Luca Cabibbo

Architettura dei Sistemi Software

Docker

dispensa asw870

ottobre 2025

Docker is not just another tool,
it's a game-changer.
And I firmly believe that every
Programmer should learn Docker.

Javin Paul

- Riferimenti

- Luca Cabibbo. **Architettura del Software**: Strutture e Qualità. Edizioni Efesto, 2021.
  - Capitolo 39, Container e virtualizzazione basata su container
- Docker
  https://www.docker.com/
  https://docs.docker.com/
- Nickoloff, J., Kuenzli, S. **Docker in Action**, Manning, second edition, 2019.
- Stoneman, E. **Learn Docker in a Month of Lunches**, Manning, 2020.

- Obiettivi e argomenti

- Obiettivi
  - presentare la piattaforma per container Docker
- Argomenti
  - introduzione
  - Docker
  - Docker in pratica
  - come funziona Docker
  - discussione

* Introduzione

- Questa dispensa presenta *Docker* – un container engine molto popolare
  - parte del materiale alla base di questa dispensa è presente sul libro nel paragrafo 39.6
  - inoltre, questa dispensa esemplifica e discute anche l'utilizzo pratico di Docker
  - l'uso di container Docker per l'esecuzione di applicazioni Spring (una tematica centrale nelle esercitazioni di questo corso) è invece l'argomento di una successiva dispensa

* Docker

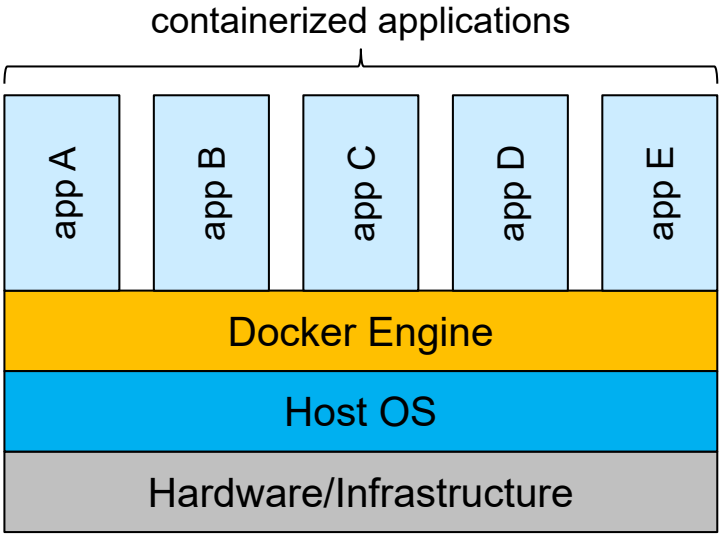
- *Docker* (www.docker.com) è una piattaforma per container (un container engine) per costruire, rilasciare ed eseguire applicazioni distribuite – in modo semplice, veloce, scalabile e portabile
  - un *container Docker* è un'unità software standardizzata, che impacchetta un servizio software, insieme alle sue configurazioni e dipendenze
    - un container contiene ogni cosa necessaria per eseguire quel servizio software – codice eseguibile, configurazioni, librerie e strumenti di sistema
    - un'immagine di container Docker diventa un'istanza di container a runtime quando viene eseguita nel Docker Engine
  - i container Docker sono leggeri (usano poche risorse e si avviano velocemente), standardizzati e aperti (e quindi portabili: si possono eseguire con le principali distribuzioni Linux e con Windows e Mac OS, e anche nel cloud) e sicuri

Storia

- La piattaforma Docker (2013) è stata inizialmente costruita sopra ai container LXC (2008)
  - LXC offre un insieme di funzionalità del kernel per la gestione di container – che però sono di basso livello e spesso difficili da usare direttamente
  - Docker si è basato su queste fondamenta per fornire un insieme di strumenti di alto livello e funzionalità più potenti e più semplici da usare
  - oggi Docker si basa sulle librerie *containerd* e *runc* (2014, 2015) – oltre che su *cgroup* e *namespace*
  - Docker è stato un successo immediato ed è utilizzato in produzione da molte aziende – poche tecnologie hanno visto un tasso di adozione simile

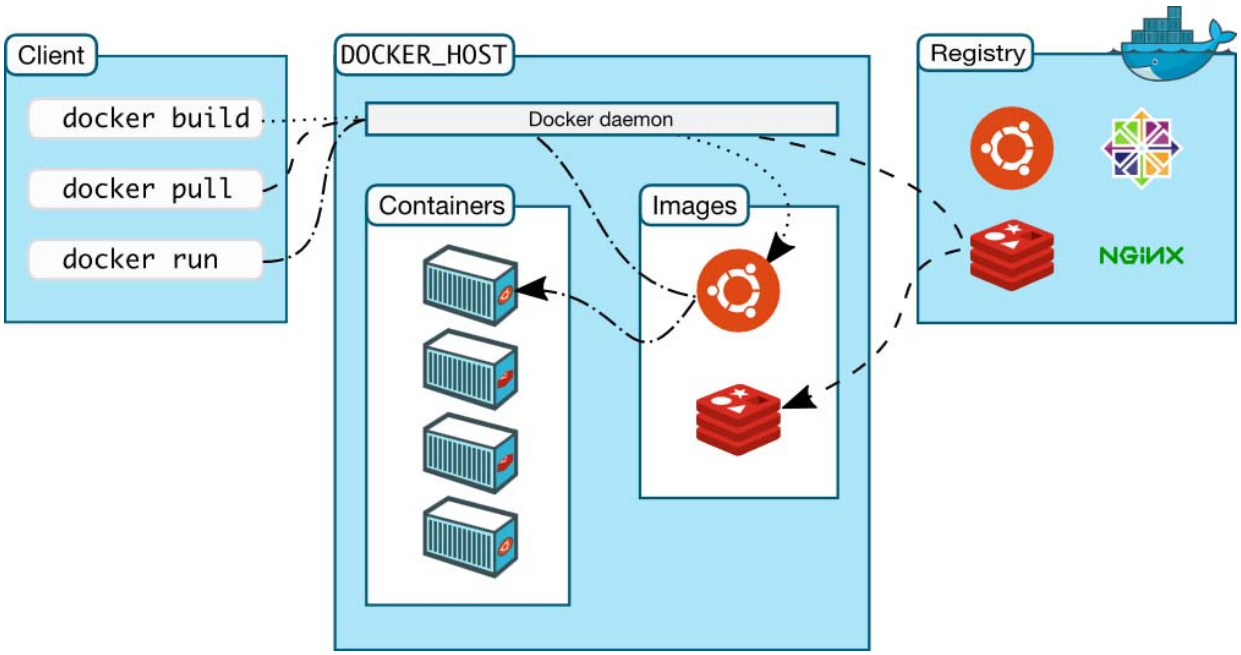
Docker

- La piattaforma *Docker* consente una separazione tra le applicazioni e l'infrastruttura di esecuzione
  - per semplificare il rilascio delle applicazioni
  - per garantire la portabilità dei servizi implementati mediante container – sia on premises che nel cloud



Docker Engine

- Il nucleo fondamentale della piattaforma Docker è *Docker Engine*



Docker Engine

- Docker Engine è basato su un'architettura client-server
  - il *server* è un host in grado di eseguire e gestire container Docker
    - basato sul processo persistente (demone) che fornisce il runtime per container Docker (dockerd) – in pratica, è un demone di alto livello che agisce da intermediario nei confronti del container manager sottostante *containerd*
    - gestisce un insieme di oggetti Docker – container, immagini, reti e volumi
  - il *client* (docker) accetta comandi dall'utente mediante un'interfaccia CLI e comunica con il demone Docker sull'host
    - la comunicazione avviene mediante un'API REST
  - il *registry* contiene un insieme di immagini
    - il registry pubblico di Docker è Docker Hub

Container e immagini

- Due tipi fondamentali di oggetti Docker
  - un *container* è, appunto, un'istanza di container, che contiene un'applicazione o un servizio – insieme a tutto ciò che serve per eseguirlo
    - è un concetto dinamico, runtime
    - può essere eseguito in un host
  - un'*immagine* è un modello per la creazione di container
    - è un concetto statico
    - non può essere eseguita direttamente
  - relazione tra container e immagini, in Docker
    - ogni container è (sempre) creato da un'immagine
    - da un'immagine è possibile creare molti container

Immagini

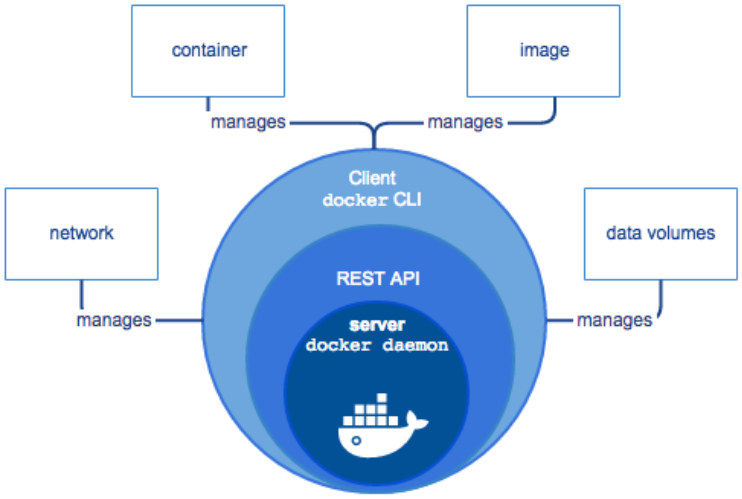
- In pratica, un'*immagine* è un insieme di file – che rappresentano lo snapshot del file system di un container
  - ad es., un'immagine contenente un OS Ubuntu, Open JDK e una specifica applicazione Java di interesse
  - un'altra immagine potrebbe essere specifica per NGINX oppure per Apache Kafka
  - un'immagine è un concetto statico, inerte
    - non viene eseguita direttamente
    - non ha un proprio stato
    - è immutabile

Container

- Un *container* è un'istanza eseguibile di container, creata da un'immagine Docker
  - un "application container" – che contiene un'applicazione o un servizio
  - ad es., un sistema software distribuito potrebbe comprendere
    - N container che sono tutte repliche di un'applicazione web di interesse (basati su una stessa immagine)
    - un ulteriore container per distribuire le richieste dei client tra le N repliche dell'applicazione web di interesse (basato su un'immagine per NGINX)
  - un container è un concetto dinamico, runtime
    - può essere eseguito su un host
    - ha un proprio stato – che può cambiare durante l'esecuzione
      - ad es., il contenuto del file system (nel disco) o lo stato delle sessioni (in memoria centrale)

Il server Docker

- Per riassumere, il server Docker
  - esegue il processo demone, runtime per container Docker
  - gestisce un insieme di oggetti Docker – soprattutto container e immagini, ma anche reti e volumi
  - consente l'accesso ai suoi client, locali e remoti, mediante CLI e REST



Registry di immagini

- Un *registry* è un servizio (pubblico o privato) che contiene una collezione di immagini di container
  - *Docker Hub* (https://hub.docker.com) è il registry pubblico di Docker – ma sono possibili anche registry privati
  - un *repository* è una porzione di un registry che contiene un insieme di immagini di container – di solito sono varianti o versioni diverse di una stessa immagine
- Un registry pubblico contiene in genere delle *immagini di base* – che contengono solo un OS, ma in alcuni casi anche del software di base – ma non software applicativo
  - esempi di immagini di base sono ubuntu, postgres, hashicorp/consul, bitnami/kafka, openjdk ed eclipse-temurin

Funzionalità e utilizzo

- Ecco le principali funzionalità offerte dalla piattaforma Docker
  - creare un container (un'istanza di container) a partire da un'immagine di container
  - avviare, monitorare, ispezionare, arrestare e distruggere container
  - creare e gestire immagini di container
  - gestire gruppi correlati di container – in cui eseguire applicazioni distribuite multi-container

* Docker in pratica

- L'interazione con un host Docker avviene mediante un'interfaccia (CLI o remota, l'interfaccia remota è basata su un'API REST)
  - questa API è basata sul comando `docker` – con numerose opzioni/comandi/operazioni per la gestione di immagini e container (e di altri oggetti Docker) e del loro ciclo di vita
    - i comandi `docker image` per la gestione delle immagini
    - i comandi `docker container` per la gestione dei container
  - alcuni comandi di uso comune esistono in due versioni, una estesa e una breve
    - ad es., `docker container run` e `docker run`
    - ad es., `docker image ls` e `docker images`

Docker in pratica

- Alcuni comandi Docker di base
  - docker image build (oppure docker build) consente di costruire un'immagine (personalizzata)
    - docker build -t *image-name context*
  - docker container create (oppure docker create) consente di creare un nuovo container da un'immagine
    - docker create --name=*container-name image-name*
  - docker container start (oppure docker start) consente di mandare in esecuzione un container (già creato)
    - docker start *container-name*
  - docker container run (oppure docker run) crea e manda in esecuzione un nuovo container (anche anonimo), mediante un comando singolo
    - docker run [--name=*container-name*] *image-name*

Creazione ed esecuzione di container

- Un primo esempio minimale – basato sull'immagine hello-world disponibile presso il Docker Hub
  - docker run hello-world

```
Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub. (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/engine/userguide/
```

Creazione ed esecuzione di un container

- Un altro esempio, basato su un'altra immagine
  - docker run aswroma3/whalesay Hello, world!

```
 _____________
< Hello, world! >
 -------------
    \
     \
      \
                    ##         .
              ## ## ##        ==
           ## ## ## ## ##    ===
       /""""""""""""""""\___/ ===
      {                       /  ===-
       \_____ o           __/
         \    \         __/
          \____\_______/
```

  - in questo caso il comportamento è parametrico

Costruzione di immagini

- Per la costruzione di un'immagine personalizzata, Docker utilizza un approccio di tipo *infrastructure-as-code* – sulla base di un file di testo speciale di nome **Dockerfile**
  - il Dockerfile contiene tutti i comandi da eseguire per costruire un'immagine personalizzata
  - il comando docker build -t *image-name context* consente di costruire automaticamente un'immagine (di nome *image-name*) a partire da un contesto *context*
    - il contesto può essere una cartella locale – in particolare, . – oppure una locazione in un repository Git
    - il contesto deve contenere il Dockerfile, insieme a ogni altro file di interesse (ad es., file binari, script e template)

Dockerfile – FROM e ENTRYPOINT

- Un **Dockerfile** è composto da una sequenza di istruzioni

```
# Hello world
FROM busybox:latest
ENTRYPOINT ["echo", "Hello, world!"]
```

  - l'istruzione **FROM** specifica l'immagine di base da cui costruire l'immagine personalizzata (ed eventualmente la sua versione)
    - ad es., busybox (è una distribuzione Linux minimale) oppure ubuntu:24.04

Dockerfile – FROM e ENTRYPOINT

- Un **Dockerfile** è composto da una sequenza di istruzioni

```
# Hello world
FROM busybox:latest
ENTRYPOINT ["echo", "Hello, world!"]
```

  - **ENTRYPOINT ["*executable*", "*param1*", "*param2*", ...]** è un'istruzione che specifica l'eseguibile o il comando che deve essere eseguito dai container che verranno creati da questa immagine
    - questa istruzione non deve essere eseguita durante la creazione dell'immagine – piuttosto deve far parte dei metadati dell'immagine, per poter essere eseguita nel container
  - un Dockerfile deve iniziare con un'istruzione FROM e, di solito, termina con una singola istruzione ENTRYPOINT

Creazione dell'immagine e del container

- Costruzione di un'immagine
  - docker build -t myhello . – dalla cartella che contiene il Dockerfile visto in precedenza
    - crea una nuova immagine di nome myhello
- Creazione di un container
  - docker create --name=myhello myhello
    - crea un nuovo container di nome myhello a partire dall'immagine myhello
- Esecuzione di un container
  - docker start -i myhello
    - avvia il container myhello (in modo interattivo)
    - in questo caso, visualizza Hello, world! e poi termina

```
Hello, world!
```


Dockerfile – CMD

- L'istruzione **CMD** consente di specificare degli argomenti per l'istruzione ENTRYPOINT – questi argomenti possono essere sovrascritti all'avvio del container, rendendo parametrico il comportamento del container

```
# Hello world
FROM bosybox:latest
ENTRYPOINT ["echo"]
CMD ["Hello, world!"]
```

  - docker build -t myhello2 .
  - docker run myhello2

```
Hello, world!
```

  - docker run myhello2 Ciao, mondo!

```
Ciao, mondo!
```

Esempio: Apache HTTP Server

- Nei Dockerfile è possibile usare anche altre istruzioni
  - ad es., il Dockerfile per un server Apache HTTP

```
# Dockerfile for Apache HTTP Server

FROM ubuntu:24.04

# Install apache2 package
RUN apt-get update && \
    apt-get install -y apache2

# Other instructions
ENV APACHE_LOG_DIR=/var/log/apache2
VOLUME /var/www/html
EXPOSE 80

# Launch apache2 server in the foreground
ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

  - ora spieghiamo le nuove istruzioni

L'istruzione RUN

- L'istruzione **RUN** specifica un comando che va eseguito durante la costruzione di un'immagine
  - ad es., per richiedere l'esecuzione di un comando o di uno script durante il provisioning dell'immagine di container – e non durante l'esecuzione del container
  - un Dockerfile può contenere più istruzioni RUN – che vengono eseguite in sequenza
- L'importante differenza tra l'istruzione ENTRYPOINT e le istruzioni RUN è il momento della loro esecuzione
  - le istruzioni specificate da RUN vengono eseguite durante la costruzione di un'immagine – ma non dai relativi container
  - l'istruzione specificata da ENTRYPOINT verrà eseguita dai container creati a partire dall'immagine – ma non durante la costruzione dell'immagine

L'istruzione RUN

- Di solito è preferibile avere in un Dockerfile una sola istruzione RUN (o comunque poche) – che specificano una sequenza di comandi separati da && \ – anziché tante istruzioni RUN
  - ad esempio

```
# Install apache2 package (migliore!)
RUN apt-get update && \
    apt-get install -y apache2
```

  - va preferito a

```
# Install apache2 package (peggiore!)
RUN apt-get update
RUN apt-get install -y apache2
```

  - la spiegazione per questo consiglio viene fornita più avanti

Altre istruzioni

- Altre istruzioni per il Dockerfile
  - l'istruzione **COPY** ***src dest*** copia un insieme di file o cartelle dalla sorgente ***src*** (che deve essere relativa al contesto della costruzione dell'immagine) alla destinazione ***dest*** (nel container)
  - l'istruzione **ADD** ***src dest*** è simile a **COPY** – ma consente di copiare nel container anche dei file remoti (ovvero esterni al contesto)
  - l'istruzione **ENV** ***key=value*** imposta una variabile d'ambiente nel container

L'istruzione VOLUME

- Altre istruzioni per il Dockerfile
  - l'istruzione **VOLUME *path*** definisce un punto di montaggio (mount) esterno – per montare dati nell'host o in un altro container
    - l'istruzione VOLUME va usata in congiunzione con altre opzioni dei comandi docker create e docker run
      - l'opzione -v *host-src:container-dest* monta nel container una cartella del sistema host – è una cartella condivisa tra l'host e il container in una posizione ben definita dell'host
      - l'opzione -v *container-dest* monta invece nel container un volume anonimo – è una cartella gestita da Docker, che in pratica risiede sempre nell'host, che viene ancora condivisa con il container (utile se il container la vuole condividere con altri container)
      - l'opzione --volumes-from=*container-name* monta nel container i volume gestito da un altro container

L'istruzione EXPOSE

- Altre istruzioni per il Dockerfile
  - l'istruzione **EXPOSE *port*** specifica che il container ascolta a runtime alla porta ***port***
    - questa istruzione viene di solito usata in congiunzione con altre opzioni dei comandi docker create e docker run per il port mapping, ovvero per pubblicare alcune porte di un container nel suo host (questi sono i termini usati da Docker per il port forwarding)
      - l'opzione -p *host-port:container-port* per pubblicare una porta specifica esposta dal container su una porta specifica dell'host
      - l'opzione -P per pubblicare tutte le porte esposte dal container su porte casuali dell'host
    - in ogni caso, i container possono comunicare tra di loro anche su porte non esposte oppure non pubblicate sull'host

Esempio: Apache HTTP Server

- Dockerfile per un server Apache HTTP

```
# Dockerfile for Apache HTTP Server

FROM ubuntu:24.04

# Install apache2 package
RUN apt-get update && \
    apt-get install -y apache2

# Other instructions
ENV APACHE_LOG_DIR=/var/log/apache2
VOLUME /var/www/html
EXPOSE 80

# Launch apache2 server in the foreground
ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```


Esempio: Apache HTTP Server

- Costruzione dell'immagine
  - docker build -t myapache . – dalla cartella che contiene il Dockerfile
- Creazione del container
  - docker create
    -v ./www:/var/www/html -p 8080:80
    --name=myapache myapache
    - le pagine servite dal server HTTP sono quelle nella cartella locale dell'host www
    - il server HTTP è reindirizzato alla porta 8080 dell'host
- Esecuzione del container
  - docker start myapache – avvia il container myapache
    - poi si potrà accedere al server HTTP dall'host su http://localhost:8080

Altri comandi Docker

- Altri comandi Docker utili
  - per elencare i container in esecuzione (o anche arrestati)
    - docker container ls – oppure docker ps [-a]
  - per ispezionare le porte usate da un container – utile soprattutto quando si usa l'opzione -P
    - docker container port *container-name* – oppure docker port
    - il risultato è della forma 80/tcp -> 0.0.0.0:8080
  - per ispezionare un container o un'immagine
    - docker container inspect *container-name* – oppure docker inspect
    - restituisce le informazioni sul container o l'immagine (in formato JSON) – ad es., la configurazione della rete (tra cui la pubblicazione delle porte) e la condivisione di volumi

Altri comandi Docker

- Altri comandi Docker utili
  - per visualizzare i log generati in un container
    - docker container logs *container-name* – oppure docker logs
  - per arrestare un container in esecuzione
    - docker container stop *container-name* – oppure docker stop
  - per rimuovere un container
    - docker container rm *container-name* – oppure docker rm
  - per arrestare tutti i container in esecuzione (da usare con cautela!)
    - docker stop $(docker ps -a -q)
  - per rimuovere tutti i container (da usare con cautela!)
    - docker rm $(docker ps -a -q)

Altri comandi Docker

- Altri comandi Docker utili
  - per elencare le immagini nella cache locale
    - docker image ls – oppure docker images
  - per rimuovere un'immagine dalla cache locale
    - docker image rm *image-name* – oppure docker rmi
  - per rimuovere tutte le immagini dalla cache locale (da usare con cautela!)
    - docker rmi -f $(docker images -q)

Altri comandi Docker

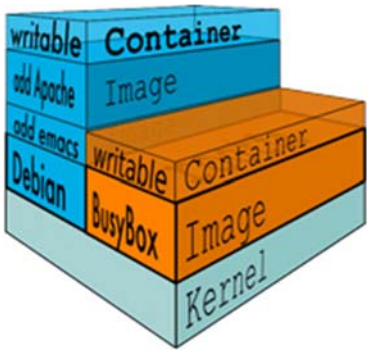
- Altri comandi Docker utili
  - il client Docker può essere utilizzato anche per specificare comandi da eseguire in un host Docker remoto *docker-host*
    - docker -H=tcp://*docker-host*:2375 *command*
    - docker -H=tcp://*docker-host*:2376 *command*
    - la porta 2376, a differenza della 2375, supporta un accesso sicuro su TLS
    - l'host Docker deve essere configurato per essere abilitato all'accesso remoto
  - in alternativa, è possibile specificare l'host Docker remoto usando la variabile d'ambiente DOCKER_HOST
    - export DOCKER_HOST=tcp://*docker-host*:2375
    - docker *command* – il comando viene eseguito su *docker-host* anziché localmente

* Come funziona Docker

- Discutiamo ancora il funzionamento di Docker – in particolare, i seguenti aspetti
  - formato delle immagini e dei container
  - costruzione di immagini
  - creazione di container
  - esecuzione di container
  - condivisione di dati (volumi)
  - reti
  - registry

- Formato delle immagini (e dei container)

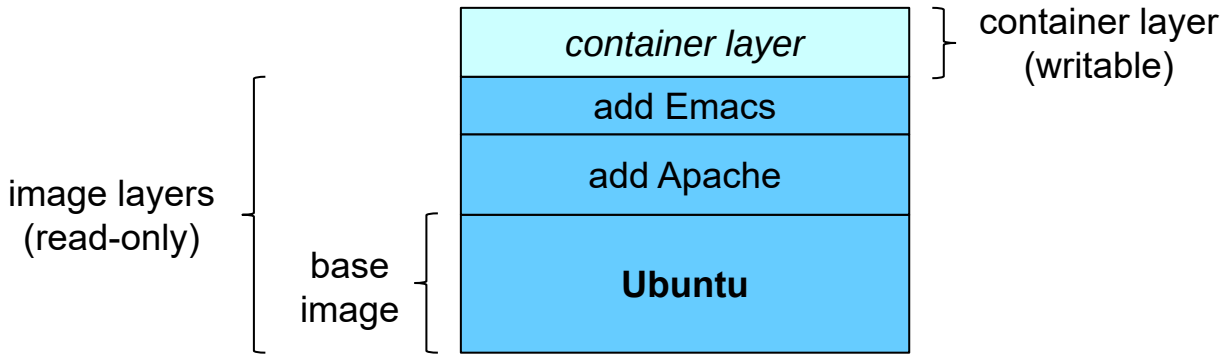
- Un elemento fondamentale di Docker è il formato usato per il file system delle immagini e dei container
  - il file system di un'immagine (o di un container) è costituito da una sequenza di strati – ciascuno strato è un insieme di file



  - questi strati sono combinati in un singolo file system coerente mediante uno *Union File System* (*UFS*)
    - un file viene letto nello strato più alto in cui si trova
    - in un container, l'unico strato che può essere scritto a runtime è lo strato più alto

Formato delle immagini (e dei container)

- Un elemento fondamentale di Docker è il formato usato per il file system delle immagini e dei container
  - il file system di un'immagine (o di un container) è costituito da una sequenza di strati – ciascuno strato è un insieme di file

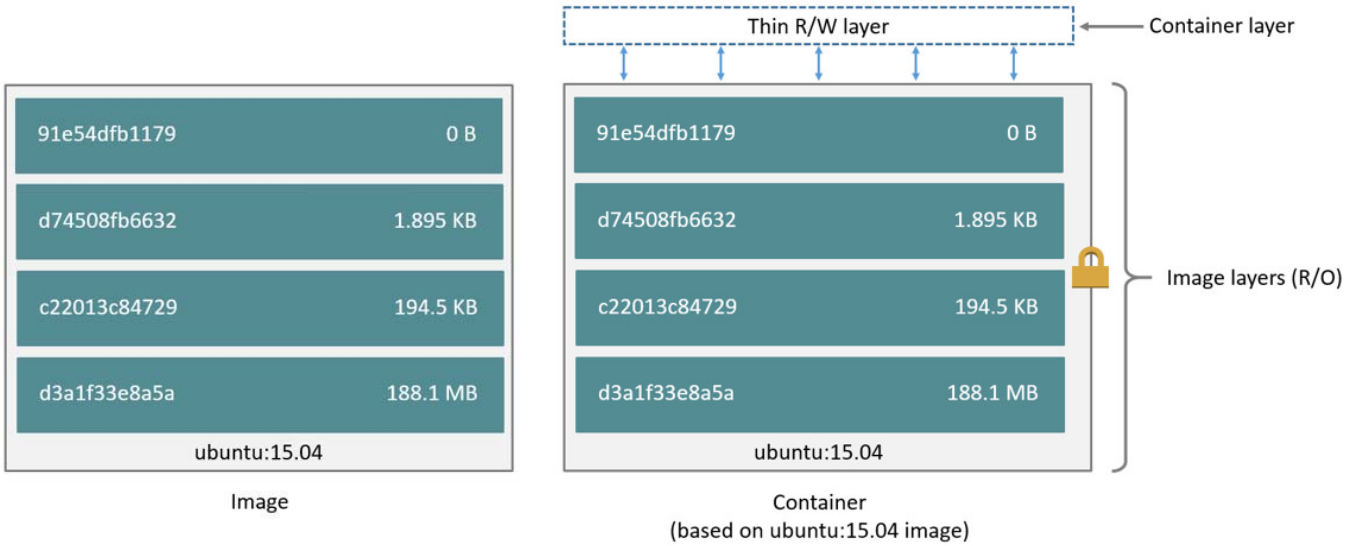


Formato delle immagini (e dei container)

- Nel file system di ogni immagine (o container), la base è sempre un'immagine di base – di solito contiene un OS e le sue librerie
  - ogni strato successivo corrisponde in genere all'installazione di un package, un middleware o di un'applicazione
  - oltre a questi strati, ciascun container (ma non le immagini) possiede un ultimo strato aggiuntivo, che rappresenta l'unica parte modificabile del file system del container
    - tutte le scritture, le modifiche e le cancellazioni eseguite nel container operano su quest'ultimo strato aggiuntivo
  - questo formato "leggero"
    - consente di condividere strati tra immagini e tra container
    - facilita l'aggiornamento delle immagini (ad es., per aggiornare un'applicazione a una nuova versione) – che può essere effettuato mediante l'aggiornamento o l'aggiunta di strati, anziché la ricostruzione completa delle immagini

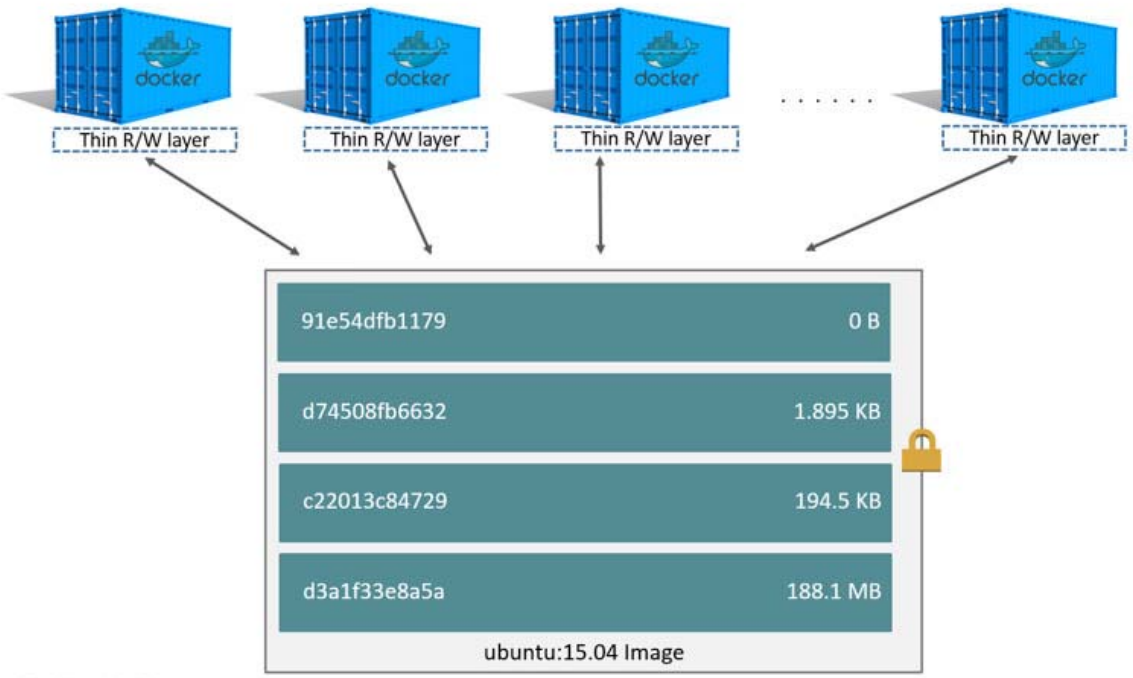
Immagini e container

- Un’immagine
- Un container (o meglio, il suo file system)



Immagini e container

- Un’immagine condivisa da più container



- Costruzione di immagini

- La costruzione di un'immagine personalizzata è basata sull'esecuzione di un Dockerfile, e avviene come segue
  - per prima cosa, l'immagine di base specificata da FROM viene scaricata dal registry in una cache di immagini dell'host (se non è già presente nella cache)
    - questa immagine di base (può essere composta da uno o più strati) viene usata come strato (o strati) di base della nuova immagine personalizzata

Costruzione di immagini

- La costruzione di un'immagine personalizzata è basata sull'esecuzione di un Dockerfile, e avviene come segue
  - poi, ripetutamente, per ciascuna istruzione X del Dockerfile (in particolare, RUN)
    - viene creato un nuovo container temporaneo C_X a partire dall'immagine corrente, a cui viene aggiunto sopra un nuovo strato scrivibile S_X
    - nel container C_X viene eseguita l'istruzione X del Dockerfile – che probabilmente modificherà lo strato S_X
    - quindi, il container C_X viene arrestato, e lo strato S_X viene "congelato" e salvato come strato (in sola lettura) dell'immagine corrente, aggiornandola
  - Docker consiglia di minimizzare il numero di strati nelle immagini e nei container (per ridurre i tempi di accesso al file system) – e dunque di minimizzare il numero di istruzioni RUN di un Dockerfile

- Creazione di container

- La creazione di un container avviene sempre a partire da un'immagine
  - un container consiste di un file system e di meta-dati
    - il file system (a strati) del container è ottenuto dall'immagine iniziale, a cui viene aggiunto sopra un nuovo strato scrivibile, specifico per il container – questo strato viene allocato nel file system dell'host
  - le immagini sono invece immutabili e possono essere condivise da più container

- Esecuzione di container

- Esecuzione di un container
  - quando viene richiesta l'esecuzione di un container, il container engine alloca le risorse runtime per il container
    - ad es., alloca (nel kernel dell'host) un insieme di namespace e configura la rete per il container
  - poi avvia il container, a partire dal suo file system
  - infine, il container esegue il comando specificato da ENTRYPOINT (con gli argomenti specificati da CMD o dalla linea di comando)

- Volumi e condivisione di dati

- Lo storage dei container è di per sé effimero – quando un container viene distrutto, tutti i suoi dati vengono persi
  - come è possibile gestire dati persistenti?
- Un *volume* è una directory al di fuori del file system di un container
  - un volume può essere acceduto, condiviso e riusato da più container
  - un volume consente di gestire dati persistenti, in modo indipendente dal ciclo di vita dei singoli container che lo possono accedere

Volumi e condivisione di dati

- Una prima possibilità è (come abbiamo già fatto) montare una cartella dell'host come volume in un container in esecuzione nell'host, mediante l'opzione -v di docker create o di docker run
  - docker create -v ./www:/var/www/html …
  - in questo caso, i dati risiedono nell'host (ovvero, in una posizione assoluta predefinita del file system dell'host) – e non nel container
    - le letture di questi dati vengono effettuate nell'host
      - un esempio di utilizzo è per servire pagine HTML residenti nell'host
    - inoltre, anche le eventuali modifiche a questi dati vengono effettuate nell'host, in modo persistente
      - un esempio di utilizzo è per redirezionare nell'host i file di log di un server eseguito in un container
  - in effetti, questi dati potrebbero anche risiedere in un volume montato nell'host

Volumi e condivisione di dati

- Un'altra possibilità consente di definire un volume condiviso tra più container – ma senza che il volume sia legato a una specifica cartella dell'host – va usata l'opzione --volumes-from
  - ad es., viene prima creato un container *container*, usando l'opzione -v per montare nel container un volume anonimo – ad es., -v /var/logs
    - questo volume risiederà in una cartella gestita da Docker associata al volume – in pratica, il volume risiede ancora nell'host, ma non in una posizione assoluta predefinita
  - è poi possibile creare altri container che accedono a quel volume condiviso, con l'opzione --volumes-from *container*
    - tutti questi container possono leggere da e scrivere su questo volume
  - se viene cancellato il container in cui risiede un volume, il volume viene comunque mantenuto (a meno che ne venga richiesta una cancellazione esplicita)

- Reti

- Docker consente di gestire la comunicazione in rete tra container, nonché con l'host
  - durante l'installazione, Docker crea automaticamente tre reti, bridge, host e none – ma è anche possibile crearne altre
  - la rete bridge (in modalità "bridge") è associata all'interfaccia virtuale docker0 sull'host e a una rete privata 172.17.0.1/16
  - quando un container viene mandato in esecuzione, Docker gli associa un indirizzo IP libero della rete bridge
    - è possibile collegare un container a una rete differente usando l'opzione --network=*network*
  - i container possono comunicare tra di loro conoscendo la posizione assoluta (indirizzo IP e porta) dei diversi servizi presenti in rete
  - la rete host aggiunge invece un container alla rete dell'host

Reti

- Altre informazioni sulle reti
  - docker inspect consente di trovare le informazioni necessarie per comunicare in rete con un container
    - ad es., il server Apache HTTP potrebbe essere esposto all'indirizzo 172.17.0.2:80 (della rete privata)
  - è anche possibile rendere questi servizi accessibili all'host e al di fuori dell'host mediante il port mapping (port forwarding) – tramite le opzioni -p e -P di docker create e docker run
    - Docker gestisce queste opzioni configurando automaticamente nell'host le regole NAT di iptables
    - l'opzione --ip consente anche di associare a un container uno specifico indirizzo IP (valido per l'host)

Reti

- Altre informazioni sulle reti
  - usando una *rete definita dall'utente* (anziché la rete bridge) i container possono comunicare tra di loro anche mediante il loro nome "logico" – oltre che mediante il loro indirizzo IP
    - il container engine opera da DNS per i suoi container
  - creazione di una rete definita dall'utente
    - docker network create -d *network-driver network-name*
    - ad es., docker network create -d bridge mynet
  - collegamento di un container a una rete
    - docker run --network=mynet --name=mycontainer -it busybox
    - gli altri container collegati a questa rete possono vedere questo container mediante il suo nome "logico" mycontainer
    - un container può anche essere collegato a più reti

- Registry

- Un registry è un servizio per la gestione di un insieme di immagini di container
  - operazioni principali di un registry
    - docker pull *image-name* – effettua il download di un'immagine dal registry alla cache locale dell'host – altrimenti, docker build lo fa automaticamente
    - docker push *image-name* – effettua l'upload di un'immagine al registry
    - interrogazione del registry
  - il registry pubblico di Docker è *Docker Hub* – alcune delle immagini che gestisce sono "ufficiali"
    - in alternativa, *Docker Registry* è uno strumento per gestire un proprio registry privato
    - nello spirito di Docker, Docker Registry può essere eseguito come un container

Uso di Docker Hub

- Utilizzo di Docker Hub – bisogna prima creare sul sito di Docker Hub un proprio account, ad es., aswroma3 – dopo di che
  - login
    - docker login [-u aswroma3] [-p *password*] [*server*]
  - creazione e taggatura (tagging) di un'immagine
    - docker build -t aswroma3/myhello . oppure
    - docker build -t myhello . seguito da docker tag myhello aswroma3/myhello
  - salvataggio di un'immagine sul registry (deve essere taggata)
    - docker push aswroma3/myhello
  - caricamento di un'immagine dal registry (opzionale)
    - docker pull aswroma3/myhello
  - creazione ed esecuzione di un container dall'immagine
    - docker run aswroma3/myhello

Docker Registry

- Gestione di un Docker Registry (privato) – nello spirito di Docker, può essere eseguito come un container
  - avvio del registry (l'opzione -d esegue il container in background) – supponiamo sul nodo myregistry
    - docker run -d -p 5000:5000 --restart=always --name registry -v /var/local/docker/registry:/var/lib/registry registry:2
  - creazione e taggatura di un'immagine
    - docker build -t myhello .
    - docker tag myhello myregistry:5000/myhello
  - salvataggio di un'immagine sul registry (deve essere taggata)
    - docker push myregistry:5000/myhello
  - caricamento di un'immagine dal registry
    - docker pull myregistry:5000/myhello
  - creazione ed esecuzione di un container dall'immagine
    - docker run myregistry:5000/myhello

- Raccomandazioni generali

- Alcune raccomandazioni sui container – e le relative immagini
  - un solo processo per container
    - sostiene il riuso di immagini e container
    - sostiene la scalabilità orizzontale
  - container "effimeri" (*ephemeral*, ovvero temporanei, passeggeri, e senza stato) – per quanto possibile
    - in modo che un container possa essere arrestato e distrutto e poi sostituito da un altro container il più rapidamente possibile
    - sostiene disponibilità e scalabilità
  - container minimali
    - usa l'immagine di base più ridotta possibile, evita l'installazione di package non necessari e minimizza il numero di strati
    - sostiene la disponibilità

* Discussione

- La piattaforma Docker si è imposta molto rapidamente come tecnologia di riferimento per i container
  - molte aziende (comprese grandi aziende come Google) usano Docker non solo per lo sviluppo e il test, ma anche come ambiente di produzione per applicazioni con requisiti critici di disponibilità, scalabilità ed elasticità
  - Docker è supportato sia on premises che nel cloud
  - grazie a Docker, i container sono divenuti una tecnologia per il rilascio di applicazioni alternativa e complementare alla virtualizzazione di sistema
  - i benefici e le motivazioni per l'uso di Docker saranno più evidenti dopo aver discusso il rilascio di un proprio sistema software distribuito con Docker, e in particolare mediante la composizione e l'orchestrazione di container Docker – che costituiscono l'argomento di un successivo capitolo e di successive dispense