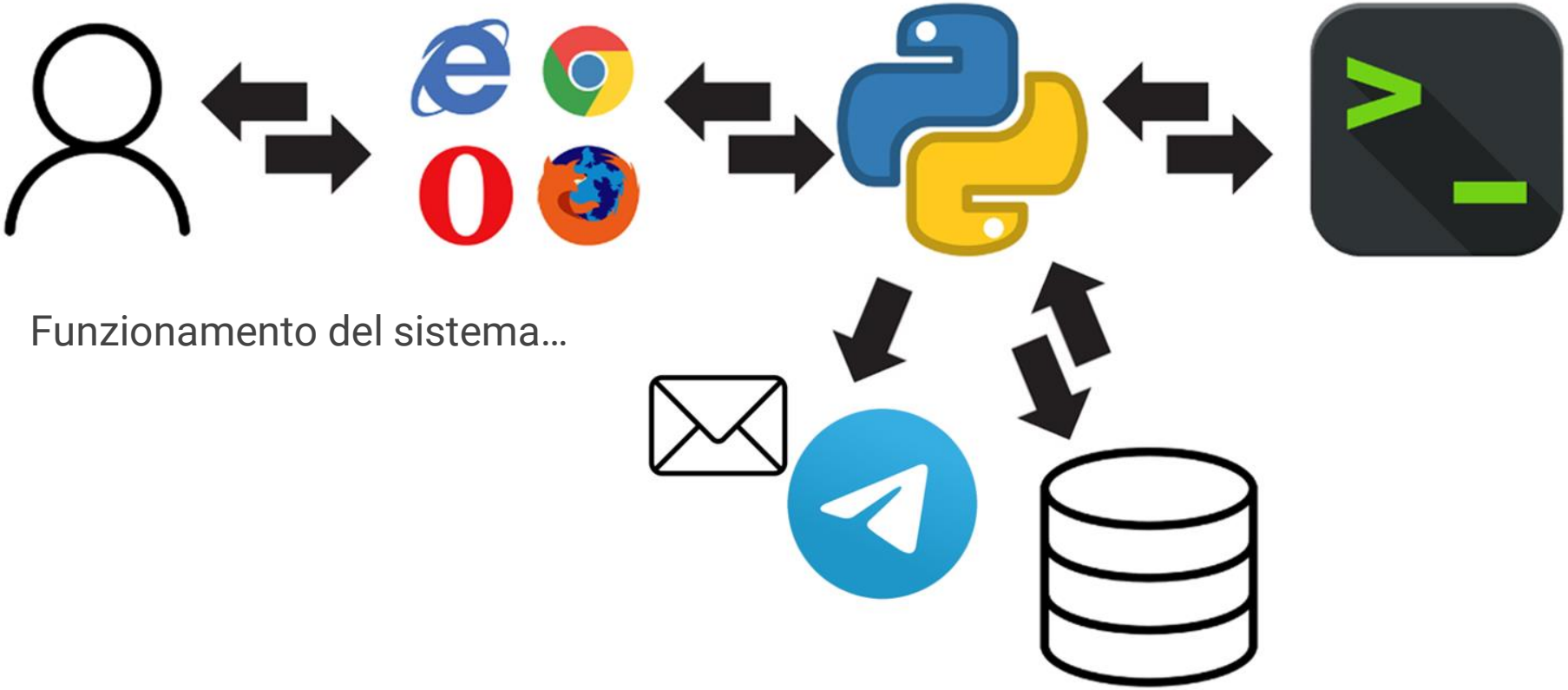
```
>docker images

REPOSITORY                TAG                 IMAGE ID            CREATED             SIZE
<none>                    <none>              77af4d6b9913        19 hours ago        1.089 GB
committ                   latest              b6fa739cedf5        19 hours ago        1.089 GB
postgres                  9                   746b819f315e        4 days ago          213.4 MB
postgres                  9.3                 746b819f315e        4 days ago          213.4 MB
postgres                  9.3.5               746b819f315e        4 days ago          213.4 MB
postgres                  latest              746b819f315e        4 days ago          213.4 MB

docker container ls

CONTAINER ID   IMAGE           COMMAND     CREATED      STATUS         PORTS                                 NAMES
a562c08b9ebd   ubuntu:latest   "bash "     9 days ago   Up 1 second    0.0.0.0:80->80/tcp                    tesi-app-1
87727f6ae91f   mysql:latest    ""          9 days ago   Up 3 seconds   33060/tcp, 0.0.0.0:32000->3306/tcp    tesi-db-1
```

Visualizzare le risorse utilizzate (CPU, RAM, disco, rete…)

```
>docker stats


CONTAINER ID   NAME      CPU %     MEM USAGE / LIMIT     MEM %     NET I/O          BLOCK I/O   PIDS
88707298c7e1   app       0.01%     31.42MiB / 7.614GiB   0.40%     38.4kB / 260kB   0B / 0B     5
5e969909ec95   db        0.19%     390.7MiB / 7.614GiB   5.01%     262kB / 37.1kB   0B / 0B     38
```

Snap4Sentinel - utility per monitoraggio e controllo dei container

Disk Data I/0	CPU%	ContainerID	Image	Mem%	Mem Used / Mem Max	Container Name	Network I/O	Ports	Running for / Exited since	Size	Status	Reboot	Is alive test	Last tests results
248MB / 7.33MB	0.60%	33a8aab0e853	kartoza/geoserver:2.20.0	5.79%	1.818GiB / 31.41GiB	192168118_geoserver_1	8.6kB / 0B	8443/tcp, 0.0.0.0:8600->8080/tcp, :::8600->8080/tcp	3 weeks ago	0B	running	Reboot	Run tests...	Success
1.83GB / 6.4MB	17.78%	c676420eda97	apache/nifi:1.16.2	3.51%	1.102GiB / 31.41GiB	192168118_nifi_1	8.56kB / 42B	8000/tcp, 8080/tcp, 0.0.0.0:1030->1030/tcp, :::1030->1030/tcp, 8443/tcp, 0.0.0.0:9090->9090/tcp, :::9090->9090/tcp, 10000/tcp	3 weeks ago	0B	running	Reboot	Run tests...	No tests found
34.3MB / 0B	0.00%	ae510cb2af86	disitlab/dashboard-backend:v1	0.10%	30.91MiB / 31.41GiB	192168118_dashboard-backend_1	90.6kB / 33.8kB		3 weeks ago	0B	running	Reboot	Run tests...	No tests found



Funzionamento del sistema…

I nostri container non hanno memoria persistente

I container non hanno a disposizione un vero “disco” dove poter salvare in modo permanente i dati generati e utilizzati; una volta che i container sono eliminati, tutti i dati sono persi.

In una macchina virtuale, si concede all’immagine dello spazio, e questo spazio sarà quello a disposizione come “hard disk” all’interno della macchina.

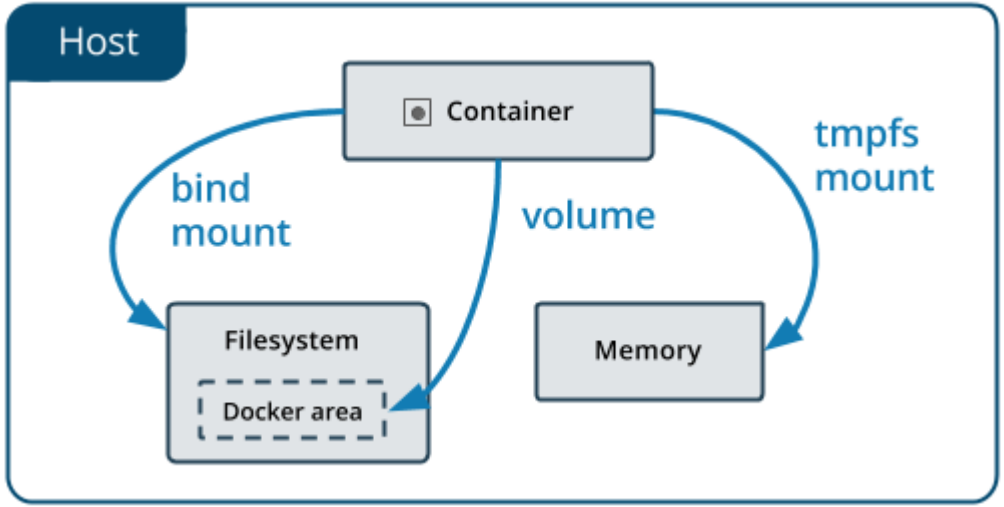
In docker abbiamo delle soluzioni differenti a questo problema.

Volumi, Bind Mounts e Temporary Filesystem

In docker abbiamo quindi 3 soluzioni:

- Volumi
- Bind Mounts
- TeMPorary File System (tmpfs)

Queste hanno proprietà diverse e le vedremo nelle seguenti slides.



Bind Mounts

In questa configurazione, il file system del container viene direttamente montato su quello dell'host, come percorso assoluto. Questo è un sistema esistente fin dagli albori di Docker, e seppur sprovvisto di molte funzionalità, è anche molto performante in termini di velocità.

È necessario prestare estrema attenzione; questo sistema offre accesso a file e cartelle indiscriminato; è possibile alterare i file e le cartelle dell'host in modo catastrofico.

"Continua sotto"

```
docker run -d -it --name devtest --mount \
type=bind,source="$(pwd)"/target,target=/app nginx:latest
```

pwd significa print working directory, siamo su bash

Volumi

I volumi sono il sistema preferito da Docker, e sono completamente gestiti da Docker. Presentano dei vantaggi rispetto ai Bind Mounts, ad esempio possono essere inizializzati da un container, possono essere gestiti da un driver che permette loro di essere memorizzati in remoto o in cloud, o possono anche essere crittografati o comunque possedere funzionalità aggiuntive.

I volumi su Docker Desktop sono più performanti dei Bind Mounts su Mac e Windows

```
docker run -d --name devtest --mount source=myvol2,target=/app nginx:latest
```

Temporary File System

Disponibili solo su Linux, tmpfs è un passo indietro in termini di persistenza dei dati; il “volume” esiste soltanto in memoria e non viene salvato da nessuna parte.

Una volta che il container viene fermato, i file contenuti al suo interno sono persi.

Questa configurazione è particolarmente utile quando il non salvare i dati generati dal container è una priorità, ad esempio quando si gestiscono dati sensibili come le credenziali di accesso a qualche servizio.

```
docker run -d -it --name tmptest --mount type=tmpfs,destination=/app nginx:latest
```

Isolamento e collegamenti

Prima abbiamo detto che i container sono isolati gli uni dagli altri.

Questo però non significa che i container non possano comunicare tra di loro.

Docker offre la possibilità ai container di vedere altri container sullo stesso host (e anche altrove, se siamo in grado di raggiungere il container, per esempio conoscendo l'ip locale dell'host oppure se esiste un nome risolvibile da un DNS), e di esporre porte che l'host stesso farà gestire al container che ne chiede accesso.

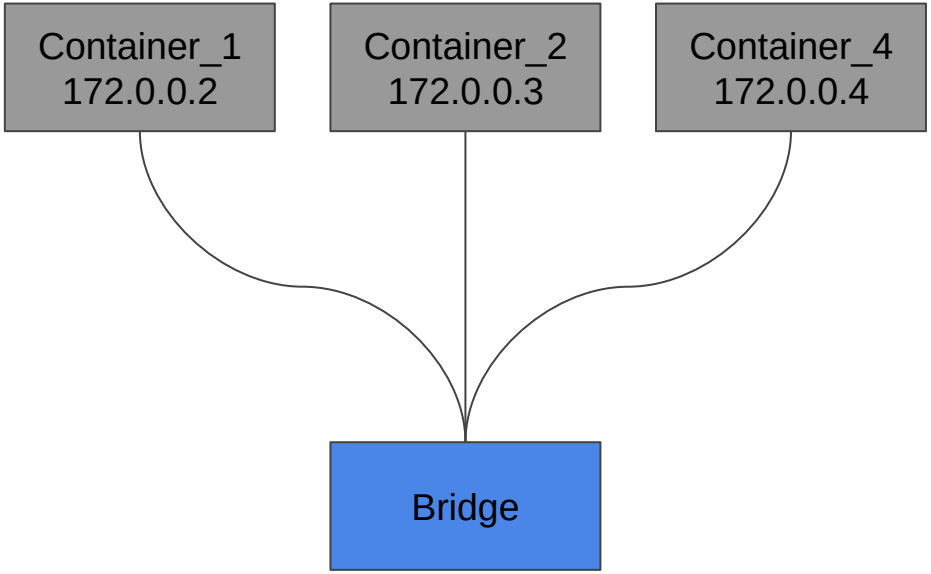
Networking

Il comportamento di default prevede che tutti i container siano connessi allo stesso bridge.

Parliamo dello stesso dispositivo che aggrega reti fisiche, ma ovviamente qui è virtuale.

I container sono raggiungibili tramite l'indirizzo ip, ma attualmente non possiamo risolvere i nomi degli **altri** container (vediamo comunque noi stessi).

Ovviamente possiamo implementare scelte differenti, ad esempio isolando alcuni containers dagli altri.




```
> ping 172.0.0.3  # funziona
```

Collegare containers

Per collegare containers in una stessa rete useremo dei bridge definiti dall'utente (e perciò da noi)

In questi bridge possiamo definire le proprietà e i componenti che sono collegati a tali bridge

Con questi bridge è possibile raggiungere gli altri container collegati tramite indirizzo ip e tramite il loro nome (sarà proprio il nome del container, se non ne abbiamo assegnato uno ricordiamoci che questo verrà generato automaticamente)

Possiamo inoltre collegare i container a "niente", il che li rende scollegati dalla rete

Esempio

Detached (in background)
Interactive (nel senso che possiamo scriverci)
TTY (teletypewriter)

```
docker network creat  driver br

docker run -dit --name alpine1 --network testnet  ine ash

docker run -dit --name alpine2 --network testnet alpine ash

docker container attach alpine1

$ ping -c 2 alpine2

# continua…
```

ash è il bash di Alpine

“-c 2” significa “esegui solo 2 ping”

Esempio

YMMV

```
PING alpine2 (172.18.0.3): 56 data bytes

64 bytes from 172.18.0.3: seq=0 ttl=64 time=0.076 ms

64 bytes from 172.18.0.3: seq=1 ttl=64 time=0.091 ms


--- alpine2 ping statistics ---

2 packets transmitted, 2 packets received, 0% packet loss

round-trip min/avg/max = 0.076/0.083/0.091 ms
```

Domande?

Creiamo un’immagine e pubblichiamola

mostra demo

Separazione tra le responsabilità

Un container Apache dovrebbe gestire un database?

Un container MySQL dovrebbe gestire un sito web?

Per entrambe le domande la risposta è no, ma anche pensando ad un semplicissimo problema dove abbiamo bisogno sia di una piattaforma per un server sia di una piattaforma per un database un singolo container non è una buona soluzione.

Possiamo risolvere il problema con le soluzioni viste fino ad ora, ma è una soluzione scomoda quando abbiamo container multipli.

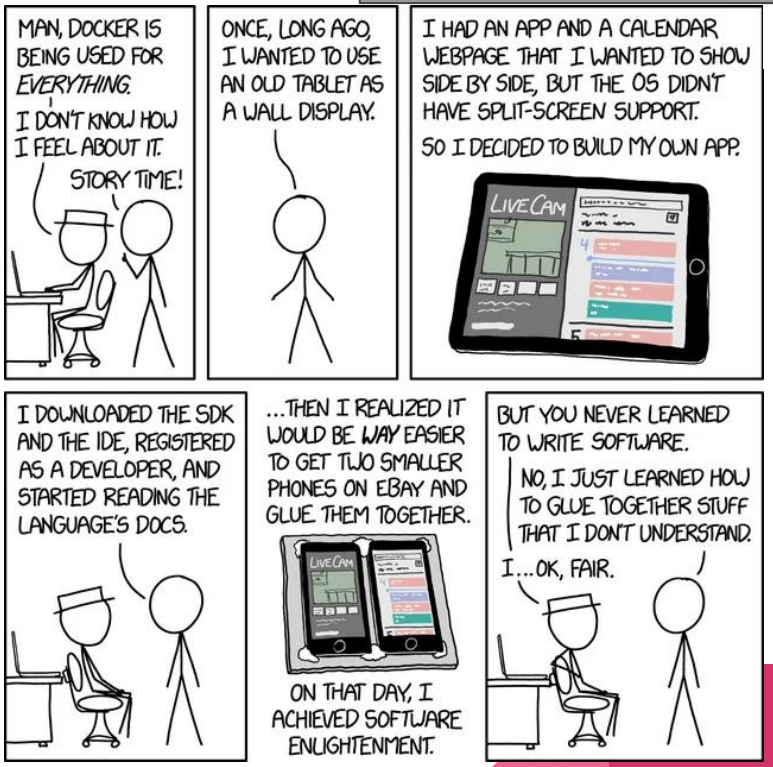
Da docker a docker-compose

Nelle ultime versioni è “docker compose”, senza trattino

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application’s services. Then, with a single command, you create and start all the services from your configuration.

Continuous Integration

Compose works in all environments: production, staging, development, testing, as well as CI workflows. It also has commands for managing the whole lifecycle of your application:



YAML - YAML Ain't Markup Language

Per definire container multipli, utilizziamo il formato YAML usato per la serializzazione di dati.

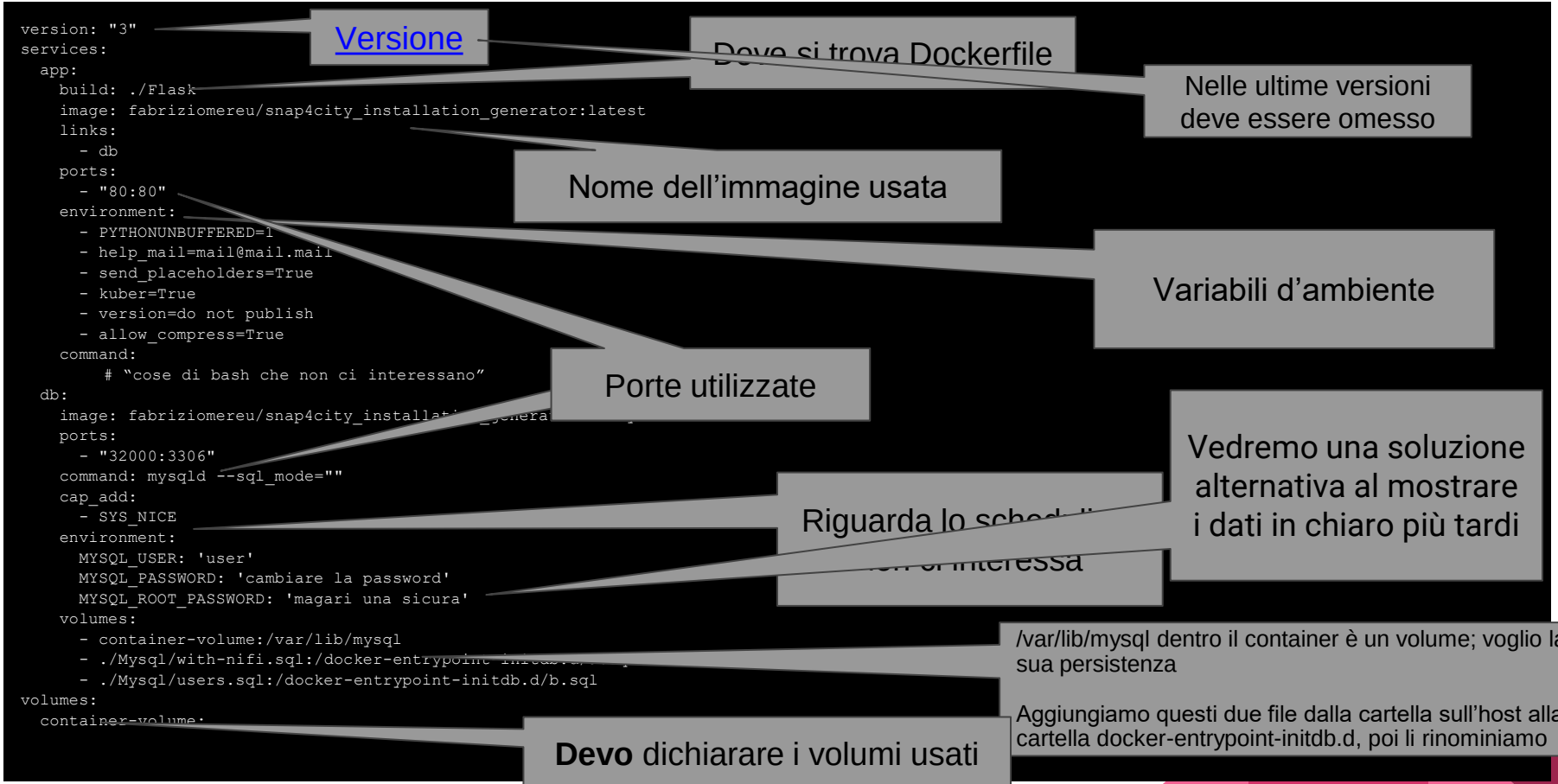
È possibile passare a JSON e XML e viceversa, ma questo è il sistema utilizzato da docker-compose; perciò è quello che useremo.

```
invoice: 34843
date   : 2018-09-13
bill-to: &id001
   given  : Chris
   family : Dumars
   address:
       city    : Royal Oak
       state   : MI
       postal  : 48046
ship-to: *id001
```

Idea di fondo

Per gestire tutti i container nel modo più semplice possibile ci comporteremo nel seguente modo

- Decidiamo di quali immagini abbiamo bisogno
- Decidiamo la configurazione di cui hanno bisogno le immagini
- Decidiamo i volumi per le immagini
- Decidiamo i collegamenti di rete tra i vari container


version: "3"
services:
app:
build: ./Flask
image: fabriziomereu/snap4city_installation_generator:latest
links:
- db
ports:
- "80:80"
environment:
- PYTHONUNBUFFERED=1
- help_mail=mail@mail.mail
- send_placeholders=True
- kuber=True
- version=do not publish
- allow_compress=True
command:
"cose di bash che non ci interessano"
db:
image: fabriziomereu/snap4city_installat
ports:
- "32000:3306"
command: mysqld --sql_mode=""
cap_add:
- SYS_NICE
environment:
MYSQL_USER: 'user'
MYSQL_PASSWORD: 'cambiare la password'
MYSQL_ROOT_PASSWORD: 'magari una sicura'
volumes:
- container-volume:/var/lib/mysql
- ./Mysql/with-nifi.sql:/docker-entrypoint
- ./Mysql/users.sql:/docker-entrypoint-initdb.d/b.sql
volumes:
container-volume:
Versione
Deve si trova Dockerfile
Nelle ultime versioni deve essere omesso
Nome dell'immagine usata
Variabili d'ambiente
Porte utilizzate
Vedremo una soluzione alternativa al mostrare i dati in chiaro più tardi
Riguarda lo sched
non ci interessa
/var/lib/mysql dentro il container è un volume; voglio la sua persistenza
Aggiungiamo questi due file dalla cartella sull'host alla cartella docker-entrypoint-initdb.d, poi li rinominiamo
Devo dichiarare i volumi usati

Compose multipli

Talvolta vogliamo combinare molteplici docker-compose; nel contesto di insiemistica, sia che questi abbiano o non abbiano intersezione (sarà più chiaro negli esempi successivi) tra i loro container e le loro configurazioni.

Per queste evenienze, possiamo utilizzare il flag -f di docker-compose, usando come parametri i vari docker-compose da fondere insieme.

Attenzione: l'ordine dei vari compose è importante se questi hanno intersezione non vuota; si parte dal primo compose fornito, poi si aggiunge il secondo, terzo e così via.

```
docker-compose -f compose-1.yml -f compose-2.yml > docker-compose.yml
```

Esempio con intersezione non vuota

```
web:
  image: example/my_web_app:latest
  depends_on:
    - db
    - cache
db:
  image: postgres:latest
cache:
  image: redis:latest
```

```
web:
  build: .
  volumes:
    - '.:/code'
  ports:
    - 8883:80
  environment:
    DEBUG: 'true'
db:
  command: '-d'
  ports:
    - 5432:5432
cache:
  ports:
    - 6379:6379
additional:
  image: alpine:latest
```

Risultato

```
name: lezione12-11
services:
  additional:
    image: alpine:latest
    networks:
      default: null
  cache:
    image: redis:latest
    networks:
      default: null
    ports:
    - mode: ingress
      target: 6379
      published: "6379"
      protocol: tcp
  db:
    command:
    - -d
    image: postgres:latest
    networks:
      default: null
    ports:
    - mode: ingress
      target: 5432
      published: "5432"
      protocol: tcp
  # continua a destra, questo è un commento senza effetto
```

```
  web:
    build:
      context: C:\some-path
      dockerfile: Dockerfile
    depends_on:
      cache:
        condition: service_started
      db:
        condition: service_started
    environment:
      DEBUG: "true"
    image: example/my_web_app:latest
    networks:
      default: null
    ports:
    - mode: ingress
      target: 80
      published: "8883"
      protocol: tcp
    volumes:
    - type: bind
      source: C:\some-path
      target: /code
      bind:
        create_host_path: true
networks:
  default:
    name: lezione12-11_default
```

Osservazioni

Come possiamo vedere il risultato finale è un po’ più complesso della somma dei singoli elementi: parte del motivo è che il comando restituisce una risposta verbosa:

- La definizione delle porte esposte è molto più esplicita.
- Il networking è comparso dal nulla; è stato semplicemente esplicitato ciò che era implicito.
- E alcune informazioni compaiono in forma diversa (ma equivalente).

Attenzione perciò a cosa succede durante questo processo automatizzato, se non conosciamo o ci aspettiamo a priori il risultato finale.

Dove eseguire i nostri container con Docker-Compose

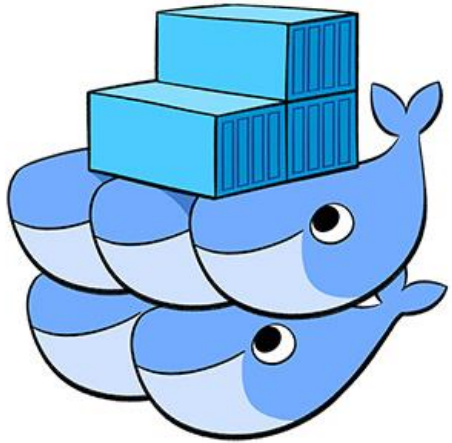
- Sulla nostra macchina
  - Ok solo in fase di sviluppo o per demo
- Su un server dedicato proprietario
  - Ok, ma sarà necessario fare il setup di tutto l'ambiente circostante
  - Non solo di docker, ma anche della pubblicazione sul web (ad esempio comprare un dominio, etc.)
- Su servizi cloud
  - Amazon, Microsoft e Google sono tra i maggiori offerenti di servizi di hosting sul mercato, con tutta l'affidabilità e i costi del marchio
  - Essendo un servizio, si paga l'abbonamento in proporzione all'uso di risorse (disco, ram, processore, dati, prestazioni, gpu…)
- Altre piattaforme…

Domande?

Avviamo dei container multipli con Docker Compose

mostra demo

Docker Swarm e Kubernetes



Abbiamo gestito più container su una sola macchina, ma se ne volessimo di più?

Una sola macchina che esegue le nostre applicazioni in un singolo host è un importante single point of failure, e potrebbe non essere in grado di gestire il carico di sistema quando il numero dei container e le relative risorse sono elevati.

Esistono delle strategie che permettono di aggirare questo problema; tra queste citiamo Docker Swarm e Kubernetes.

Noi ci concentreremo su Kubernetes

Strumenti utilizzati

Di recente, sono passati a containerd

- Docker: i container eseguiti usano questa piattaforma
- Minikube: gestisce un cluster locale di Kubernetes, ideale per lo sviluppo, ma non per un deployment in produzione
- Kubectl: CLI per comunicare con un cluster di Kubernetes
- Kubeadm: Gestisce un cluster da produzione
- Yaml: i vari elementi vengono descritti con la precedentemente vista formattazione, seppur con differenze (che vedremo)

Containers su hosts multipli

L’idea alla base di Docker Swarm e Kubernetes è di avere i container delle nostre applicazioni e servizi distribuiti su un numero di hosts maggiore di 1 (o anche 1, ma è fatica sprecata)

Si prevede inoltre la possibilità di eseguire lo stesso identico container in più repliche, per aumentare l’affidabilità dei container; un blackout potrebbe rendere non disponibili i container eseguiti in un certo host, ma se questi container erano multipli e non tutti in esecuzione sull’host non più disponibile allora il nostro servizio o applicazione rimane utilizzabile.

Si prevede inoltre di distribuire il carico delle risorse sugli host, a seconda delle risorse disponibili su ognuno di questi hosts.

Nomenclature ed elementi

In docker, avevamo a disposizione i container (services), i volumi (volumes) e le reti (networks)

In Kubernetes abbiamo a disposizione sia elementi simili che nuovi.

Un **node** è un host

Un **pod** equivale grossomodo ad un container di docker, ma possono esistere più containers dentro un singolo pod, per esempio un container secondario potrebbe essere necessario per un altro container in fase di setup

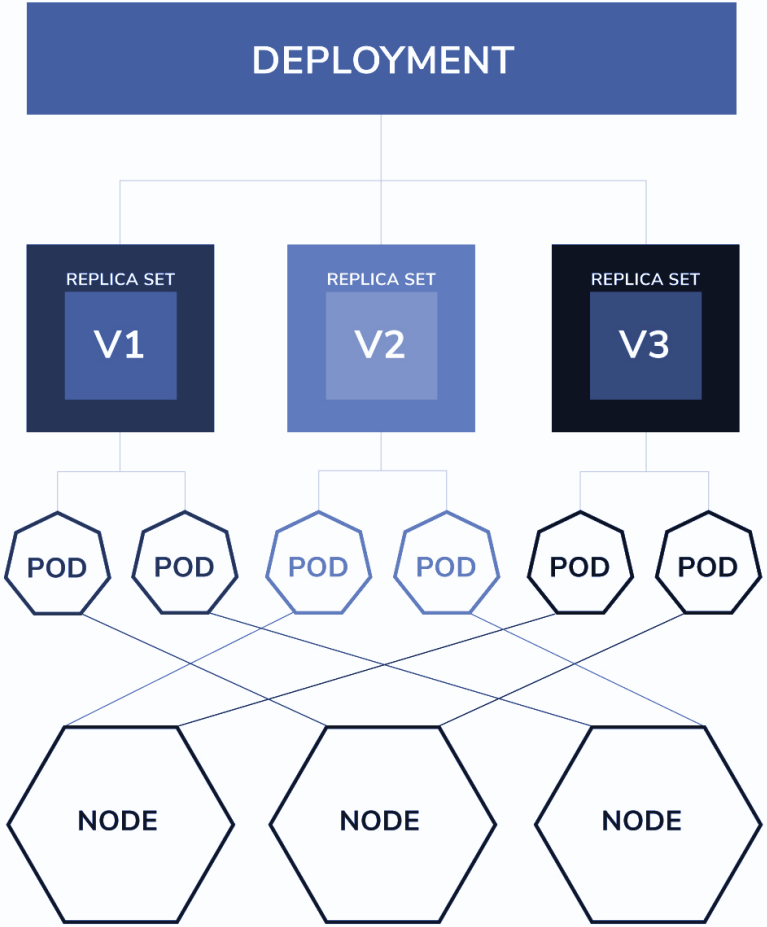
Un **replicaSet** gestisce un insieme di pod che devono essere uguali e sincronizzati

Un **deployment** gestisce un replicaSet

Un **statefulSet** è come un deployment, ma con ulteriori garanzie su stabilità e raggiungibilità dei container

Un **service** espone un deployment (non confondiamolo con un service di docker!)

Un **volume** è la stessa cosa in docker, ma dobbiamo aggiungere altro al riguardo



Deployments, replicaSets, Pods, Nodes

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  # continua a destra, indentatura a partire del cancelletto
template:
  metadata:
    labels:
      app: nginx
  spec:
    containers:
    - name: nginx
      image: nginx:1.14.2
      ports:
      - containerPort: 80
```

Taints e Tolerations

È corretto lasciare eseguire un pod che gestisce un database su un host con una scheda video molti potente? Probabilmente è uno spreco.

È corretto lasciare eseguire un pod che si occupa di addestrare una rete neurale per il riconoscimento di immagini su un RaspBerry? No, non avremmo le risorse necessarie.

Disaccoppiare il voler eseguire un container dal dove volerlo eseguire rende concrete queste difficoltà, che vengono risolte con **taints** (macchie o contaminazioni, io dirò “puzzo” o “puzza”) e **tolerations** (tollerazioni)

Taints e Tolerations

Un **Taint** si applica ad un **Node** e repelle i **Pods**

Una **Toleration** si applica ad un **Pod** e permette a quest'ultimo di lasciarsi attrarre dai **Nodes**

In entrambi i casi i requisiti imposti possono essere meri suggerimenti o obblighi, a seconda dei casi desiderati. I requisiti possono essere multipli in entrambi i casi; per quelli obbligatori, ogni requisito deve essere soddisfatto

Si possono inoltre definire i comportamenti da seguire quando un pod è eseguito su un nodo con taint/toleration incompatibili, a seguito di un cambiamento in itinere di almeno uno dei due componenti

Volumi

In Kubernetes esistono molti tipi di volumi diversi; quelli principali sono comunque simili a quelli presenti in docker. Ricordiamo che i volumi effimeri vengono persi in caso di spengimento/crash del pod che ne fa uso, mentre quelli persistenti possono sopravvivere a tali eventi.

Esiste un elemento intermedio per i volumi persistenti (**PersistentVolumes** o **PV**); l **PersistentVolumeClaim** (**PVC**); questi, in breve, allacciano i container agli utenti che li richiedono.

Esiste anche la **StorageClass**, che definisce appunto la classe a cui un certo volume appartiene, definendone alcune proprietà.

I volumi possono inoltre essere gestiti/creati dinamicamente, previa configurazione da parte dell'amministratore del cluster

Volumi statici e dinamici

Volumi statici: in caso di volumi statici, è l'amministratore del cluster che si deve occupare della creazione dei volumi e delle loro caratteristiche.

Volumi dinamici: nell'altro caso, quando un PVC chiede un PV che non si rispecchia in nessun volume statico già esistente, il cluster può provvedere alla creazione di un volume sul momento. Questo comportamento deve essere esplicitamente permesso e configurato dall'amministratore del cluster, ma una volta impostato il sistema è automatico.

Dove stanno effettivamente i volumi?

I volumi, tranne quelli effimeri, saranno salvati da qualche parte accessibile dagli host, come per la loro definizione. Se abbiamo multipli hosts, non è però garantito a priori che per ogni host, i dati letti e salvati siano sempre gli stessi; se io creo un file di testo su POD_1 in esecuzione su NODE_1, più tardi, quando POD_1 è in esecuzione su NODE_2 (per qualsivoglia motivo), non troverò quel file.

È quindi importante che i vari nodi che ospitano i pod siano in grado di fornire gli stessi dati. Per i cluster in cloud, la gestione dell'accesso ai dati di solito è trasparente. Negli altri casi, la maggior parte delle soluzioni prevedono l'utilizzo di un NFS (Network File System).

Services

Un container, in Kubernetes, esiste senza testa; non essendo garantite a priori le condizioni della sua esistenza (per design), non posso fare ipotesi sul dove si trovi e sul come possa comunicarci.

Un **Service** colma questa lacuna e permette agli altri container (e all'esterno, se vengono esposti i servizi e le porte) di comunicare tra loro, ad esempio risolvendo i nomi piuttosto che tirare ad indovinare l'indirizzo ip assegnato ad un certo container.

Domande?

Vediamo un caso pratico

mostra demo

Riferimenti

https://docs.docker.com

https://guide.bash.academy/

https://yaml.org/

https://xkcd.com/

https://www.snap4city.org

https://kubernetes.io/