Luca Cabibbo

Architettura dei Sistemi Software

Docker

dispensa asw890

marzo 2018

Containers are almost becoming synonymous to Docker.

Dinesh Subhraveti

- Fonti

- Docker

  https://www.docker.com/

  https://docs.docker.com/

- Obiettivi e argomenti

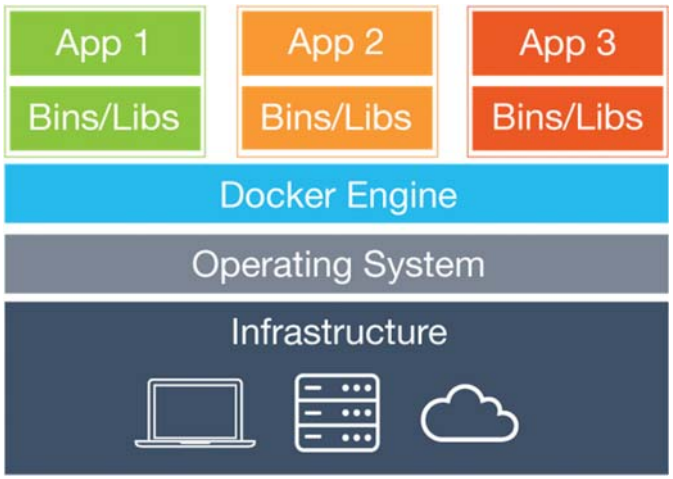
- Obiettivi
  - presentare Docker
- Argomenti
  - Docker
  - Docker in pratica
  - come funziona Docker
  - un'applicazione contenitorizzata
  - discussione

* Docker

- *Docker* (www.docker.com) è una piattaforma per contenitori per costruire, rilasciare ed eseguire applicazioni distribuite – in modo semplice, veloce, scalabile e portabile
  - un *contenitore Docker* è un'unità software eseguibile standardizzata che incapsula un servizio software
    - un contenitore contiene ogni cosa necessaria per eseguire quel servizio – codice eseguibile, configurazioni, librerie e strumenti di sistema
  - i contenitori Docker sono leggeri (usano poche risorse e si avviano velocemente), standardizzati e aperti (si possono eseguire sulle principali distribuzioni Linux e su Windows) e sicuri

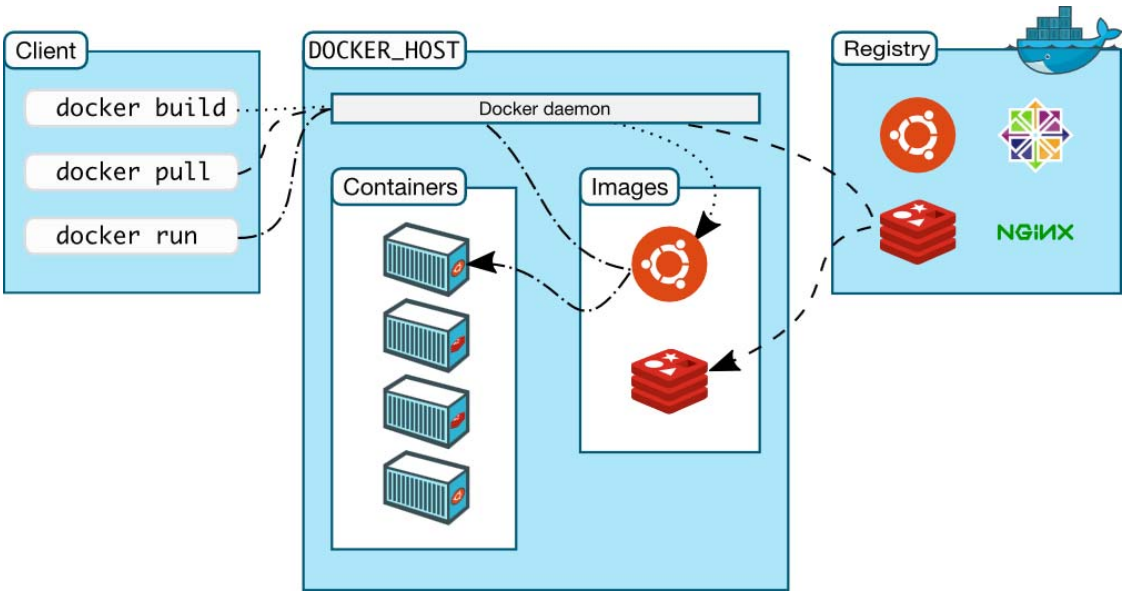
Docker

- La piattaforma *Docker* consente una separazione tra le applicazioni e l'infrastruttura di esecuzione, in modo da semplificare il rilascio delle applicazioni
  - garantisce che il servizio implementato da un contenitore possa essere eseguito sempre nello stesso modo, indipendentemente dal suo ambiente di esecuzione – sia on premise che sul cloud



Architettura di Docker Engine

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



Architettura di Docker Engine

- La piattaforma Docker è basata su un'architettura client-server
  - il *server* è un host in grado di eseguire e gestire contenitori Docker
    - esegue il processo demone Docker (dockerd)
    - inoltre, gestisce un insieme di oggetti Docker – contenitori, immagini (compresa una cache locale di immagini), reti e volumi
  - il *client* (docker) accetta comandi dall'utente (mediante un'interfaccia dalla linea di comando) e comunica con il demone Docker sull'host
    - il client e il demone Docker comunicano mediante un'API REST
  - il *registry* (pubblico o privato) contiene un insieme di immagini
    - il registry pubblico di Docker è Docker Hub

Contenitori e immagini

- Due concetti fondamentali di Docker sono il contenitore e l'immagine – si tratta di concetti correlati ma distinti
  - un *contenitore* è una specie di macchina virtuale che contiene un'applicazione o un servizio – *è l'analogo di un'istanza di VM*
    - insieme all'OS, alle librerie, al middleware e a tutti i pezzi di software necessari per eseguire l'applicazione o il servizio
    - un contenitore è un concetto dinamico, runtime – un contenitore può essere eseguito su un host
  - un'*immagine* è un modello completo per la creazione di uno o più contenitori – *è l'analogo di un'immagine di VM*
    - un'immagine è un concetto statico – le immagini non vengono eseguite direttamente
  - relazione tra contenitori e immagini
    - ogni contenitore è creato da un'immagine
    - da un'immagine è possibile creare molti contenitori

Immagini

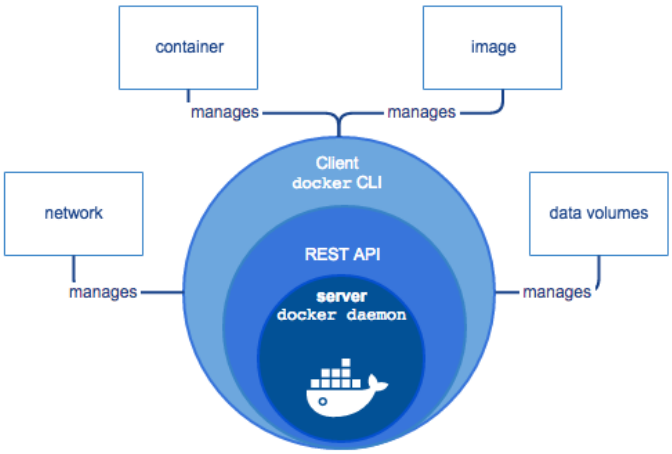
- In Docker, un'*immagine* è un modello completo per la creazione di uno o più contenitori
  - un'immagine è sostanzialmente un file (o un insieme di file) che è lo snapshot di un contenitore
    - ad es., un'immagine potrebbe contenere un OS Ubuntu con un application server WildFly
    - ad es., un'altra immagine (diversa dalla precedente) potrebbe contenere un OS Ubuntu con Wildfly, insieme a un'applicazione web di interesse
  - un'immagine è un concetto statico, inerte
    - un'immagine non viene eseguita direttamente
    - un'immagine non ha un proprio stato
    - un'immagine è immutabile

Contenitori

- In Docker, un *contenitore* è un'istanza eseguibile di un'immagine Docker – in pratica, è una specie di macchina virtuale che contiene un'applicazione o un servizio
  - ad es., un sistema software distribuito potrebbe comprendere
    - N contenitori che sono N repliche di una stessa applicazione web di interesse (basati su una stessa immagine)
    - un ulteriore contenitore per distribuire le richieste dei client tra le N repliche dell'applicazione web di interesse (basato su un'altra immagine)
  - un contenitore è un concetto dinamico, runtime
    - un contenitore può essere eseguito su un host
  - ogni contenitore 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 Docker
  - gestisce un insieme di oggetti Docker – soprattutto contenitori e immagini
  - consente l'accesso ai suoi client, locali e remoti, mediante CLI e REST



Registry di immagini

- Inoltre, in Docker, un *registry* è un servizio (pubblico o privato) che contiene una collezione di immagini di contenitori
  - *Docker Hub* (https://hub.docker.com) è il registry pubblico di Docker – ma un'organizzazione potrebbe gestire un proprio registry privato
  - inoltre, un *repository* è una porzione di registry che contiene un insieme di immagini di contenitori – che sono di solito varianti o versioni diverse di una stessa immagine
- Un registry pubblico contiene di solito delle *immagini di base* – ovvero, immagini che contengono solo un OS e/o del software di base, ma non del software applicativo
  - ad es., un'immagine di base è un'immagine con un application server WildFly pre-installato
  - invece, un'immagine con WildFly insieme ad una specifica applicazione di interesse è un'immagine "personalizzata"

* 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) – che offre un insieme di comandi/operazioni per la gestione di immagini e contenitori e del loro ciclo di vita
  - docker build consente di costruire un'immagine (personalizzata)
    - docker build -t *image-name context*
  - docker create consente di creare un nuovo contenitore da un'immagine
    - docker create --name=*container-name image-name*
  - docker start consente di mandare in esecuzione un contenitore
    - docker start *container-name*
  - docker run crea e manda in esecuzione un nuovo contenitore (eventualmente anonimo) con un singolo comando
    - docker run [--name=*container-name*] *image-name*

Creazione ed esecuzione di un contenitore

- 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 contenitore

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

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


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 su 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 (e la sua versione)
    - ad esempio, ubuntu oppure busybox (BusyBox è una distribuzione Linux minimale)
  - **ENTRYPOINT ["*executable*", "*param1*", "*param2*", …]** è un'istruzione che specifica l'eseguibile o il comando che deve essere eseguito dai contenitori che verranno creati da questa immagine
  - un Dockerfile deve iniziare con un'istruzione FROM e, di solito, termina con una singola istruzione ENTRYPOINT

Creazione dell'immagine e del contenitore

- Costruzione di un'immagine
  - docker build -t hello-img . – dalla cartella che contiene il Dockerfile visto in precedenza
    - crea una nuova immagine di nome hello-img
- Creazione di un contenitore
  - docker create --name=hello hello-img
    - crea un nuovo contenitore di nome hello a partire dall'immagine hello-img
- Esecuzione di un contenitore
  - docker start -i hello
    - avvia il contenitore hello (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 – con l'osservazione che questi argomenti possono essere sovrascritti all'avvio del contenitore

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

  - docker build -t hello-cmd-img .
  - docker run hello-cmd-img

```
Hello, world!
```

  - docker run hello-cmd-img Ciao, mondo!

```
Ciao, mondo!
```


Esempio: Apache HTTP Server

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

```
# Dockerfile for Apache HTTP Server

FROM ubuntu:14.04

# 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 contenitore
  - un Dockerfile può contenere più istruzioni RUN – che vengono eseguite in sequenza
- La differenza principale 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
  - l'istruzione specificata da ENTRYPOINT verrà eseguita dai contenitori creati a partire dall'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
RUN apt-get update && \
    apt-get install -y apache2
```

  - va preferito a

```
# Install apache2 package
RUN apt-get update
RUN apt-get install -y apache2
```

  - la spiegazione per questo suggerimento 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 contenitore)
  - l'istruzione **ADD *src dest*** è simile – ma consente di copiare nel contenitore anche dei file remoti (ovvero esterni al contesto)
  - l'istruzione **ENV *key value*** imposta una variabile d'ambiente nel contenitore

L'istruzione VOLUME

- Altre istruzioni per il Dockerfile
  - l'istruzione **VOLUME *path*** definisce un punto di montaggio (mount) esterno – per dati nel sistema host o in un altro contenitore
    - l'istruzione VOLUME va usata in congiunzione con altre opzioni dei comandi docker create e docker run
      - l'opzione -v *host-src:container-dest* per montare nel contenitore una cartella del sistema host – è una cartella condivisa tra l'host e il contenitore
      - l'opzione -volumes-from=*container-name* per montare nel contenitore un volume gestito da un altro contenitore

L'istruzione EXPOSE

- Altre istruzioni per il Dockerfile
  - l'istruzione **EXPOSE *port*** specifica che il contenitore ascolta a runtime alla porta ***port***
    - questa istruzione va di solito usata in congiunzione con altre opzioni dei comandi docker create e docker run per pubblicare (questo è il termine usato da Docker per il port forwarding) alcune porte di un contenitore nel suo host
      - l'opzione -p *host-port:container-port* per pubblicare una porta specifica esposta dal contenitore su una porta specifica dell'host
      - l'opzione -P per pubblicare tutte le porte esposte dal contenitore su porte casuali dell'host
    - va notato che i contenitori possono comunicare comunque 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:16.04

# 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 apache-img . – dalla cartella che contiene il Dockerfile
- Creazione del contenitore
  - docker create -v ~/docker/www:/var/www/html -p 8080:80 --name=apache apache-img
    - le pagine servite sono quelle nella cartella ~/docker/www dell'host
    - il server HTTP è reindirizzato alla porta 8080 dell'host
- Esecuzione del contenitore
  - docker start apache
    - manda in esecuzione il contenitore apache
    - poi potrò accedere al server HTTP dall'host su http://localhost:8080

Altri comandi Docker

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

Altri comandi Docker

- Altri comandi Docker utili
  - per arrestare un contenitore in esecuzione
    - docker stop *container-name*
  - per rimuovere un contenitore
    - docker rm *container-name*
  - per rimuovere un'immagine
    - docker rmi *image-name*
  - per arrestare tutti i contenitori (attenzione!)
    - docker stop $(docker ps -a -q)
  - per rimuovere tutti i contenitori (attenzione!)
    - docker rm $(docker ps -a -q)
  - per rimuovere tutte le immagini (attenzione!)
    - docker rmi -f $(docker images -q)

Altri comandi Docker

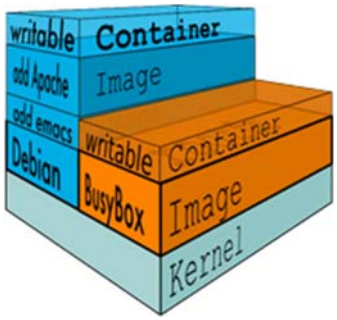
- Altri comandi Docker utili
  - il client Docker può essere utilizzando anche per specificare comandi da eseguire su un host Docker remoto *docker-host*
    - docker -H=tcp://*docker-host*:2375 *command*
    - docker -H=tcp://*docker-host*:2376 *command*
    - la porta 2376, a differenza della 2375, supporta un accesso sicuro su TLS
  - 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*

* Come funziona Docker

- Discutiamo ancora il funzionamento di Docker – che si basa su
  - costruzione di immagini
  - creazione di contenitori da immagini
  - esecuzione di contenitori
- Discutiamo anche aspetti relativi alla condivisione di dati (volumi), alle reti, al registry

- Formato delle immagini

- Un elemento fondamentale di Docker è il formato delle immagini
  - un'immagine è costituita da una sequenza di strati – in cui ciascuno strato è un insieme di file



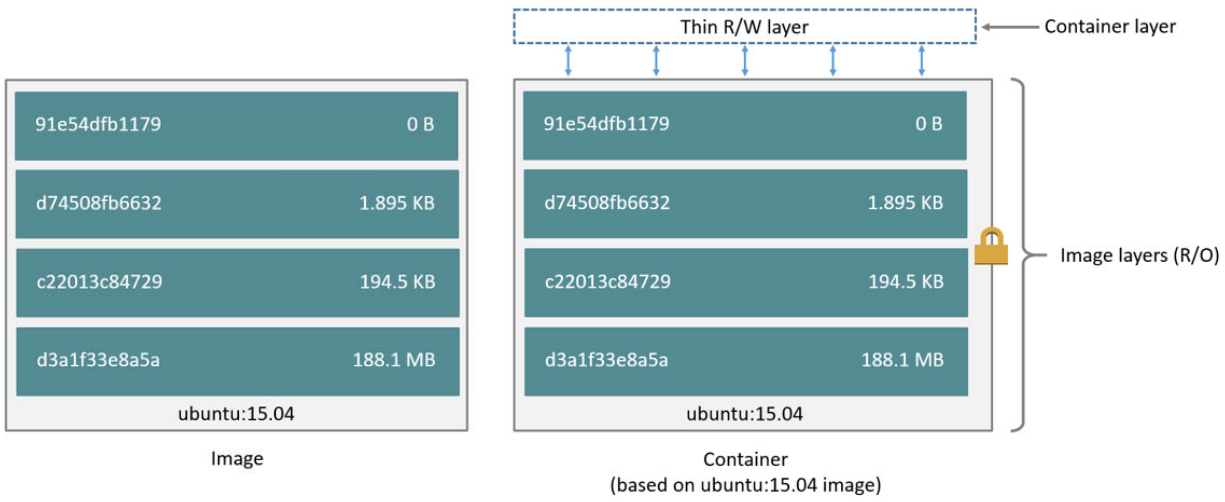
  - questi strati sono combinati mediante AUFS (che è uno *Union File System*) in un singolo file system coerente
    - un file viene letto nello strato più alto in cui si trova
    - un file viene scritto solo nello strato più alto – che è l'unico strato che può essere scritto a runtime

Formato delle immagini

- La base di ogni immagine è sempre un'immagine di base – che di solito contiene un OS e le sue librerie di sistema
  - ogni strato successivo corrisponde di solito all'installazione di un package, un middleware o di un'applicazione
  - oltre a questi strati, ciascun contenitore possiede un ultimo strato aggiuntivo, che rappresenta la parte modificabile del file system del contenitore
    - tutte le scritture, le modifiche e le cancellazioni eseguite nel contenitore operano su quest'ultimo strato aggiuntivo
  - questo formato "leggero"
    - consente di condividere strati tra immagini e contenitori
    - 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

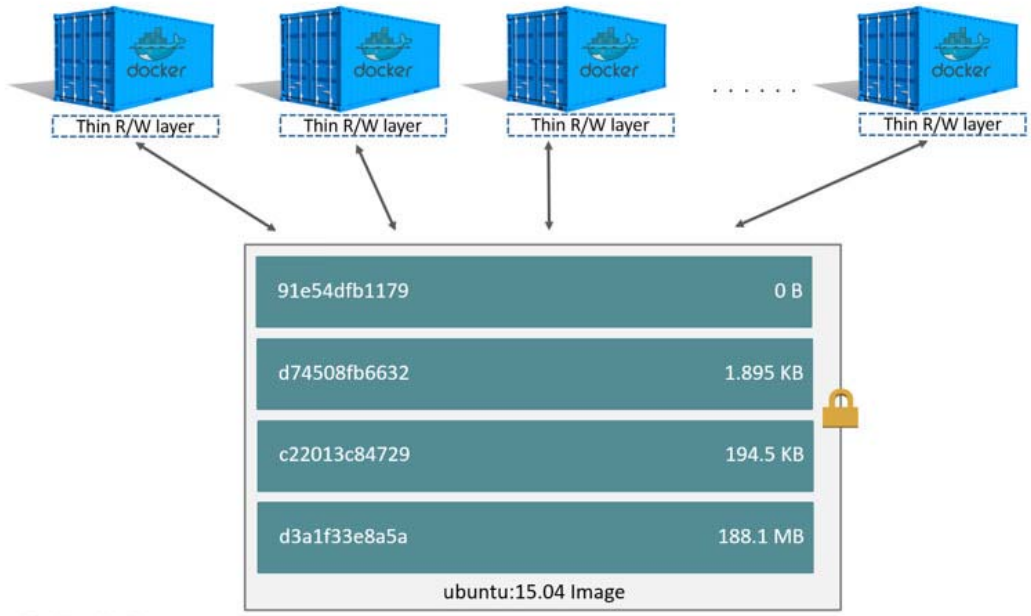
Immagini e contenitori

- Un'immagine
- Un contenitore (o meglio, il suo file system)



Immagini e contenitori

- Un'immagine condivisa da più contenitori



- Costruzione di immagini

- La costruzione di un'immagine personalizzata è basata sull'esecuzione di un Dockerfile, che avviene così
  - 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 viene usata come strato di base della nuova immagine personalizzata
  - poi, ripetutamente, ciascuna istruzione del Dockerfile (in particolare, RUN) viene eseguita in un nuovo strato sopra all'immagine corrente
    - il risultato dell'esecuzione di un'istruzione del Dockerfile viene poi salvato (prima di eseguire la successiva istruzione)
    - Docker consiglia di minimizzare il numero di strati nelle immagini e nei contenitori – e dunque di minimizzare il numero di istruzioni RUN di un Dockerfile

- Creazione ed esecuzione di contenitori

- La creazione di un contenitore avviene a partire da un'immagine
  - un contenitore consiste di un file system e di meta-dati
    - il file system (a strati) è ottenuto dall'immagine iniziale, più un nuovo strato scrivibile, specifico per il contenitore – questo strato viene allocato nel file system dell'host
    - le immagini sono immutabili e possono essere condivise da più contenitori
- Esecuzione di un contenitore
  - il container engine alloca le risorse runtime per il contenitore – ad es., alloca un gruppo di processi e un insieme di namespace e configura la sua rete
  - poi avvia il contenitore, a partire dal suo file system
  - infine, il contenitore esegue il comando specificato da ENTRYPOINT (con gli argomenti specificati da CMD o dalla linea di comando)

- Volumi e condivisione di dati

- Un *data volume* è una directory al di fuori dell'UFS dei contenitori
  - un volume può essere acceduto, condiviso e riusato da più contenitori
  - un volume consente di gestire dati persistenti, indipendenti dal ciclo di vita dei singoli contenitori che lo possono accedere
  - infatti, lo storage dei contenitori è di per sé effimero
    - quando un contenitore viene distrutto, tutti i suoi dati vengono persi

Volumi e condivisione di dati

- Una prima possibilità è montare una cartella dell'host come volume in un contenitore in esecuzione nell'host, mediante l'opzione -v di docker create o di docker run
  - docker create -v ~/docker/www:/var/www/html …
  - in questo caso, i dati risiedono nell'host – e non nel contenitore
    - pertanto, le 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 contenitore

Volumi e condivisione di dati

- Un'altra possibilità è avere volumi condivisi tra contenitori – ma non legati alla posizione di una specifica cartella dell'host – in questo caso va usata anche l'opzione -volumes-from
  - va prima creato un contenitore, usando l'opzione -v per indicare una cartella condivisa del contenitore – senza però legare questo volume a nessuna cartella dell'host
    - il volume risiederà in questo contenitore
    - in pratica, questo volume risiede nell'host, ma non in una posizione assoluta predefinita
  - poi è possibile creare altri contenitori che accedono a quel volume condiviso, con l'opzione -volumes-from *container-name*
  - se viene cancellato il contenitore su cui risiede un volume, il volume viene comunque mantenuto (a meno che ne venga richiesta una cancellazione esplicita)

- Contenitori e rete

- Docker consente di gestire la comunicazione in rete tra contenitori, 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 contenitore viene mandato in esecuzione, Docker gli associa un indirizzo IP libero della rete bridge
    - è possibile collegare un contenitore ad una rete diversa usando l'opzione --network=*network*
  - i contenitori 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 contenitore alla rete dell'host

Contenitori e rete

- Altre informazioni sulle reti
  - docker inspect consente di trovare le informazioni necessarie per comunicare in rete con un contenitore
    - 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 (port forwarding o port mapping) – 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 contenitore uno specifico indirizzo IP (valido per l'host)

Contenitori e rete

- Altre informazioni sulle reti
  - usando una *rete definita dall'utente* (anziché la rete bridge) i contenitori possono comunicare tra di loro anche mediante il loro nome – oltre che mediante il loro indirizzo IP
    - il container engine opera da DNS per i suoi contenitori
  - creazione di una rete definita dall'utente
    - docker network create -d *network-driver network-name*
    - ad es., docker network create -d bridge my-net
  - collegamento di un contenitore ad una rete
    - docker run --network=my-net --name=container1 -it busybox
    - gli altri contenitori collegati a questa rete possono vedere questo contenitore mediante il suo nome container1
    - un contenitore può anche essere collegato a più reti

- Registry

- Un registry è un servizio per la gestione di un insieme di immagini di contenitori
  - 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 contenitore

Docker Registry

- Gestione di un Docker Registry in un contenitore
  - avvio del registry (l'opzione -d esegue il contenitore in background) – supponiamo sul nodo myregistry
    - docker run -d -p 5000:5000 --restart=always --name registry registry:2
  - creazione e "taggatura" di un'immagine
    - docker build -t hello .
    - docker tag hello myregistry:5000/hello
  - salvataggio di un'immagine sul registry (deve essere "taggata")
    - docker push myregistry:5000/hello
  - caricamento di un'immagine dal registry
    - docker pull myregistry:5000/hello
  - creazione ed esecuzione di un contenitore dall'immagine
    - docker run myregistry:5000/hello

- Raccomandazioni generali

- Alcune raccomandazioni sui contenitori – e le relative immagini
  - un solo processo per contenitore
    - sostiene il riuso di immagini e contenitori
    - sostiene la scalabilità orizzontale
  - contenitori "effimeri" (*ephemeral*, ovvero temporanei, passeggeri, e senza stato) – per quanto possibile
    - in modo che un contenitore possa essere arrestato e distrutto e poi sostituito da un altro contenitore il più rapidamente possibile
    - sostiene disponibilità e scalabilità
  - contenitori 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

- Docker e sistemi di provisioning

- Docker può essere integrato con alcuni sistemi di provisioning – ecco alcuni esempi
  - Vagrant fornisce un provisioner per Docker, che può automaticamente installare Docker in una macchina virtuale, creare dei contenitori e anche avviare dei contenitori all'avvio della macchina virtuale
  - inoltre Vagrant può usare Docker come "provider" – ovvero, per definire degli ambienti costituiti direttamente da contenitori
  - è possibile usare Puppet, nel provisioning di una VM, per specificare l'installazione di Docker nella VM, nonché la costruzione di immagini Docker e la creazione di contenitori Docker nella VM
  - è inoltre possibile usare Puppet nella costruzione di immagini Docker, richiedendo l'esecuzione di `puppet apply` in un'istruzione `RUN` di un `Dockerfile`

* Un'applicazione contenitorizzata

- Prima di concludere, ecco un esempio relativo all'esecuzione di una semplice applicazione web in un contenitore Docker
  - l'applicazione lucky-word – si veda la dispensa su Spring Boot
  - ecco il `Dockerfile`, che utilizza un OS preconfigurato con Oracle JDK

```
# Dockerfile for the lucky-word application

FROM frolvlad/alpine-oraclejdk8

# Install the application binary
ADD build/libs/lucky-word-0.0.1-SNAPSHOT.jar lucky-word.jar

EXPOSE 8080

# Launch the Java application
ENTRYPOINT ["/usr/bin/java", "-Xmx128m", "-Xms128m"]

CMD ["-jar", "-Dspring.profiles.active=english", "lucky-word.jar"]
```

Un'applicazione contenitorizzata

- Ecco come eseguire questa applicazione
  - prima di tutto, bisogna effettuare la build dell'applicazione lucky-word (nell'ambiente di sviluppo)

```
gradle build
```

  - dopo di che, bisogna costruire un'immagine di contenitore per l'applicazione e poi avviare il contenitore (nell'ambiente Docker)

```
# crea l'immagine del contenitore
docker build --rm -t lucky-word-img .

# il profilo di default è quello inglese
docker run -p 8080:8080 lucky-word-img

# oppure, per eseguire l'applicazione con il profilo italiano
# docker run -p 8080:8080 lucky-word-img
#        -jar -Dspring.profiles.active=italian lucky-word.jar
```


* Discussione

- La piattaforma Docker si è imposta molto rapidamente come tecnologia di riferimento per i contenitori
  - molte aziende usano Docker 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 sul cloud
  - grazie a Docker, i contenitori sono divenuti una tecnologia per il rilascio di applicazioni alternativa e complementare alla virtualizzazione
  - le motivazioni per l'uso di Docker saranno più evidenti dopo aver discusso la composizione e l'orchestrazione di contenitori Docker – che è l'argomento di una successiva dispensa
  - il consiglio di Sam Newman (autore di Building Microservices) è
    - "I strongly suggest you give Docker a look"