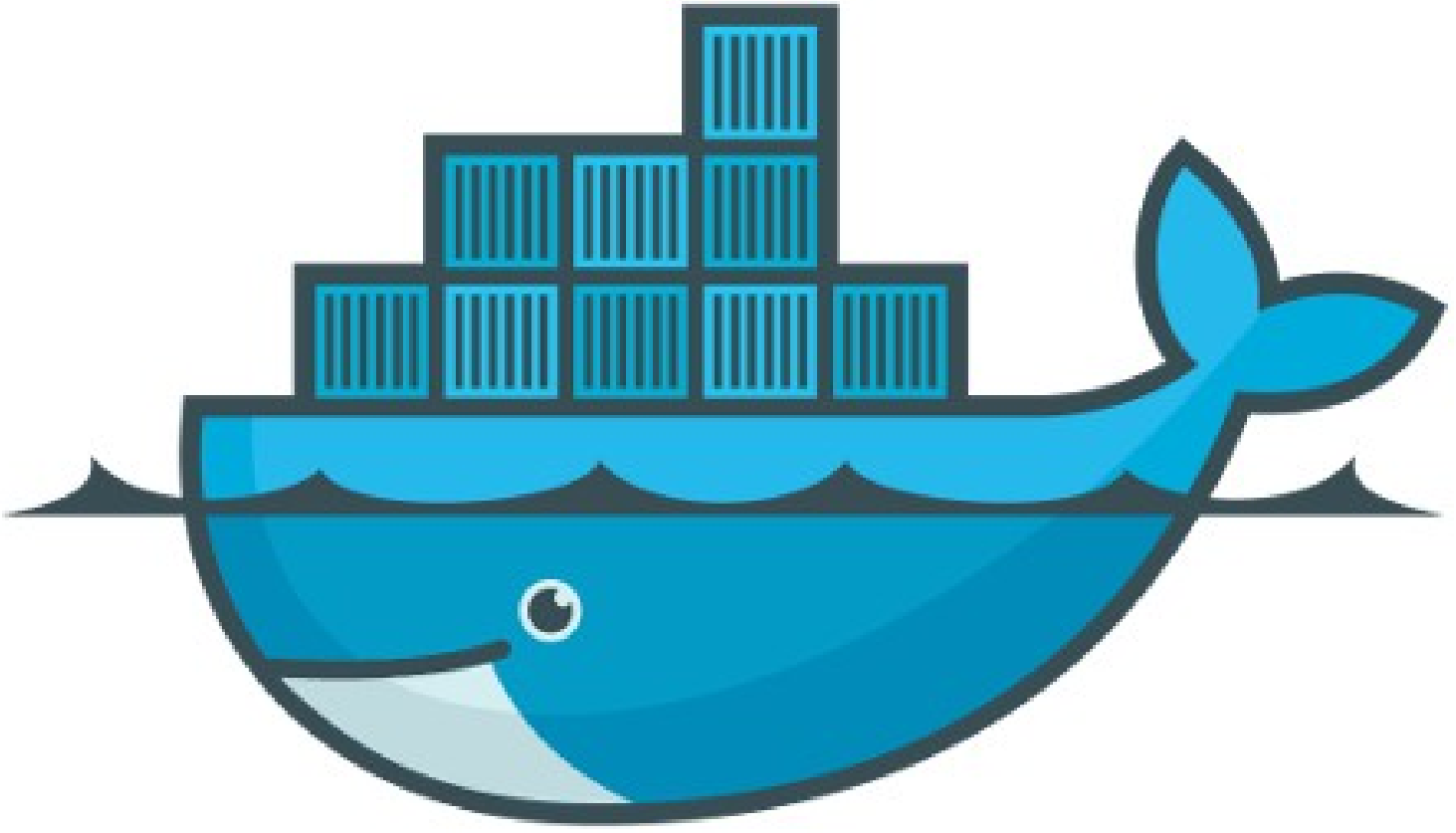
DOCKER



https://docs.docker.com/

Cos’è Docker

- **Docker** è un progetto opensource nato con lo scopo di automatizzare e semplificare la distribuzione di applicazioni.

- A tal scopo si è introdotto il concetto di contenitore “**Container**”:un oggetto leggero, portabile e autosufficiente che può essere eseguito su **cloud** (pubblici o privati) o in **locale**.

- I **Container**, sono quindi l’insieme dei dati di cui necessita un’applicazione per essere eseguita: librerie, altri eseguibili, rami del file system, file di configurazione, script, ecc.

Docker – Processo di distribuzione

- Il **processo di distribuzione** di un'applicazione si riduce quindi alla semplice creazione di una **immagine Docker**, ovvero di un file contenente tutti i dati di cui necessita l'applicazione da distribuire.

- L'immagine viene utilizzata da **Docker** per creare un **Container**, cioè **un'istanza dell'immagine che eseguirà l'applicazione in essa contenuta**.

Docker – Processo di distribuzione

- Ogni **container** è identificato da un **`id` alfanumerico unico** e generato al momento dell'esecuzione e/o da un **`nome`** assegnato automaticamente (o manualmente tramite una opzione) dal sistema.
- Ogni operazione da effettuare sul **container** deve far riferimento al suo **`id`** o **`nome`**.
- Ad ogni "riavvio del **container**" il suo **`id`** e il suo **`nome`** (se assegnato automaticamente) cambiano.
- Quindi fissare un **`nome`** manualmente permette di riferirsi sempre senza ambiguità al **container** voluto.

Docker – Proprietà dei Container

- **autosufficienti** perché contengono già tutte le dipendenze dell'applicazione e non richiedono particolari configurazioni sull'host.

- **portabili** perché sono distribuiti in un formato standard (le **immagini** appunto), che può essere letto ed eseguito da qualunque server **Docker**.

- **leggeri** perché sfruttano i servizi offerti dal kernel del sistema operativo ospitante, invece di richiedere l'esecuzione di un kernel virtualizzato come avviene nel caso delle VM. Ciò permette di ospitare un gran numero di container nella stessa macchina fisica.

Docker – Come funziona

- Il modello di esecuzione basato su **container** si fonda su una serie di *funzionalità sviluppate nel kernel Linux* (e successivamente adottate anche da altre piattaforme), espressamente studiate per questo scopo.

- Come alternativa alla **virtualizzazione**, deve garantire prestazioni superiori pur offrendo le stesse caratteristiche in termini di flessibilità nella gestione delle risorse e di sicurezza.

- L'incremento prestazionale è dovuto essenzialmente all'eliminazione di uno strato: a differenza di quanto avviene quando un processo viene eseguito all'interno di una **macchina virtuale**, **i processi eseguiti da un container sono di fatto eseguiti dal sistema ospitante, usufruendo dei servizi offerti dal kernel che esso esegue.** Viene quindi eliminato l'overhead dovuto all'esecuzione di ogni singolo kernel di ogni VM.

Docker – Come funziona

- In un sistema virtualizzato i requisiti di **flessibilità** e **sicurezza** sono a carico dell'**Hypervisor**, lo strato software in esecuzione nella macchina ospitante che si occupa di gestire le risorse allocate a ciascuna VM e che adotta (anche con l'ausilio dell'hardware) tutte le politiche necessarie per isolare i processi in esecuzione su VM differenti.

- In un ambiente basato su **container**, dove quindi non è presente un **Hypervisor**, queste funzionalità sono assolte dal **kernel del sistema operativo** ospitante.

- Linux dispone di due caratteristiche progettate proprio per questo scopo: **Control Groups** (o **cgroups**) e **Namespaces**

Docker – Control Groups

- I **control groups** sono lo strumento utilizzato dal kernel Linux per gestire l'utilizzo delle risorse di calcolo da parte di un gruppo specifico di processi.

- Grazie ai **cgroups** è possibile limitare la quantità di risorse utilizzate da uno o più processi.

- Ad esempio, è possibile limitare il quantitativo massimo di memoria RAM che un gruppo di processi può utilizzare.

Docker – Control Groups

- In un sistema Linux esistono più **cgroups**, ciascuno associato ad uno o più **resource controllers** (talvolta chiamati anche **subsystems**), in base alle risorse che gestiscono.

- Ad esempio, un **cgroup** può essere associato al **resource controller memory** per gestire la quantità di memoria allocabile da un certo insieme di processi.

- I **cgroup** sono organizzati in modo gerarchico in una **struttura ad albero**. Ogni **nodo** dell'albero rappresenta un **gruppo**, definito dalle regole che gestiscono la risorsa a cui è associato (ad esempio la dimensione massima di memoria allocabile) e dalla lista dei processi che ne fanno parte. Ciascun processo in quella lista risponderà alle regole definite nel gruppo.

Docker – Control Groups

- Si può interagire manualmente con i **cgroup** attraverso il filesystem virtuale `/sys`.

- I **cgroup** attualmente in uso dal kernel sono accessibili come subdirectory di `/sys/fs/cgroup`.

- Per creare un nuovo **cgroup** è sufficiente creare una subdirectory in quel ramo del filesystem.

Docker – Control Groups - Esempio

- Si supponga di voler limitare la memoria massima allocabile da un processo, il cui `PID` è 1001, a 1024 MB. Per far ciò è possibile creare un nuovo **cgroup** sotto memory, impostare il limite, ed aggiungere il processo al **cgroup**, con i comandi:

```
# mkdir /sys/fs/cgroup/memory/miocgroup
# echo 1048576000 >
/sys/fs/cgroup/memory/miocgroup/memory.limit_in_bytes
# echo 1001 >
/sys/fs/cgroup/memory/miocgroup/cgroup.procs
```

Docker – Control Groups

- **Docker** sfrutta questa caratteristica del kernel Linux per implementare i limiti delle risorse allocate ai **container**.

- Quando un **container** è configurato con un limite su una o più risorse, **Docker** crea i corrispondenti **cgroups** in /`sys/fs/cgroup/` ed aggiunge automaticamente i `PID` dei processi in esecuzione nel container.

Docker – Namespaces

- Il kernel ospitante ha anche il compito di garantire **l'isolamento dei processi** in esecuzione in **container** differenti.

- Non deve essere possibile per un processo in esecuzione in un **container** accedere direttamente alla macchina ospite o ad altri **container**.

- Questa funzionalità è implementata nel kernel Linux mediante l'uso dei **namespaces**.

Docker – Namespaces

- Un **namespace** è essenzialmente un “contenitore” che astrae le risorse offerte dal kernel.

- Quando un processo fa parte di un certo **namespace** esso potrà accedere soltanto alle risorse presenti nel **namespace** stesso.

- Esistono diversi **namespaces di default**, ciascuno associato ad una tipologia differente di risorse: **cgroup**, IPC, Network, Mount, PID, User, UTS.

Docker – Namespaces

- Per ciascun **container** in esecuzione **Docker** crea un opportuno gruppo di **namespaces** ed associa i processi in esecuzione nel **container** a quel gruppo di **namespace**.
- I processi in esecuzione nel **container** non accederanno ai **namespaces** della macchina ospitante, né a quelli di altri **container**.
- Questo permette effettivamente di isolare i processi in esecuzione in un **container**.
- Se non si creassero dei **namespace** ad hoc per ogni **container**, i processi al suo interno potrebbero accedere direttamente alle risorse dell'host. Assegnare dei namespace differenti ad ogni **container** significa quindi creare delle **sandbox** in grado di isolare i processi del **container** dal resto della macchina.

Docker – Namespaces

- **Cgroups** e **Namespaces** sono caratteristiche fondamentali per l'implementazione di un sistema basato su **container**.
- Sono feature mature essendo presenti in Linux da più di un decennio e basandosi su concetti già concepiti durante lo sviluppo di **OpenVZ** (nell'ormai lontano 2005).
- Funzionalità analoghe sono state successivamente integrate anche in ambiente Windows, come parte dei **Windows Native Containers**. Tuttavia, non è possibile eseguire **container** Linux direttamente sul kernel Windows (dato che, per l'appunto, non si virtualizza il sistema operativo ospite). In questi casi (host Windows ed immagini Linux), **Docker** ricorre comunque ad una macchina virtuale Linux, in esecuzione su **HyperV**.

Docker – Le immagini

- Tutto quello che è necessario per il funzionamento di un **container** è definito nella sua **immagine**.
  - si può pensare all'immagine come ad una classe di un qualunque linguaggio OOP e al container come all'oggetto da essa istanziato.
- Quando chiediamo a **Docker** di eseguire un **container** a partire dalla sua **immagine**, quest'ultima deve essere presente sul disco locale. In caso contrario, **Docker** ci avvertirà del problema:

```
# docker run hello-world
Unable to find image 'hello-world:latest' locally
```

  e provvederà a scaricarla in maniera autonoma da un **repository**.

```
(latest: Pulling from library/hello-world).
```

- Per sapere quali immagini sono state scaricate da **Docker** sul nostro computer, utilizziamo il comando:

```
# docker images
```

Docker – Repository e tag

- Con il termine **repository** intendiamo un **contenitore di immagini correlate**:
  - il **repository** `hello-world` conterrà varie versioni dell'immagine `hello-world`.
  - Ogni versione dell'immagine è associata ad una stringa che la identifica detta **`tag`**. L'ultima versione, quella che viene installata di default con il comando `docker pull`, ha sempre **`tag latest`**!
  - Se un **repository** contiene più di un **`tag`** per un immagine, è possibile riferirsi ad una data immagine con la coppia `repository:tag`.
- Il **`tag latest` è il `tag` di default**: quando facciamo riferimento ad un repository senza specificare il nome del **`tag`**, **Docker** farà implicitamente riferimento al **`tag latest`**, e se questo non esiste verrà mostrato un errore.
- La forma `repository:tag` è preferibile in quanto permette una maggiore prevedibilità riguardo il contenuto dell'**immagine**, evitando possibili conflitti tra **container** ed eventuali errori dovuti alla mancanza del **`tag latest`**.

Docker – Union FS

- Una peculiarità delle **immagini Docker** è la loro stratificazione in **layer**, ognuno dei quali contribuisce alla definizione del file system del **container**.
- I **layer** sono immutabili: sono infatti accessibili in sola lettura e non sono modificabili direttamente.
  - Questo è un grande vantaggio: infatti più immagini posso condividere un layer comune.
    Ad esempio, le immagini `redis:3.0.0` e `nginx:1.7` si basano entrambe su un layer comune (`debian:jessie`), sebbene la prima immagine definisca ad un key/value store e la seconda ad un web server.
- Digitando il comando `docker pull nginx:1.7` vedrete tutti i **layer** che vengono "uniti per creare" l'immagine che state scaricando.

Docker – I container

- Un’**immagine** è quindi una serie di **layer immutabili**: questa caratteristica permette a **immagini** diverse di condividere un layer **comune**.
- Una pila di layer accessibili in sola lettura (detti **layer immagine**),non permette modifiche, quindi ciò che avviene nel momento in cui **Docker** istanzia un **container** è la creazione, in cima a tutti gli altri **layer**, di un singolo **layer scrivibile** detto **layer container**.
  D’ora in avanti, tutte le modifiche apportate al **container** verranno memorizzate nel **layer container** **che verrà distrutto spegnendo il container.**
- **Riassumendo: un’immagine è una pila di layer accessibili in sola lettura; il container relativo è la stessa pila di layer con sopra un layer scrivibile. Il contenuto finale del container è ottenuto per sovrapposizione di tutti questi layer.**

Docker – I container sono oggetti isolati

- **A livello di filesystem**, ogni **container** è completamente isolato dal sistema sottostante e da quello di tutti gli altri **container**: ad esempio un file creato dentro un **container** persisterà fino allo spegnimento dello stesso.
- Nel caso in cui sia necessario salvare dei dati generati nel **container** è necessario ricorrere all'utilizzo di un **data volume**.
- **data volume**: permette di creare un ponte tra il filesystem del **container** e quello del sistema operativo sottostante.

Docker – I container sono oggetti isolati

- Ad esempio supponiamo di voler salvare dei file creati all'interno di un **container** con immagine Ubuntu:

```
# docker run -it -v /home/faromano/data_volume:/docs
ubuntu /bin/bash

root@b4afd75514bd:/# cd docs && touch pippo.txt
```

- Tramite l'opzione `-v` creiamo un **container** con all'interno una nuova directory, `docs`, che può essere considerata un **mount point** per la directory `data_volume` presente nella home dell'utente faromano.
- Alla chiusura del **container** in `data_volume` troveremo tutti gli elementi creati e/o modificati in `/docs`.

Docker – I container sono oggetti isolati

- A **livello di processi** il discorso è analogo a quello del filesystem: ogni **container** è a conoscenza solo dei suoi processi di sistema e non sa assolutamente nulla riguardo ai processi degli altri **container** o quelli dell'OS sottostante.
  - Collegandoci in modalità **interattiva** ad un **container** (ad esempio con il comando `docker run -it ubuntu /bin/bash`) possiamo vedere che gli unici processi attivi sono relativi al **container** stesso (usiamo il comando `ps ax`).
  - Per evitare di dover entrare in un **container** per vedere i processi in esso attivi, abbiamo a disposizione il comando `docker top`, seguito dall'id del **container**.

Docker – I container sono oggetti isolati

- Anche a **livello di rete** i **container** sono isolati (o quasi):
  - Supponiamo di dover utilizzare un container con un webserver. Ad esempio Apache:

```
# docker run httpd:latest
```

  - Tramite questo comando costruiamo ed eseguiamo un contenitore **Docker** con un web server funzionante. Ma come ci colleghiamo ad esso? Che indirizzo ip ha?
    - Sotto Linux, l'indirizzo ip corrispondente (dall'esterno) è lo stesso della macchina che ospita **Docker**.
    - Sotto Windows, essendo **Docker** ospitato su una vm con dentro linux è necessario utilizzare il comando `docker-machine ip` per conoscere l'indirizzo corretto.

Docker – I container sono oggetti isolati

- Ok, ora che conosciamo l'ip della macchina basta puntare il browser per collegarsi….. Invece no!
  - I **container** blindano tutto ciò che è contenuto al loro interno, quindi il nostro server http non risulta accessibile.
- Esistono due soluzioni per rendere accessibile dall'esterno un contenitore:
  1) mappare ad una determinata porta della macchina virtuale Linux, la porta su cui risponde il demone HTTP (la classica 80, se non diversamente specificato), tramite il comando:

     ```
     #  docker run -d -p 8088:80 httpd:2.6
     ```

  2) Utilizzare un web server (di solito `nginx`), installato sulla **docker machine** con funzione di **proxy**.

Docker – Dockerfile

- Nel caso di container complessi,cioè con molti parametri ed opzioni che descrivono l'immagine da cui creare il container, risulta utile servirsi di un **Dockerfile**.
- Un **Dockerfile** è un semplice file di testo che permette descrivere le personalizzazioni da apportare ai vari **template** affinché questi possano diventare delle **immagini** adatte allo scopo.
- il **Dockerfile** è un scritto tramite un **domain-specific language** (**DSL**), ovvero un insieme di istruzioni per la definizione di immagini **Docker**.

Docker – Dockerfile

- Una volta decise tutte le scelte in tema di configurazioni, il **Dockerfile** viene dato in pasto all'engine **Docker** che si occuperà di validarlo e di generare così una nuova **immagine**.
- Come ogni linguaggio che si rispetti ci sono delle clausole specifiche. Vedremo ora le più importanti.
- NB: tutti i comandi hanno una doppia sintassi: **`shell form`** o **`exec form`**. Per semplicità vedremo solo la sintassi **`shell form`**.
- Per un riferimento più completo: `https://sodocumentation.net/it/docker/topic/3161/dockerfiles`

Docker – Dockerfile FROM

- **FROM** non può mai mancare all'interno di ogni **Dockerfile**. Permette di specificare **un'immagine di base** (base image) da cui partire per derivare l'immagine personalizzata.
- La sintassi è simile a quella del comando `docker pull`:

```
FROM <nome_immagine>
FROM <nome_immagine>:<tag>
FROM <nome_immagine>@<hash>
```

- Esempio:

```
FROM ubuntu:latest
```

- NB: In un **Dockerfile** è possibile inserire dei commenti utizzando, ad inizio riga, il carattere `#`.

Docker – Dockerfile ENV

- **ENV** offre la possibilità di impostare variabili di ambiente valide per tutto il contesto di esecuzione di un **Dockerfile**.
- La sintassi è la seguente:

```
ENV <chiave>=<valore>
```

- Esempio:

```
ENV WWW_HOME='/var/www/'
```

- Per recuperare il valore della variabile d'ambiente all'interno del **Dockerfile**, basta riferirci ad essa anteponendo il carattere `$` alla chiave stessa (per esempio: `$WWW_HOME`)

Docker – Dockerfile RUN

- **RUN** tramite questa istruzione possiamo istruire **Docker** affinchè esegua dei comandi all'interno dell'immagine che andremo a generare.
- La sintassi è la seguente:

```
RUN <comando> <parametro1> ... <parametroN>
```

- Esempio:

```
1.FROM ubuntu:latest
2.ENV DA_INSTALLARE='vim apache2'
3.RUN apt install $DA_INSTALLARE
```

Docker – Dockerfile RUN

- L'immagine così come ottenuta dal **FROM** rimane intatta.
- In fase di *building dell'immagine*, il comando **RUN** fa sì che si generi un nuovo layer che porti con sé le modifiche derivate dal comando stesso: ogni volta che l'engine incontra questa istruzione lungo il suo percorso, esso creerà un nuovo layer!
- Quindi è **necessario riunire più comandi correlati all'interno di una singola istruzione RUN**.

```
1.RUN apt update && apt install $DA_INSTALLARE
```

- Ad esempio, nel caso dell'installazione di un nuovo pacchetto, dovesi utilizza la coppia di comandi `apt update/install`, l'utilizzo di due istruzioni **RUN** diverse per `apt update` e `apt install` potrebbe far fallire l'installazione.

Docker – Dockerfile ADD/COPY

- **ADD/COPY** entrambe producono lo stesso risultato: spostare file e directory dal **build context** (il path sul nostro computer locale dove si trova il **Dockerfile**) all'**interno del filesystem dell'immagine creata**.
- La sintassi è la seguente:
  - `ADD <src> <dest>`
  - `COPY <src> <dest>`
- Esempio:

```
1.FROM ubuntu:latest
2.ENV DA_INSTALLARE='vim apache2'
3.RUN apt update && apt install $DA_INSTALLARE
4.COPY *.txt /docs
```

Docker – Dockerfile ENTRYPOINT

- **ENTRYPOINT** è una delle clausole più importanti in quanto permette di eseguire un comando all'interno del **container** non appena questo si è avviato.
- Si differenzia dall'istruzione **RUN** poiché agisce sul **container** stesso piuttosto che sull'immagine che l'ha generato (non crea un nuovo layer).
- Utilizzata quando si vuole far partire un demone subito dopo l'avvio del container.
- La sintassi è la seguente:
  - `ENTRYPOINT <comando> <parametro_1> ... <parametro_n>`
- Esempio:

```
1 FROM ubuntu:latest
2 ENV DA_INSTALLARE='vsftpd'
3 RUN apt update && apt install $DA_INSTALLARE
4 ENTRYPOINT vsftpd
```

Docker – Dockerfile CMD

- **CMD** permette di eseguire un comando all'interno di un **container**.
- A seconda della presenza o meno della clausola **ENTRYPOINT** ha un comportamento diverso. All'interno di un **Dockerfile** è ammessa una sola istruzione **CMD**. Se ne sono specificate diverse, solo l'ultima verrà presa in considerazione.
- La sintassi è la seguente:
  - `CMD <comando> <parametro_1> ... <parametro_n>`
- Se non esiste un'istruzione **ENTRYPOINT** all'interno del **Dockerfile** in esame, l'utilizzo dell'istruzione **CMD** farà in modo che all'avvio del container venga eseguito il comando indicato.
- Esempio:

```
1 FROM ubuntu:latest
2 ENV DA_ESEGUIRE='du -hs'
3 CMD $DA_ESEGUIRE
```

- Questo è esattamente lo stesso comportamento dell'istruzione **ENTRYPOINT**, ma in questo caso possiamo sovrascrivere questo comando specificandone uno diverso nel momento in cui eseguiamo il comando `docker run <image> <comando>` da terminale.

Docker – Dockerfile CMD

- Se, invece, all'interno del nostro **Dockerfile** esiste già l'istruzione **ENTRYPOINT**, l'istruzione **CMD** avrà il compito di fornire dei parametri di default al comando espresso nell'istruzione **ENTRYPOINT** e l'unica sintassi disponibile sarà:

```
#exec form
CMD ["<parametro_1>", ..., "<parametro_n>"]
```

- In questo modo il comando non sarà più sovrascrivibile dall'esterno (a meno che non si utilizzi il flag `entrypoint` nel comando `docker run`), mentre resteranno sovrascrivibili tutti i parametri di default al comando specificati nel **CMD**.
- Esempio:

```
1. FROM ubuntu:latest
2. ENTRYPOINT du
3. CMD [“-hs”]
```

Docker – Dockerfile WORKDIR

- **WORKDIR**: viene utilizzata per impostare la **working directory**, ovvero la directory all'interno del **container** su cui avranno effetto tutte le successive istruzioni.
- Sintassi:

```
# path assoluto
WORKDIR /path1/path2
# path relativo
WORKDIR path1/path2
```

- E' sempre consigliabile l'utilizzo di **path assoluti**, ma nel caso si utilizzi un **path relativo** esso farà riferimenti all'istruzione **WORKDIR** immediatamente precedente. Inoltre, per definizione, se questa non è fornita esplicitamente, si assume che la **working directory** specificata sia **/**.
- Esempio:

```
1. FROM ubuntu:latest
2. WORKDIR /usr
3. WORKDIR local/src
4. CMD ls -l
```

Docker – Dockerfile LABEL

- **LABEL** permette di aggiungere dei metadati alle nostre immagini, esprimibili come una coppia chiave valore:

```
LABEL "<chiave>"="<valore>" ...
```

- Indica informazioni utili non direttamente legate alla definizione dell'immagine stessa. Ad esempio, è possibile specificare un *maintainer* o indicare un numero di *versione*, una descrizione dell'immagine e così via.
- Ogni istruzione **LABEL** crea un nuovo **layer**, per cui come nel caso dell'istruzione **RUN** è consigliabile unire più istruzioni **LABEL** in una unica che le raggruppi tutte.
- Esempio:

```
LABEL VERSION=”1.0.0” MAINTAINER=”fr76”
```

Docker – Dockerfile EXPOSE

- **EXPOSE** con questa istruzione è possibile dichiarare su quali porte il **container** resterà in ascolto. Non apre direttamente le porte specificate, ma grazie ad essa **Docker** saprà, in fase di avvio dell'immagine, che sarà necessario effettuarne il forwarding.
- Sintassi:

```
EXPOSE <porta_1> [<porta_n>]
```

- È possibile inserire più porte semplicemente aggiungendo uno spazio. Una volta effettuato il build del **container**, all'avvio dello stesso potremo utilizzare il parametro `-P` in modo da esporre le porte dichiarate nell'attributo **EXPOSE.**
- Una volta avviato il **container**, per sapere su quale porta è stata mappata la porta esposta nel **Dockerfile**, abbiamo a disposizione l'istruzione `docker port <porta esposta> <nome del container | id del container>:`

```
# docker port 80 nginx
```

- Esempio:

```
1FROM ubuntu:latest
2EXPOSE 80 443
3CMD apt update && apt install nginx
```

Docker – Dockerfile VOLUME

- **VOLUME** permette di far comunicare fra loro il file system dell'host con quello del **container** e viceversa permettendo di specificare un path all'interno del file system del **container**.
- Sintassi
  - `VOLUME ["path"]`
- Una volta avviato il **container**, è possibile specificare a quale path locale far corrispondere il path definito nel **container**. Tale mapping è realizzabile solo al momento dell'avvio del container e non all'interno del **Dockerfile**, in quanto quest'ultima soluzione creerebbe potenziali incompatibilità.
- Esempio:

```
1.FROM ubuntu:latest
2.EXPOSE 80 443
3.VOLUME ["/var/www"]
4.CMD apt update && apt install nginx
```

Docker – Building images

- Ora che conosciamo **Dockerfile** e alcune istruzioni importanti, cerchiamo di capire come creare immagini **Docker** su misura.
- Il comando fondamentale per la creazione di immagini **Docker** è "`docker build`". Tale comando, partendo da un **Dockerfile**, genererà l'immagine che poi utilizzeremo.
- La sintassi è:

```
# docker build [OPT ...] <build context>
```

- Il parametro **`<build context>`** è **obbligatorio** e fornisce all'engine di build il **relativo contesto**.

Docker – Build context

- Ad esempio, l'istruzione `COPY`, utilizzabile all'interno di un **Dockerfile**, ha come sorgente un `path` obbligatoriamente relativo, e per risolverlo l'engine ha bisogno di conoscerne il contesto: è proprio qui che entra in gioco il `<build context>`
- Tale informazione è esprimibile come un normale percorso di filesystem Unix, (il più delle volte viene usato il carattere punto (`.`)), in modo da specificare la directory corrente come contesto di build
- Il `<build context>` è un concetto abbastanza ostico. Il miglior modo per padroneggiarlo a pieno è sperimentare e capire le varie combinazioni che ne possono derivare.

Docker – Building images: Esempio

- Vogliamo costruire un immagine **Docker** che generi un container per servire pagine web statiche html.
- Supponiamo di creare queste pagine nella nostra home al percorso:

```
WEBPATH=//home//$USER/sitoweb/www
WEBPATH=//C/Users/$USER/sitoweb/www
```

- Per comodità vogliamo partire da un immagine già pronta di **Engine-X**.

Docker – Building images: Esempio

- Cominciamo creando il file “`dockerfile`” nella cartella `$WEBPATH`:

```
# touch dockerfile
```

- Inseriamo nel `dockerfile` la seguente configurazione:

```
FROM nginx:latest
LABEL Author="Fabrizio Romano 767330"
EXPOSE 80
COPY ./www/ /var/www/html
```

Docker – Building images: Esempio

- Così facendo stiamo definendo una nuova immagine a partire dal layer di Engine-X `nginx:latest`.
- Il **container** risultante esporrà la porta `80` in modo da poter mostrare all'esterno i file che abbiamo copiato nella directory `/var/www/html` grazie all'istruzione `COPY`.
- Per effettuare il **build dell'immagine**, ci sposteremo dove è posizionato il `Dockerfile`, ovvero in **`WEBPATH`**, e digiteremo il comando:

```
# docker build -t mywebserver_1:v1 .
```

- Notate che abbiamo specificato anche la versione dell'immagine (`v1`).

Docker – Building images: Esempio

- Attenzione al punto! Spesso non lo si nota, ma una parte fondamentale del comando di `build` è proprio quel “**.**” **che rappresenta il contesto di build**. Questo è un concetto fondamentale in quanto da un senso ai path relativi in cui ci imbatteremo lungo il processo di building.
- Riguardando il `Dockerfile`, infatti, si nota come l’istruzione `COPY` abbia un path relativo `./www/`. Il “.” che abbiamo fornito nel comando build esprime la directory corrente, e siccome siamo in `$WEBPATH`, il path relativo `./www/` corrisponderà proprio al path assoluto `$WEBPATH/www`.

Docker – Building images: Esempio

- Una volta che l'istruzione `docker build` ha avuto luogo, la nuova immagine si troverà nel computer locale insieme a tutte le altre e sarà possibile vederla eseguendo `docker image ls`.
- Per eseguire l'immagine appena costruita, poichè il `Dockerfile` espone la porta utilizzata da `Engine-X` (ovvero la classica 80), basterà utilizzare il parametro `-P` nel comando `docker run`:

```
# docker run -d --name mywebserver_1 -P mywebserver_1
```

- Oppure volendo specificare una porta:

```
# docker run -d --name mywebserver_1 -p 8088:80 mywebserver_1
```

- Come faccio ad accedere al server web appena creato? Una volta scoperto l'ip (ip dell'host ospitante se linux oppure se windows con il comando `docker-machine ip`), basta puntare il browser a quell'indirizzo aggiungendo eventualmente la porta se specificata in `docker run`.

Docker – Docker Compose

- Nel caso un cui sia necessario creare applicazioni che utilizzano piu container si può utilizzare lo strumento **compose** di **Docker**.
- **Compose** è uno strumento per la definizione e l'esecuzione di applicazioni **Docker multi-container**.
- Con **Compose**, si utilizza un file **YAML** per configurare i servizi che compongono l'applicazione. Quindi, con un solo comando,si creano e si avviano tutti i servizi da cui dipende l'applicazione.
- L'utilizzo di **Compose** è fondamentalmente un processo in tre fasi:
  1. Definizione dell'ambiente della applicazione tramite un **`dockerfile`** in modo che possa essere riprodotta ovunque.
  2. Definizione dei servizi che compongono l'applicazione tramite un file **`docker-compose.yml`** in modo che possano essere eseguiti insieme in un ambiente isolato.
  3. Esecuzione di `docker-compose up` per avviare ed eseguire l'intera applicazione.
- `Reference: https://docs.docker.com/compose/compose-file/`

Docker – Esempio di `docker-compose.yml`

```
version: "3.8"
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
      - logvolume01:/var/log
    links:
      - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

Docker – Esempio: Docker Compose e Wordpress

- Installiamo **Docker Compose**:

```
# apt install docker-compose
```

- Creiamo una directory vuota:

```
# mkdir wordpress
```

- Questa directory conterrà l'ambiente dell'immagine dell'applicazione. La directory dovrebbe contenere solo le risorse per costruire quell'immagine tra cui un file `docker-compose.yml` che definisce un'applicazione wordpress di base.
- Creiamo nella directory wordpress un file di nome: `docker-compose.yml`:

```
# cd wordpress
```

```
# touch docker-compose.yml
```

- Inseriamo nel file il seguente contenuto:

Docker – Esempio: Docker Compose e Wordpress

```
version: '3.3'

services:
   db:
     image: mysql:5.7
     volumes:
       - db_data:/var/lib/mysql
     restart: always
     environment:
       MYSQL_ROOT_PASSWORD: somewordpress
       MYSQL_DATABASE: wordpress
       MYSQL_USER: wordpress
       MYSQL_PASSWORD: wordpress

   wordpress:
     depends_on:
       - db
     image: wordpress:latest
     ports:
       - "8000:80"
     restart: always
     environment:
       WORDPRESS_DB_HOST: db:3306
       WORDPRESS_DB_USER: wordpress
       WORDPRESS_DB_PASSWORD: wordpress
       WORDPRESS_DB_NAME: wordpress
volumes:
    db_data: {}
```

Docker – Esempio: Docker Compose e Wordpress

- Il volume **docker** `db_data` mantiene gli aggiornamenti effettuati da WordPress al database.
- WordPress funziona solo sulle porte 80 e 443.
- Eseguiamo `docker-compose up -d` dalla directory del progetto (vedi slide successiva).
- Se si utilizza **Docker Machine**, puoi eseguire il comando `docker-machine ip MACHINE_VM` per ottenere l'indirizzo della macchina, quindi aprire in un browser web.
- Se si utilizza Docker Desktop per Mac o Docker Desktop per Windows, utilizzare `http://localhost` come indirizzo IP e aprire `http://localhost:8000` in un browser web.

Docker – Esempio: Docker Compose e Wordpress

Questo comando esegue `docker-compose` in modalità **detach**, estrae le immagini **Docker** necessarie e avvia i contenitori

wordpress

e database..

```
root@las:~/wordpress# docker-compose up -d
Creating network "wordpress_default" with the default driver
Creating volume "wordpress_db_data" with default driver
Pulling db (mysql:5.7)...
5.7: Pulling from library/mysql
f7ec5a41d630: Pull complete
9444bb562699: Pull complete
6a4207b96940: Pull complete
181cefd361ce: Pull complete
8a2090759d8a: Pull complete
15f235e0d7ee: Pull complete
d870539cd9db: Pull complete
7310c448ab4f: Pull complete
4a72aac2e800: Pull complete
b1ab932f17c4: Pull complete
1a985de740ee: Pull complete
Digest: sha256:e42a18d0bd0aa746a734a49cbbcc079ccdf6681c474a238d38e79dc0884e0ecc
Status: Downloaded newer image for mysql:5.7
Pulling wordpress (wordpress:latest)...
latest: Pulling from library/wordpress
f7ec5a41d630: Already exists
941223b59841: Pull complete
a5f2415e5a0c: Pull complete
b9844b87f0e3: Pull complete
5a07de50525b: Pull complete
caeca1337a66: Pull complete
5dbe0d7f8481: Pull complete
a12730739063: Pull complete
fe0592ad29bf: Pull complete
c3e315c20689: Pull complete
8c5f7fdfcedf: Pull complete
8b40a9fa66d5: Pull complete
81830aebb3f8: Pull complete
7b04d4658443: Pull complete
0e596b6c428e: Pull complete
ec84879c7faf: Pull complete
5f211a0d2061: Pull complete
47c48169dcd4: Pull complete
19b34857b097: Extracting [=====================>                             ]  6.881MB/15.58MB
78810a623bdc: Download complete
25db262383c2: Download complete
```

Docker – Installazione su Linux Ubuntu

- La versione di **Docker** presente nella distribuzione potrebbe non essere aggiornata. Per questo motivo andremo ad aggiungere delle sorgenti al sistema di pacchetti predefinito della distro.
- Anzitutto aggiorniamo l'elenco dei pacchetti e installiamo alcuni pacchetti richiesti:

```
# sudo apt update
# sudo apt install apt-transport-https ca-certificates curl
software-properties-common
```

- Aggiungiamo al sistema il repository ufficiale di **Docker** e la relativa chiave GPG:

```
# curl -fsSL https://download.docker.com/linux/ubuntu/gpg |
sudo apt-key add -
# sudo add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/ubuntu bionic stable"
```

Docker – Installazione su Linux Ubuntu

- Aggiorniamo di nuovo l'elenco dei pacchetti, in modo da caricare nell'elenco anche i pacchetti del repository di Docker:

```
# sudo apt update
```

- Installiamo Docker:

```
# sudo apt install docker-ce
```

- Controlliamo lo stato del demone Docker:

```
# sudo systemctl status docker
```

```
● docker.service - Docker Application Container Engine
   Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
   Active: active (running) since Thu 2018-07-05 15:08:39 UTC; 2min 55s ago
     Docs: https://docs.docker.com
 Main PID: 10096 (dockerd)
    Tasks: 16
   CGroup: /system.slice/docker.service
           ├─10096 /usr/bin/dockerd -H fd://
           └─10113 docker-containerd --config /var/run/docker/containerd/containerd.toml
```

Docker – Installazione su Windows 10

- https://www.docker.com/products/docker-desktop

Docker – Sintassi del comando docker

- Sinteticamente usare **Docker** significa *effettuare chiamate al comando omonimo creando una o più catene formate da comando sottocomando e parametri* seguendo la sintassi:

```
# docker [opzioni] [comando] [argomenti]
```

- Ad esempio per ottenere le informazioni sul sistema docker:

```
# docker info
```

- Per ottenere un elenco dei comandi disponibili digitare:

```
# docker
```

- Per ottenere un aiuto per ogni singolo comando:

```
# docker comando --help
```

Docker – Gestire i container

- I **contenitori Docker** vengono creati da **immagini Docker**.
- Per impostazione predefinita, **Docker** estrae queste immagini da **Docker Hub**, un registro gestito dalla società che ha sviluppato il progetto.
- Chiunque può ospitare le proprie immagini nel **Docker Hub**, quindi la maggior parte delle applicazioni e delle distribuzioni Linux di cui si può bisogno hanno immagini ospitate lì.
- Verifichiamo di poter accedere al **Docker Hub**:

```
# docker run hello-world
```

- Cosa succede?

Docker – Gestire i container

```
(base) C:\Users\Utente>docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:10d7d58d5ebd2a652f4d93fdd86da8f265f5318c6a73cc5b6a9798ff6d2b2e67
Status: Downloaded newer image for hello-world:latest

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/get-started/
```

Docker – Gestire i container

- **Docker** inizialmente non è stato in grado di trovare l'immagine `hello-world` localmente, quindi ha scaricato l'immagine dal **Docker Hub**, che è il repository predefinito. Una volta scaricata l'immagine, **Docker** ha creato un contenitore dall'immagine e ha eseguito l'applicazione all'interno del contenitore visualizzando il messaggio.
- E' possibile cercare le immagini disponibili su **Docker Hub** utilizzando il comando `docker` con il sottocomando di ricerca. Ad esempio, per cercare l'immagine di Ubuntu:

```
# docker search ubuntu
```

Docker – Gestire i container

- Dovreste ottenere un elenco di tutte le immagini che coinvolgono `ubuntu` presenti nel **Docker Hub**.
- Importante: nella colonna **OFFICIAL** la clausola **OK** indica che l'immagine è costruita e mantenuta dalla società che sviluppa il progetto.
- Per scaricare ed installare l'immagine ufficiale di Ubuntu:

```
# docker pull ubuntu
```

- Per eseguire un **container** con l'immagine appena scaricata è sufficiente utilizzare il comando `run`:

```
# docker run ubuntu
```

Docker – Gestire i container

- Per vedere quali immagini sono scaricate sul server **Docker** è sufficiente usare il comando `docker images`:

```
# docker images
```

```
Output
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              latest              113a43faa138        4 weeks ago         81.2MB
hello-world         latest              e38bc07ac18e        2 months ago        1.85kB
```

Docker – Gestire i container

- Per avviare un **container** in modo iterativo e quindi poterci accedere:

```
# docker -it run [-d -p 8080:80] ubuntu
```

- Dovreste ottenere un accesso a terminale del tipo:

```
root@d9b100f2f636:/#
```

  - Dove `d9b100f2f636` indica l'**id** del **container**.

Docker – Gestire i container

- Ora, all'interno del **container** potete fare ciò che volete, ad esempio installare `nodejs`:

```
# apt update
# apt install nodejs
# node -v
```

- Una volta finito, potete uscire dal **container** digitando:

```
# exit
```

- E' possibile sapere quali **container** sono in esecuzione nel sistema tramite il comando:

```
# docker ps
```

Docker – Gestire i container

- Per sapere quali **container** sono presenti nel sistema tramite il comando:

```
# docker ps -a
```

- Per avviare un **container** che non è in esecuzione si utilizza il comando `start` seguito dall'id (o nome) del **container**:

```
# docker start d9b100f2f636
```

- Per fermare un container di utlizza il comando `stop` seguito dal l'id (o nome) del **container**:

```
# docker stop d9b100f2f636
```

- Per cancellare un **container**:

```
# docker rm d9b100f2f636
```

Docker – Gestire i container

- E’ possibile **assegnare un nome** ad un **container** che lo identifica all’interno del sistema. Tale nome, assegnato tramite la clausola `--name` può sostituire l’id del **container** nelle varie operazioni.

```
# docker run -it --name myLinux ubuntu:latest
```

```
See 'docker run --help'.

(base) C:\Users\Utente>docker rm b16100c7ba86f6c07ed2e3f47994717eaece822a3a644c37f6d5b8eb45b197b0
b16100c7ba86f6c07ed2e3f47994717eaece822a3a644c37f6d5b8eb45b197b0

(base) C:\Users\Utente>docker run -it --name myLinux ubuntu:latest
root@c4bfda6e215c:/# ls
bin  boot  dev  etc  home  lib  lib32  lib64  libx32  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@c4bfda6e215c:/#
```

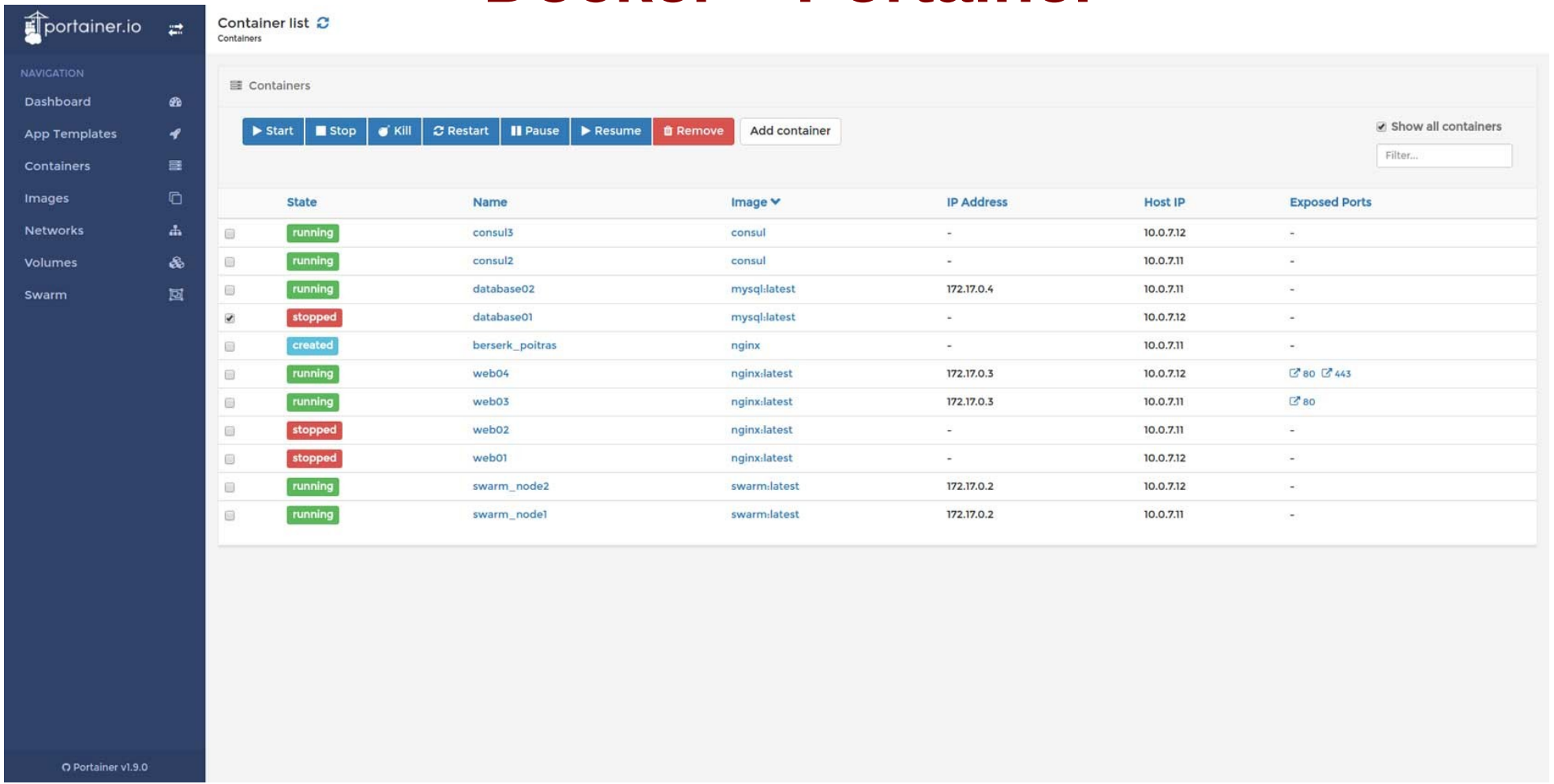
```
# docker ps
```

```
Anaconda Prompt (Miniconda3)

(base) C:\Users\Utente>docker ps
CONTAINER ID   IMAGE           COMMAND       CREATED          STATUS          PORTS     NAMES
c4bfda6e215c   ubuntu:latest   "/bin/bash"   14 seconds ago   Up 13 seconds             myLinux

(base) C:\Users\Utente>
```

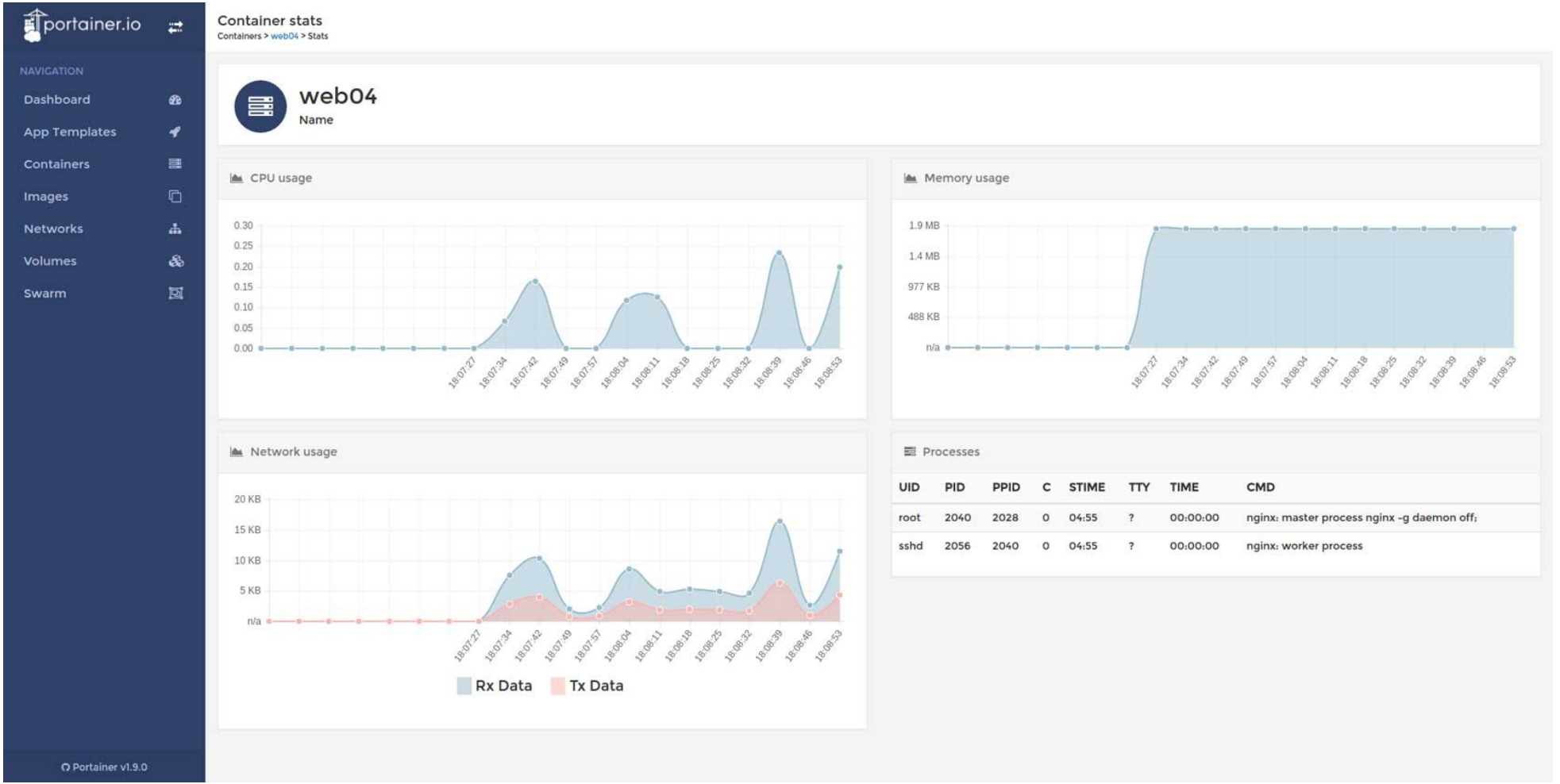
Docker – Portainer

- **Portainer** è una soluzione Open Source e multipiattaforma che mette a disposizione un'interfaccia utente con cui gestire più facilmente **Docker:** consente di amministrare **container**, **immagini**, **network** e volumi attraverso una dashboard in grado di rendere visibili i dettagli relativi a tutte le entità gestibili.
- Funziona su tutti i sistemi operativi o quasi.
- L'**interfaccia di Portainer** consente di visualizzare le statistiche in tempo reale, potendo monitorare l'utilizzo delle risorse fornite da CPU e memoria mentre vengono consumate, oppure verificando lo stato delle attività di networking e dei processi funzionanti all'interno di ciascun **container**.

Docker – Portainer



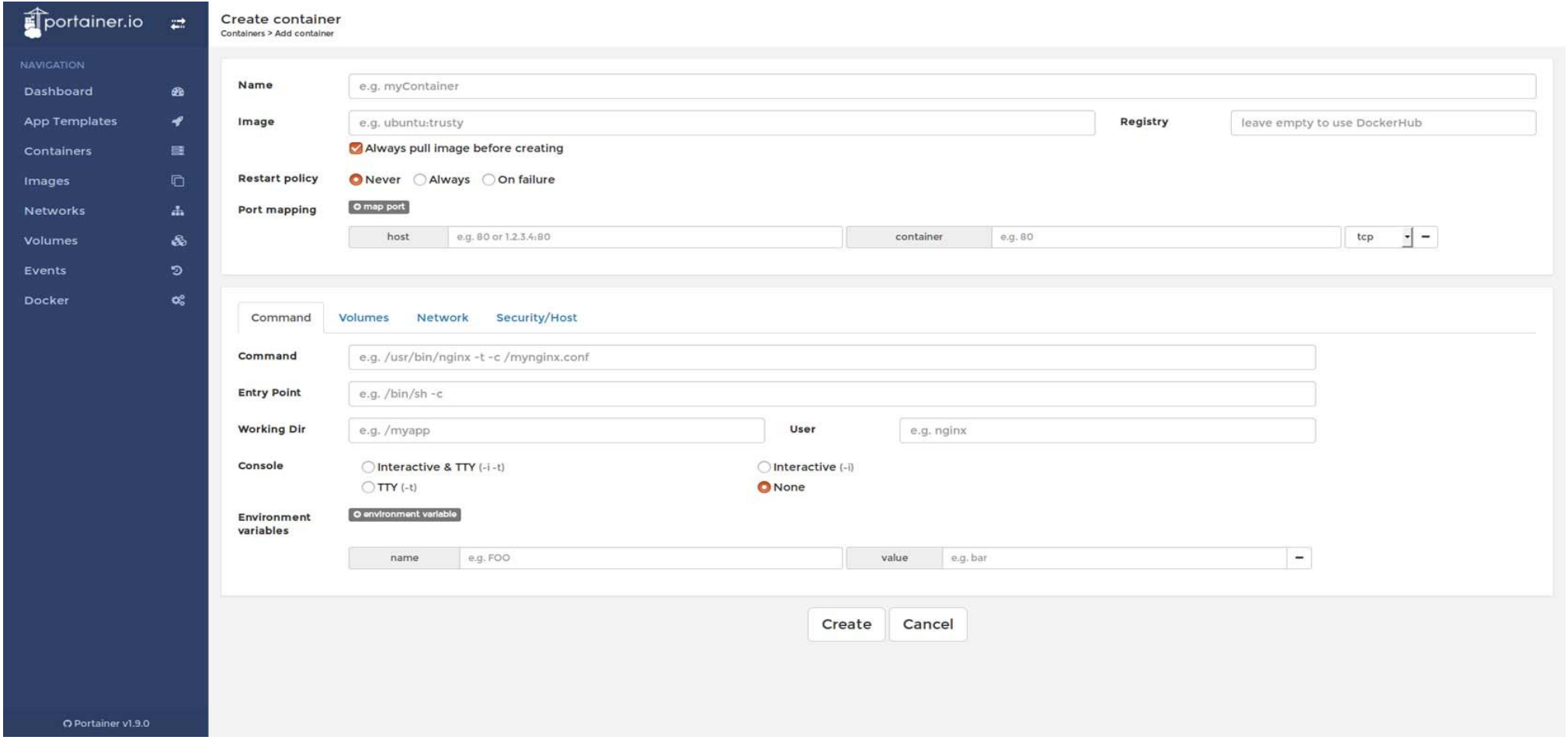
Docker – Portainer



Docker – Portainer

- **Portainer** consente di:
  - Amministrare la struttura esistente.
  - Creare nuovi **container**.
  - Effettuare fasi di deploy semplificati utilizzando un apposito sistema basato sui **template**.
- I **template** contengono diversi i modelli pronti all'uso per la messa in produzione di applicazioni e piattaforme diffusamente utilizzate: database (MySQL, MongoDB, PostgreSQL, MariaDB..), in-memory data store (Redis), CMS (WordPress, Joomla, Drupal..), Web server (Httpd, nginx..), media server e piattaforme per la messaggistica.

Docker – Portainer



Docker – Portainer, Installazione

- **Portainer** è già esistente come immagine **Docker**. Quindi l'installazione consiste nell'effettuare un pull di un nuovo **contenitore**:

```
# docker volume create portainer_data
# docker run -d -p 8000:8000 -p 9000:9000 --name=portainer --
restart=always -v /var/run/docker.sock:/var/run/docker.sock -v
portainer_data:/data portainer/portainer-ce
```

- Ora è sufficiente accedere alla porta `9000` del server **Docker** su cui è in esecuzione **portainer** utilizzando un browser.
- Nota: la porta `9000` è la porta generale utilizzata da **Portainer** per l'accesso all'interfaccia utente. La porta `8000` viene utilizzata esclusivamente dall'agente EDGE per la funzione di tunneling inverso.