



# Who we are



**Alessandra Lombroso**  
Sr. HRBP



**Andrea De Gaetano**  
Manager, Engineering



**Emanuele Biancardi**  
Sr. Manager, Engineering



**Cristiano Spadaro**  
Director, Quality Assurance



**Claudio Giordano**  
Manager, Engineering



**Andrea Briozzo**  
Manager, Engineering



**Francesco Nassano**  
Manager, Engineering



**Alessandra Puddu**  
Manager, Quality Assurance



# Agenda

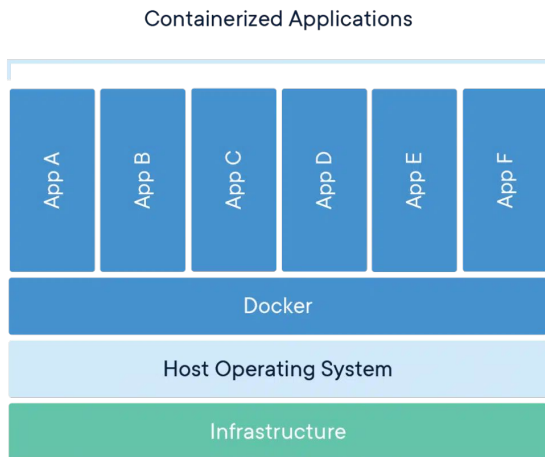
- [01](#) Introduction to Docker
- [02](#) Basics & Prerequisites
- [03](#) A glance into the future
- [04](#) Images & Containers
- [05](#) Data & Volumes
- [06](#) Networking
- [07](#) Build with Github Actions

# Agenda

01 Introduction to Docker

# What is Docker?

Docker è una “container technology”: uno strumento per creare e gestire containers



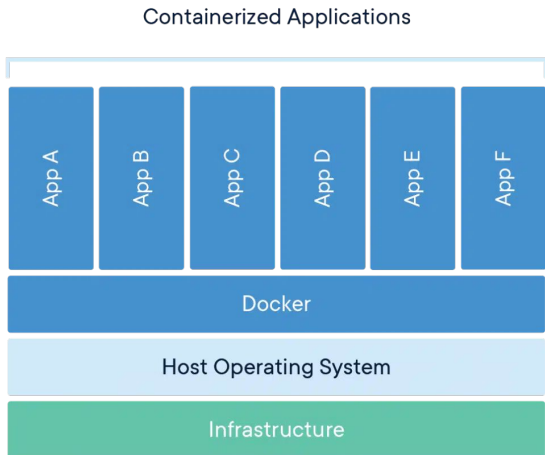
A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. Container images become containers at runtime and in the case of Docker containers – images become containers when they run on [Docker Engine](#).



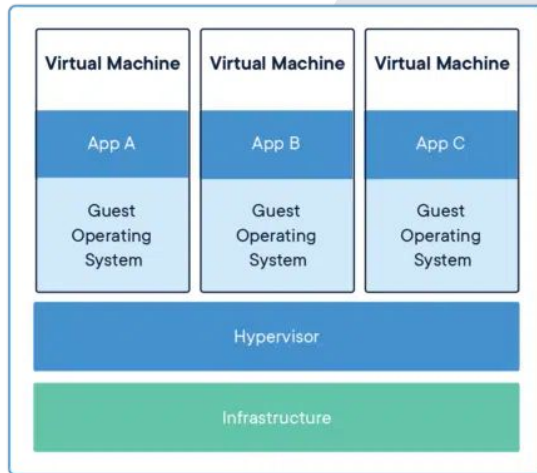
# What is Docker?

## Confrontiamo Containers e Virtual Machines

**Containers** are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine as isolated processes



**Virtual machines** (VMs) are an abstraction of physical hardware turning one server into many servers. Each VM includes a full copy of an operating system



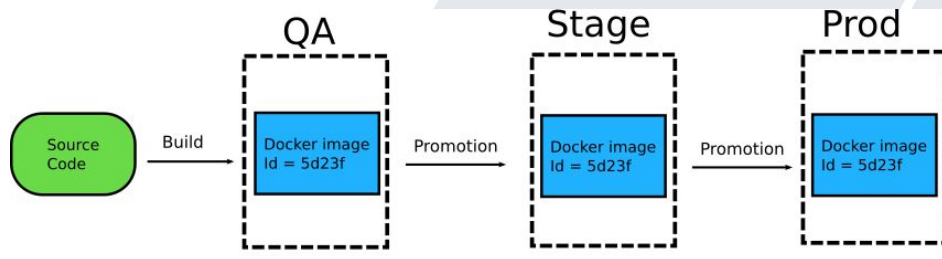
# Why Docker?

## Perché volere “application packages” indipendenti e standardizzati?

Vogliamo costruire ed eseguire l'applicazione sempre **esattamente** nello stesso ambiente.

Un determinato container riproduce sempre la stessa applicazione e lo stesso comportamento. Non importa dove o da chi viene eseguito.

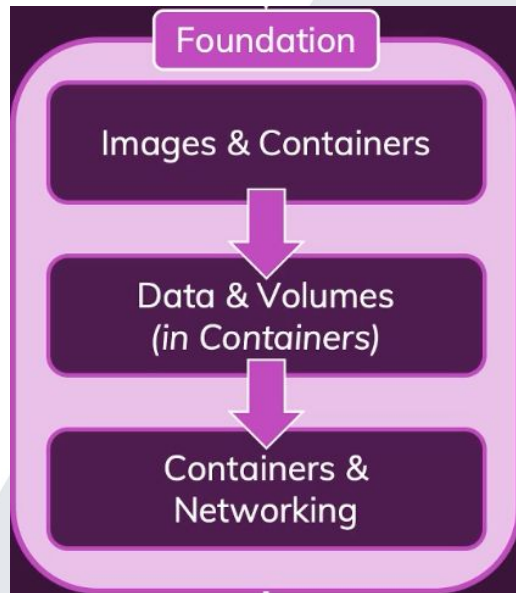
- Stesso esatto ambiente Dev/Prod
- Facile condividere un ambiente di sviluppo comune
- Facile passare da un progetto all'altro, anche con versioni differenti



# Course outline

Cercheremo di capire i primi elementi fondamentali per utilizzare proficuamente Docker

- Immagini e containers per impacchettare la nostra applicazione
- Gestione di persistenze, come memorizzare i nostri dati
- Il networking, come connettere i container e farli comunicare tra loro





# Agenda

## 02 Basics & Prerequisites

# Basic Concepts

**Container:** sono le istanze dal vivo in esecuzione di immagini Docker.

**Image:** contengono il codice sorgente dell'applicazione eseguibile nonché tutti gli strumenti, le librerie e le dipendenze di cui il codice dell'applicazione ha bisogno per l'esecuzione come container.

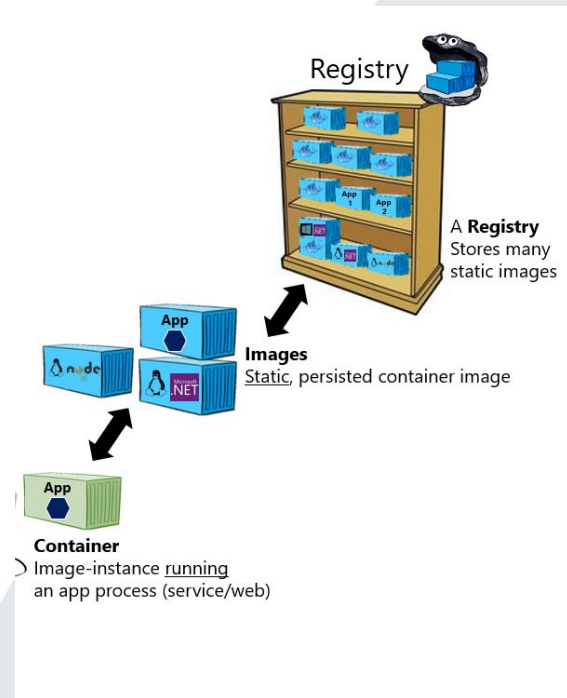
**Registry:** archivio per salvare e condividere immagini.

**Docker Hub:** è il repository pubblico di immagini Docker. Include immagini che sono state prodotte da Docker, Inc., immagini certificate appartenenti al Docker Trusted Registry e molte migliaia di altre immagini.

**DockerFile:** semplice file di testo contenente le istruzioni relative a come creare l'immagine container Docker. È essenzialmente un elenco di istruzioni CLI.

**Docker engine:** è il client/server che costruisce ed esegue containers. E' composto da un daemon, un set di REST APIs, una CLI

**Docker Desktop:** applicazione per Mac, Linux, Win per gestire docker tramite UI in un ambiente virtualizzato



# Prerequisites: Get Docker

Install Docker: <https://docs.docker.com/get-docker/>

Docker Playground: <https://labs.play-with-docker.com/>

## Get Docker

### Docker Desktop terms

Commercial use of Docker Desktop in larger enterprises (more than 250 employees OR more than \$10 million USD in annual revenue) and in government entities requires a paid subscription.

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

You can download and install Docker on multiple platforms. Refer to the following section and choose the best installation path for you.



### Docker Desktop for Mac

A native application using the macOS sandbox security model which delivers all Docker tools to your Mac.



### Docker Desktop for Windows

A native Windows application which delivers all Docker tools to your Windows computer.



### Docker Desktop for Linux

A native Linux application which delivers all Docker tools to your Linux computer.

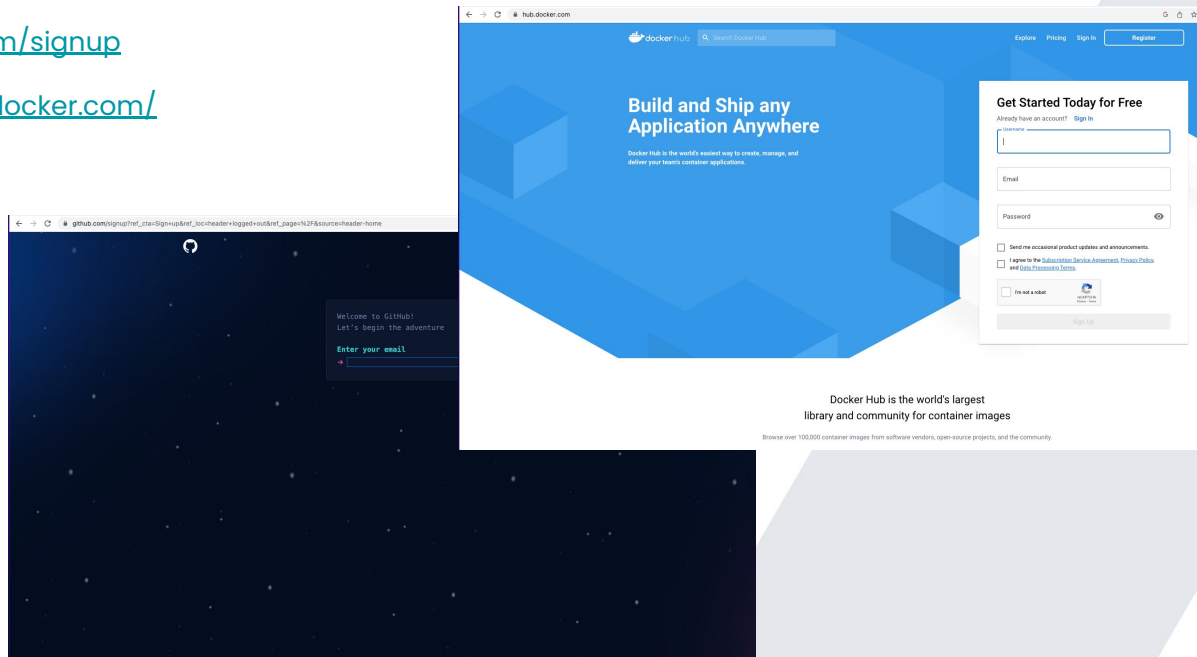


# Prerequisites: Docker Hub & Github

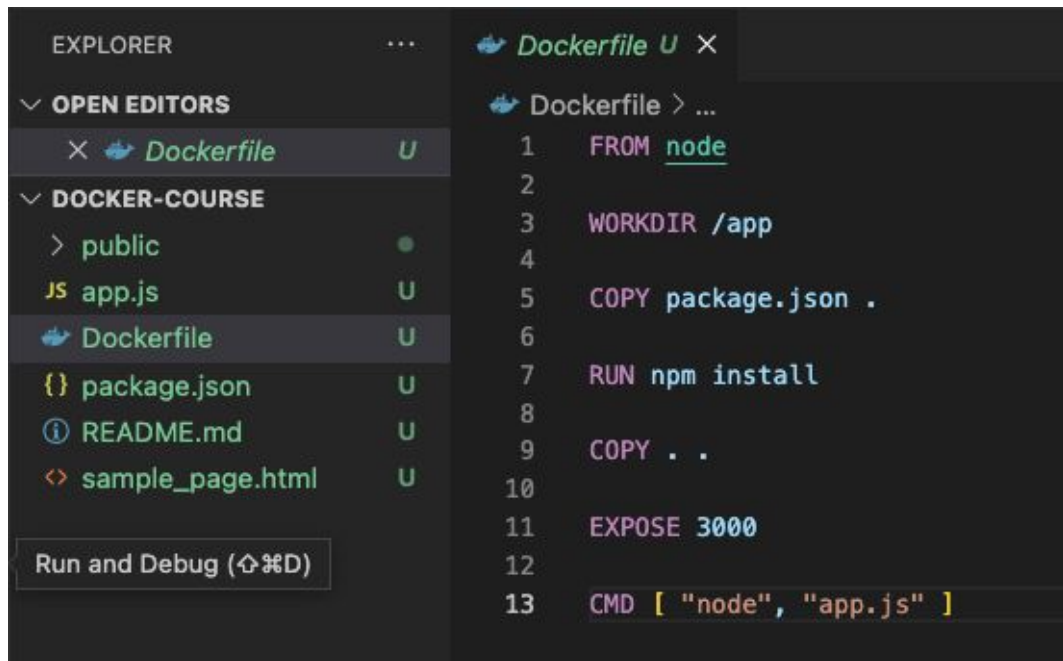
Ci serviranno un paio di account (free)

Github: <https://github.com/signup>

Docker Hub: <https://hub.docker.com/>



# Prerequisiti- Visual Studio Code



Un tipico, minimale, progetto docker è simile a questo. Per editare i file che ci servono è necessario un editor di testo.

Possiamo installare in VSCode alcune utili estensioni:

- **Docker**
- Python/Javascript/whatever

E altre per la customizzazione, come:

- indent-rainbow
- Peacock

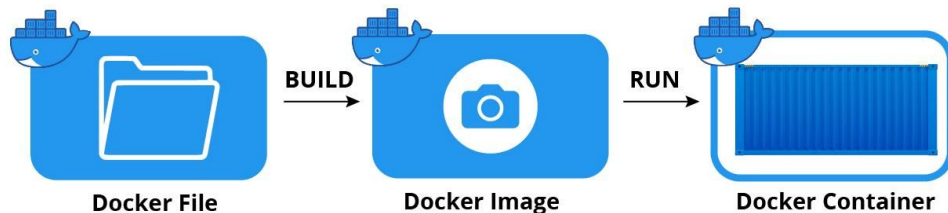


# Agenda

03 A glance into the future

# Build the container's image

Partiamo da un semplice Dockerfile per generare un'immagine pronta ad essere eseguita come container, avremo bisogno di pochi comandi delle docker CLI



**I file che andiamo ad utilizzare sono disponibili in questo repo:**

<https://github.com/vengomatto/docker-course-1>

Docker command-line reference:

<https://docs.docker.com/engine/reference/commandline>

Docker cheat-sheet:

<https://www.docker.com/wp-content/uploads/2022/03/docker-cheat-sheet.pdf>

```
Dockerfile U x
Dockerfile > ...
1 FROM node
2
3 WORKDIR /app
4
5 COPY package.json .
6
7 RUN npm install
8
9 COPY . .
10
11 EXPOSE 3000
12
13 CMD [ "node", "app.js" ]
```



# Docker build and run

Questo è un esempio basico, vediamo

- il comando di **build**, senza opzioni
- Otteniamo l'**identificativo** della nostra nuova immagine, al momento presente solamente sul nostro laptop
- Il comando di **run**, che espone la porta 3000 sul nostro laptop in modo da poter visualizzare l'app nel browser

```
% docker build .
```

```
=> [1/5] FROM docker.io/library/node:14@sha256:d82d512aec5de4fac53b92b2aa148948c2e72264d650de9e1570283d4f503dbe
=> => sha256:7a8bb5c30f836b9486d6a3b5ab3a63ba2b84b836f94383ae4bc3934f45e172a7 7.52kB / 7.52kB 0.0s
=> => sha256:2730d739afad9b8ff3e3029e23fd69d9533603751d6e42053ce0068c2b58e258 50.45MB / 50.45MB 89.8s
=> => sha256:a122751b35336c158fc53a3bb03c6b11b414387589e5455e99baecdd803c6318 7.86MB / 7.86MB 26.1s

.....

=> [2/5] WORKDIR /app 0.5s
=> [3/5] COPY package.json . 0.0s
=> [4/5] RUN npm install 6.1s
=> [5/5] COPY . . 0.0s
=> exporting to image 0.3s
=> => exporting layers 0.3s
=> => writing image sha256:ff164cb53602c181545f92ef046ccd131f453eaa316552d5e1fb6e7c5f3e2a29
```

```
% docker run -p 3000:3000 ff164cb
```





# Voilà!

← → ↻ ⓘ localhost:3000/html

Ciao!

Sembra che stiamo facendo progressi qui!



Se apriamo il nostro browser e andiamo su

<http://localhost:3000/html>

Vediamo renderizzata la nostra pagina html, questo significa che:

- La nostra immagine è stata correttamente generata
- A partire da quella immagine è stato correttamente eseguito un container che ora rimane "running" e disponibile a rispondere alle nostre richieste, sulla porta 3000



# Recap

A questo punto abbiamo imparato alcune cose:

- Cos'è Docker e cosa sono i containers
- A cosa serve e perchè risolve brillantemente alcuni problemi
- Abbiamo predisposto tutto il nostro ambiente di lavoro
- Abbiamo fatto una piccola prova che racchiude alcuni concetti fondamentali

E abbiamo messo alla prova quanto imparato fino a questo momento.

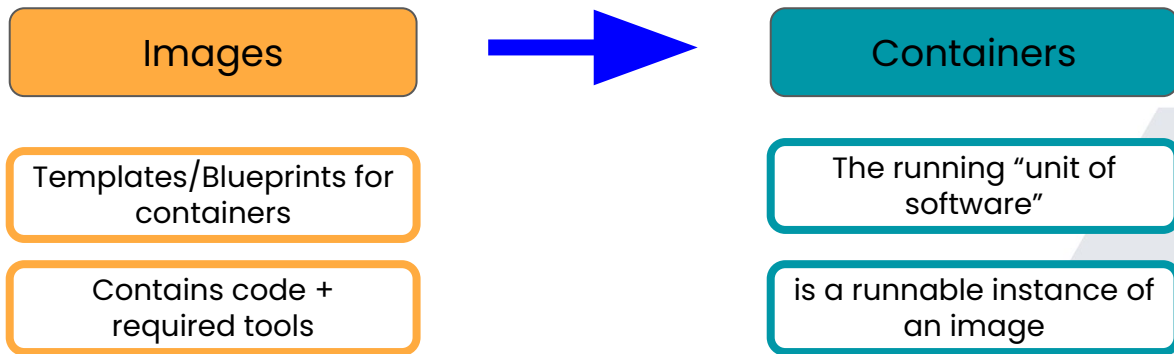
- Con il Dockerfile abbiamo dichiarato, con un linguaggio semplice, ciò che volevamo ottenere
- Il comando di build ha generato l'immagine desiderata
- Il comando di run ha eseguito il container che rende disponibile la nostra semplice applicazione, identica su qualunque piattaforma dotata di Docker



# Agenda

04 Images & Containers

# Images vs Containers



# Finding/Creating Images

Per utilizzare Docker avremo sempre bisogno di una immagine da eseguire, questa immagine può avere due origini:

Use a pre-built  
image

Ex from: Docker Hub, Github  
Packages, AWS ECR

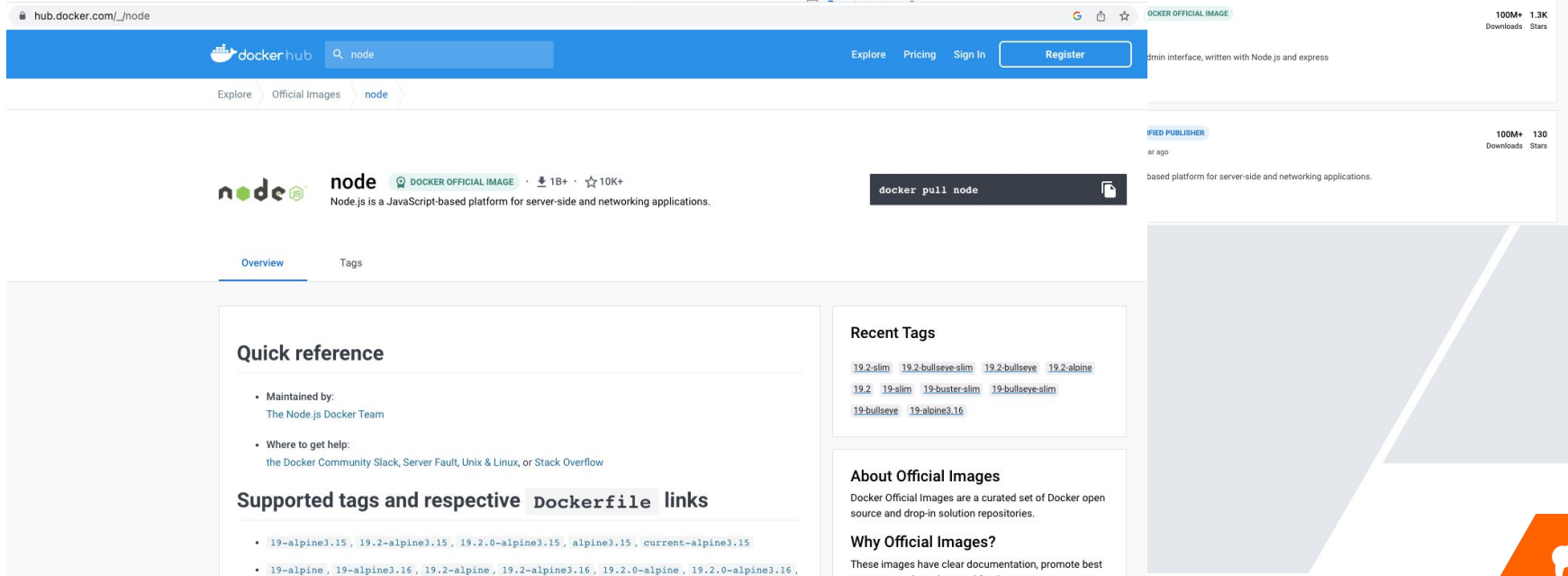
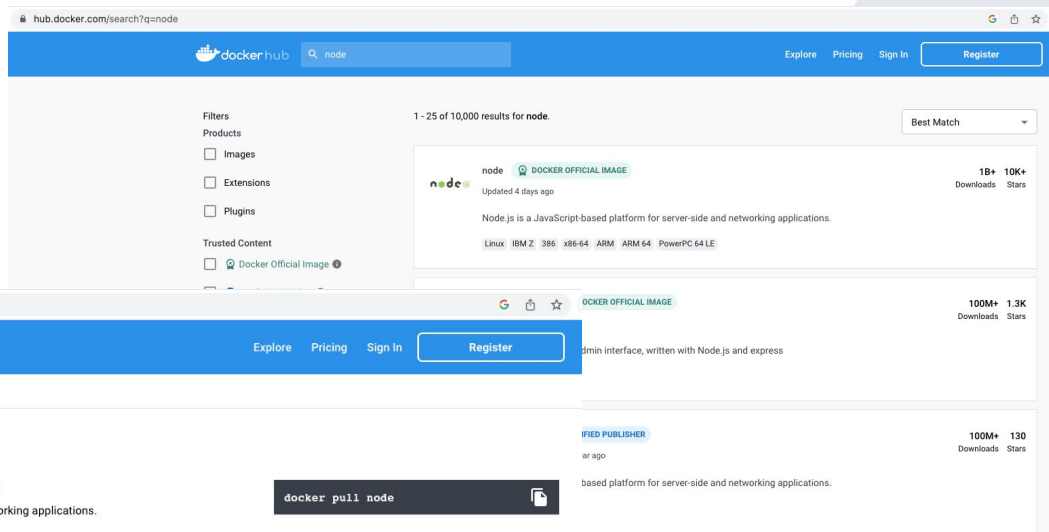
Create your own  
custom image

Write your Dockerfile  
(based on another image)



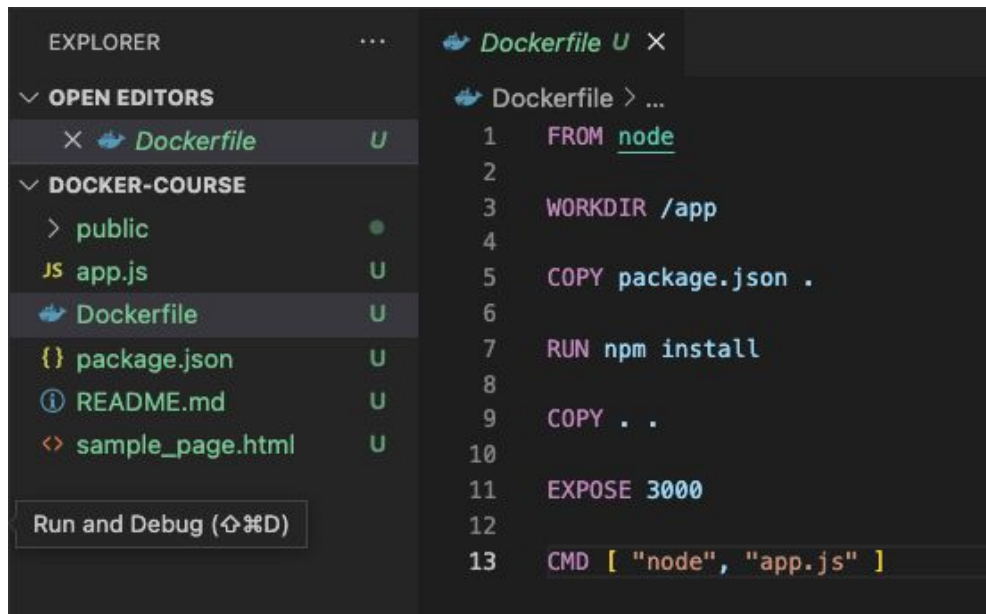
# Pre-built images

Il caso più comune è l'uso di Docker Hub, dove potete trovare l'immagine "[node](#)" che usiamo nei nostri esempi



# Custom Images

Quando invece decidiamo di costruire una nostra immagine personalizzata dovremo iniziare scrivendo un Dockerfile, e per fare questo ripercorriamo in maggior dettaglio l'esempio che abbiamo usato precedentemente



```
1 FROM node
2
3 WORKDIR /app
4
5 COPY package.json .
6
7 RUN npm install
8
9 COPY . .
10
11 EXPOSE 3000
12
13 CMD [ "node", "app.js" ]
```

Dockerfile reference:

<https://docs.docker.com/engine/reference/builder/>

**FROM** - base image

**WORKDIR** - setta la directory di lavoro per ogni istruzione seguente

**COPY** - copia da sorgente al filesystem dell'immagine

**RUN** - esegue il comando in un nuovo layer "on-top" di quelli esistenti

**EXPOSE** - informa docker che il container ascolta sulla specifica porta (ma non la espone realmente, per quello serve "run" con opzione "-p")

**CMD** - fornisce i parametri di default per eseguire il container



# Docker CLI to manage images and containers

Le principali caratteristiche di immagini e containers possono essere efficacemente gestite con le seguenti funzionalità che andremo a vedere brevemente.

## Images

Can be **tagged**  
*-t, docker tag ...*

Can be **listed**  
*docker images*

Can be **analyzed**  
*docker image inspect*

Can be **removed**  
*docker rmi, docker prune*

## Containers

Can be **named**  
*-name*

Can be **configured** in detail  
*-help*

Can be **listed**  
*docker ps*

Can be **removed**  
*docker rm*





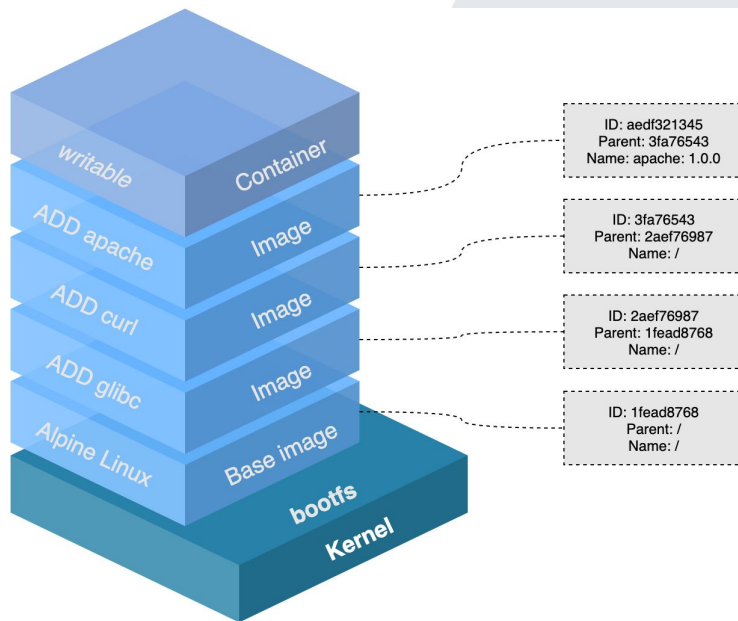
# [OPTIONAL] Image layers

Le immagini Docker sono formate da *livelli* e ogni livello corrisponde a una versione dell'immagine.

Ogni qualvolta uno sviluppatore apporta modifiche all'immagine, viene creato un nuovo livello superiore che sostituisce quello precedente come versione corrente dell'immagine. I livelli precedenti vengono salvati e vengono riutilizzati ad ogni evento di build o anche in altri progetti.

Ogni volta che viene creato un container da un'immagine Docker, viene creato un ulteriore nuovo livello denominato livello container.

Le modifiche apportate al container, come l'aggiunta o l'eliminazione di file, vengono salvate solo nel livello container ed esistono solo mentre il container è in esecuzione.

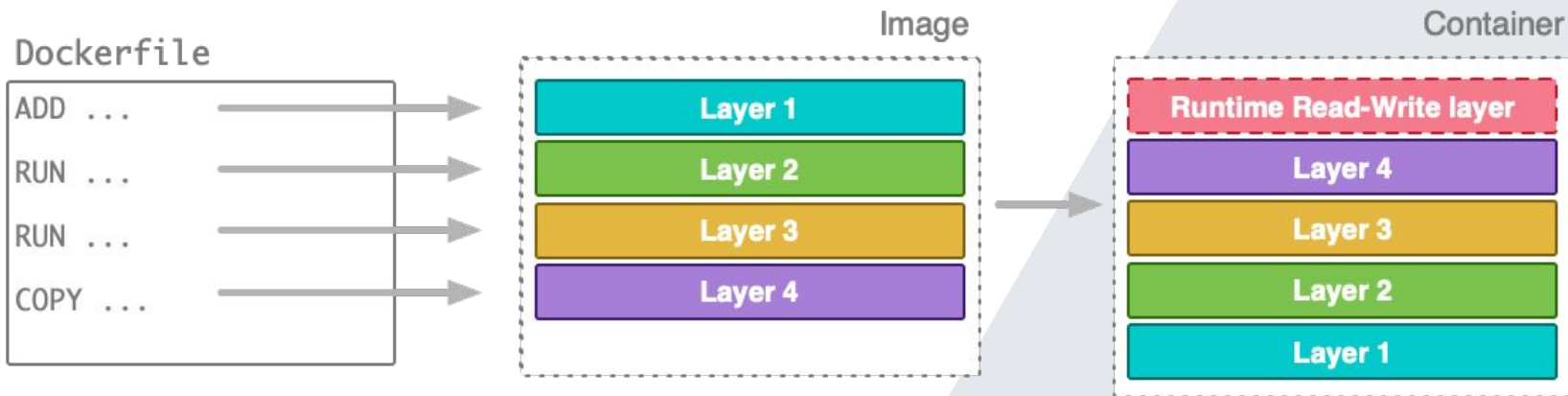


# [OPTIONAL] Image layers

Ogni istruzione contenuta nel Dockerfile crea un layer corrispondente della nostra immagine

Se modifico un layer, ad esempio aggiornando il codice iniettato nell'immagine con il comando COPY, allora quel layer verrà ri-generato e dopo di lui anche TUTTI i layers successivi.

Ne consegue che ordinare opportunamente le istruzioni del Dockerfile può portare ad una grande ottimizzazione



# Pratica 1

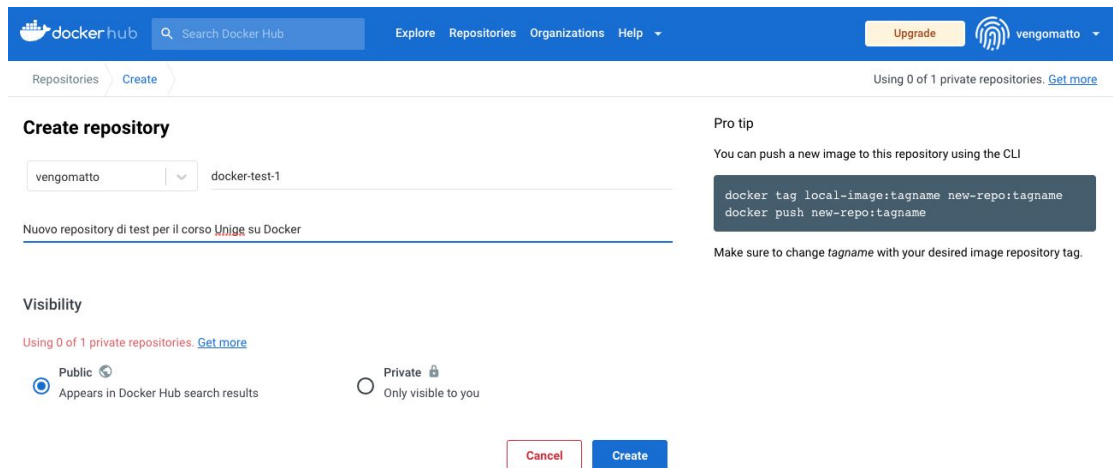
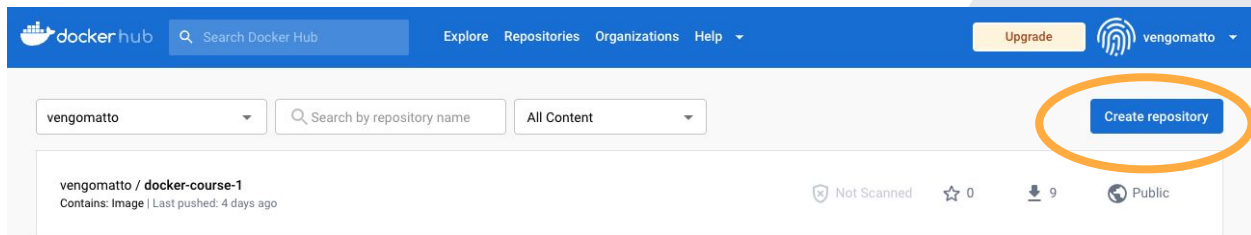
- Clone the repo  
git clone <https://github.com/docker/getting-started.git>
- In the “getting-started/app” folder create a Dockerfile like the following

```
FROM node:18-alpine
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
EXPOSE 3000
```
- Check the node application
- Build the image with a custom tag  
docker build -t first-test .
- Start a container listening on port 3000  
docker run -dp 3000:3000 -name first-test-container first-test
- Open web browser to <http://localhost:3000>
- Check running containers with “docker ps” or the Docker Desktop Dashboard



# Condivisione di immagini

Login su [hub.docker.com](https://hub.docker.com) e creazione di un nuovo repository



Possiamo pubblicare l'immagine su Hub assegnando alla nostra immagine un tag opportuno e poi eseguendo "push"

```
docker login -u YOUR-USER
docker tag <img-name> YOUR-USER/<repo-name>
docker push YOUR-USER/<repo-name>
```



# Condivisione di immagini

Adesso la nostra immagine è disponibile per chiunque e su qualunque piattaforma  
Possiamo sfruttare [Play with Docker](#) per fare una prova su un sistema “pulito” e diverso dal nostro.

```
docker run YOUR-USER/<img-name>
```

The screenshot displays the Play with Docker web interface. On the left, a sidebar shows a digital clock at 03:56:49, a 'CLOSE SESSION' button, and a list of instances with one instance named 'node1' at IP 192.168.0.48. The main panel shows details for a container named 'br0lcrti\_br0ldviosm4g00b718h0' with IP 192.168.0.48. It includes an 'OPEN PORT' button, resource usage (Memory: 0.82%, CPU: 0.54%), an SSH command, and 'DELETE' and 'EDITOR' buttons. At the bottom, a terminal window shows a warning message and a shell prompt.

```
#####  
#                               #  
# WARNING!!!!                   #  
# This is a sandbox environment. Using personal credentials   #  
# is HIGHLY! discouraged. Any consequences of doing so are   #  
# completely the user's responsibilities.                     #  
#                               #  
# The PWD team.                                                #  
#####  
[node1] (local) root@192.168.0.48 -  
$
```



# Pratica 2

Adesso che abbiamo creato un'immagine sul nostro laptop la possiamo condividere, partiamo quindi dall'immagine precedente:

- Esegui il login su Docker Hub
- Crea un repository su Docker Hub, scegli un nome per il tuo progetto/app
- Assegna un tag alla tua immagine locale, adeguato alla naming convention di Docker Hub
- Esegui il push dell'immagine
- Esegui il login su [Docker Playground](#)
- Prova ad eseguire ( "run" ) questa immagine in una istanza nel Docker Playground



# Agenda

05 Data & Volumes

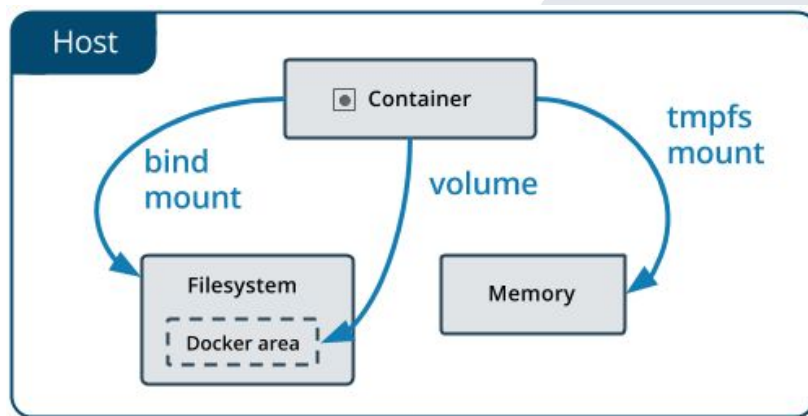
# Persistenza dati

I dati che scriviamo all'interno di un container sono effimeri, e avranno vita solo fino a quando non verrà rimosso il container e saranno comunque accessibili solo dal container stesso.

Esistono due tipi di volumi per risolvere questo problema:

- Volumes
- Bind mounts

Facciamo riferimento a:  
<https://docs.docker.com/storage/>





# Persistenza dati - Volumes

I volumi sono il metodo generalmente da preferire per conservare dati in Docker, possiamo immaginarli come un'area sul filesystem del sistema host gestita da Docker stesso e in cui nessun altro dovrebbe fare modifiche.

## How to create and manage volumes:

```
docker volume create test-vol
```

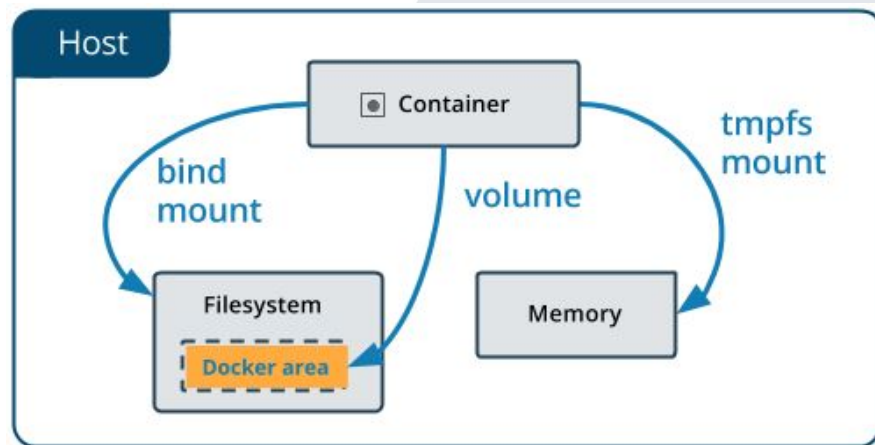
```
docker volume ls
```

```
docker volume inspect test-vol
```

```
docker volume rm test-vol
```

## How to mount a volume:

```
Example: docker run -it -v test-vol:/test-folder alpine
```



# Persistenza dati - Bind mounts

Quando decidiamo di usare i *bind mounts* stiamo montando un file o una cartella del sistema host all'interno del container.

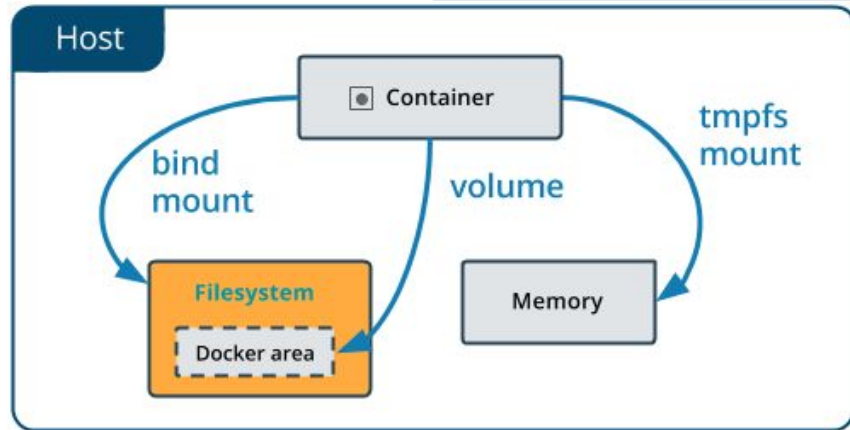
Il caso più ovvio per preferire questa soluzione è lo sviluppo attivo di codice, quando desideriamo che ogni nostra modifica sia immediatamente disponibile e visibile dal container

## How to use a bind mount:

Example:

```
docker run -it -v "$(pwd)"/local-folder:/test-folder alpine
```

```
docker run -it -v "$(pwd)":/usr/share/nginx/html/ -p 3000:80 nginx
```



# Pratica 3

Sperimentiamo l'uso di un volume con la nostra applicazione

- Creiamo un volume vuoto  
`docker volume create datavol2`
- Lanciamo il nostro solito container montando il volume con target `/app`  
*In questo modo testiamo una particolarità dei volumi, se montati su una cartella target già popolata, allora i dati esistenti vengono automaticamente copiati nel volume*  
`docker run -v datavol2:/app -p 3000:3000 docker-course-1`
- [Opzionale] Lanciamo un secondo container, montando il medesimo volume, e modifichiamo la sample page  
`docker run -it -v datavol2:/datacontainer alpine`  
Verifichiamo i cambiamenti dal browser



# Agenda

06 Networking

# Networking

Abbiamo imparato che i containers sono ambienti isolati, ma molto spesso abbiamo necessità di farli dialogare tra loro, pensate al caso molto frequente di una applicazione che ha bisogno del suo database, o di un web server posto davanti al backend node o php.

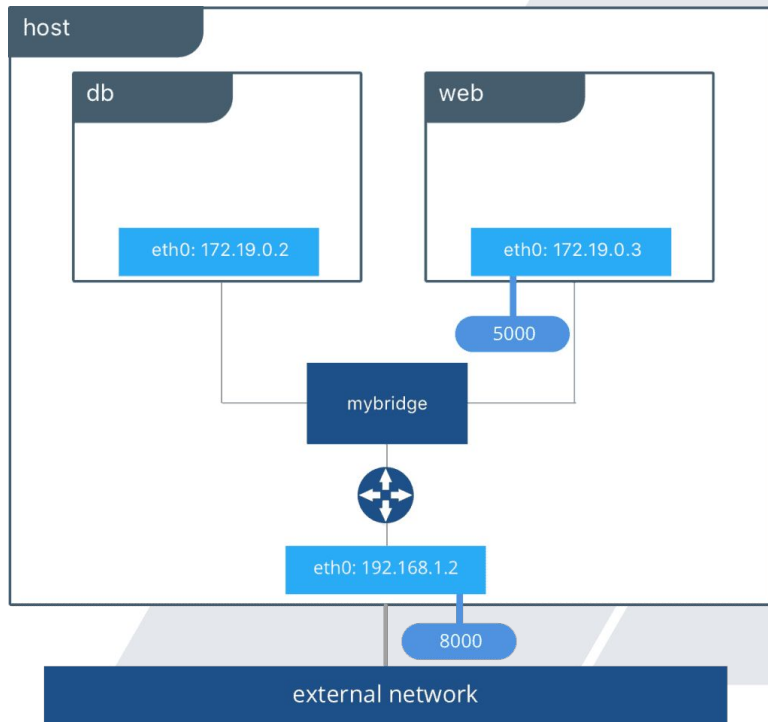
Ci sono 5 drivers di base in Docker per gestire il networking, ma noi andiamo a vedere il caso più tipico (e il default)

Facciamo riferimento a:

<https://docs.docker.com/network/>

## Bridge Networking

- La **default bridge network**, consente una semplice comunicazione container-to-container tramite indirizzo IP, ed è creata di default.
- Una **user-defined bridge network**, che creiamo noi, e che consente la comunicazione tra containers, ma in questo caso usando il container name come hostname, molto più user-friendly!



# Pratica 4

- Crea una user-defined network  
`docker network create devnet1`
- Lancia un container a partire dall'immagine `docker-course-1`, questa volta senza esporre alcuna porta sull'host  
`docker run -d --rm --network devnet1 --name course-1 vengomatto/docker-course-1`
- Verifica che il container sia running, e che effettivamente sia isolato
- Lancia un container `alpine` sulla medesima rete, e verifica che da tale container si possa in effetti dialogare con `docker-course-1`  
`docker run -it --network devnet1 --name alpine-1 alpine`  
`wget -q -O - course-1:3000/html`
- [Opzionale] Lancia un container `nginx` configurato come reverse-proxy davanti alla nostra applicazione e che la renda nuovamente disponibile esponendo la porta 80  
(A questo scopo sfrutta il file di configurazione `nginx_proxy.conf` presente nel progetto [docker-course-1](#))  
`docker run --rm --network devnet1 --name nginx-1 \`  
`-v "$(pwd)/nginx_proxy.conf:/etc/nginx/conf.d/default.conf -p 80:80 nginx:1.18`



# Agenda

xx

Docker CLI

common commands

*- just for reference -*

# Docker CLI

La docker CLI ha molti comandi, ma probabilmente solo un numero limitato di questi sarà di vostro interesse

## Management Commands:

builder	Manage builds
buildx*	Docker Buildx (Docker Inc., v0.7.1)
compose*	Docker Compose (Docker Inc., v2.2.3)
config	Manage Docker configs
container	Manage containers
context	Manage contexts
image	Manage images
manifest	Manage Docker image manifests and manifest lists
network	Manage networks
node	Manage Swarm nodes
plugin	Manage plugins
scan*	Docker Scan (Docker Inc., v0.16.0)
secret	Manage Docker secrets
service	Manage services
stack	Manage Docker stacks
swarm	Manage Swarm
system	Manage Docker
trust	Manage trust on Docker images
volume	Manage volumes

## Commands:

attach	Attach local standard input, output, and error streams ...
build	Build an image from a Dockerfile
commit	Create a new image from a container's changes
cp	Copy files/folders between a container and the local filesystem
create	Create a new container
diff	Inspect changes to files or directories .

events	Get real time events from the server
exec	Run a command in a running container
export	Export a container's filesystem as a tar archive
history	Show the history of an image
images	List images
import	Import the contents from a tarball to create a filesystem image
info	Display system-wide information
inspect	Return low-level information on Docker objects
kill	Kill one or more running containers
load	Load an image from a tar archive or STDIN
login	Log in to a Docker registry
logout	Log out from a Docker registry
logs	Fetch the logs of a container
pause	Pause all processes within one or more containers
port	List port mappings or a specific mapping for the container
ps	List containers
pull	Pull an image or a repository from a registry
push	Push an image or a repository to a registry
rename	Rename a container
restart	Restart one or more containers
rm	Remove one or more containers
rmi	Remove one or more images
run	Run a command in a new container
save	Save one or more images to a tar archive (streamed to STDOUT by default)
search	Search the Docker Hub for images
start	Start one or more stopped containers
stats	Display a live stream of container(s) resource usage statistics
stop	Stop one or more running containers
tag	Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
top	Display the running processes of a container
unpause	Unpause all processes within one or more containers
update	Update configuration of one or more containers
version	Show the Docker version information
wait	Block until one or more containers stop, then print their exit codes





# Docker CLI to manage IMAGES

## How to build an image

Abbiamo visto che è necessario un Dockerfile e il comando

```
docker build .
```

## How to tag an image

E' possibile assegnare un nome all'immagine, che altrimenti avrebbe solo un ID

Un nome immagine è composto da `<image repository>/<image name>:<image tag>`

```
docker build -t <image repository>/<image name>:<image tag> .
```

```
docker tag <image id> <image repository>/<image name>:<image tag>
```

Example: `docker build -t vengomatto/docker-course-1:v1.0.0`

## How to list images

Per elencare le immagini disponibili sul sistema

```
docker images
```

## How to remove an image

Naturalmente anche le immagini possono essere rimosse dal sistema

```
docker rmi <image id>
```



# Docker CLI to manage IMAGES

## How to remove unused images

Quando avremo un elenco di immagini riconoscibili solo tramite il loro ID è difficile eseguire una pulizia manuale, pertanto si usa

```
docker image prune
```

O anche il più generico

```
docker system prune
```

Che provvede ad una pulizia generale del sistema

```
WARNING! This will remove:
- all stopped containers
- all networks not used by at least one container
- all dangling images
- all build cache
Are you sure you want to continue? [y/N] y
```



# Docker CLI to manage CONTAINERS

## How to run a container

Abbiamo già visto con il nostro esempio iniziale che per eseguire un container è necessario utilizzare:

```
docker run <image name>
```

Example: `docker run node`

## How to publish a port

E sappiamo anche che i container sono ambienti isolati e per rendere l'applicazione al loro interno raggiungibile è necessario pubblicare una porta con:

```
--publish <host port>:<container port>
```

Example: `docker run -p 3000:80 node`

## How to name a container

I containers hanno, di default, un ID e un nome generato casualmente da docker. È possibile specificare il nome del container con l'opzione `--name`

Example: `docker run -d -p 3000:80 --name my_node_container node`



# Docker CLI to manage CONTAINERS

## How to list containers

Per visualizzare quali containers sono al momento in esecuzione sul nostro laptop possiamo usare:

```
docker ps
```

## How to stop a container

I containers in foreground si possono stoppare con ctrl+c

Mentre quelli in background hanno bisogno di:

```
docker stop <container id>
```

```
FNASSAN020M~/GIT(:|✓) % docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS                NAMES
82993bbd44ce   654       "docker-entrypoint.s..." 19 seconds ago Up 18 seconds   0.0.0.0:3000->3000/tcp sad_payne
FNASSAN020M~/GIT(:|✓) %

FNASSAN020M~/GIT(:|✓) %
FNASSAN020M~/GIT(:|✓) % docker stop sad_payne
sad_payne
FNASSAN020M~/GIT(:|✓) %
```



# Docker CLI to manage CONTAINERS

## How to start/restart a container

Possiamo voler fare il reboot di un container running o potremmo voler far partire di nuovo un container stopped (ma non rimosso)

```
docker start <container id>  
docker restart <container id>
```

## How to remove a container

Abbiamo capito che containers stoppati rimangono nel sistema, occupando spazio e potenzialmente entrando in conflitto con nuovi containers, possiamo quindi rimuoverli definitivamente

```
docker rm <container id>
```

## Detached mode

I container possono essere avviati in due modalità, **default foreground** mode o **detached** mode.

Nel primo caso il container resta in primo piano “attached” al terminale, e noi possiamo usare ctrl-c per chiuderlo e uscire.

Per far lavorare il container in background possiamo includere l’opzione `—detach` o `-d`

In questo caso il terminale non riceverà output e noi dovremo utilizzare altri comandi per visualizzare i dettagli di nostro interesse.

```
Example: docker run -d -p 3000:80 node
```



# Docker CLI to manage CONTAINERS

## How to show logs of a detached container

Per visualizzare i logs (in genere l'output) di un container detached

```
docker logs <container id>
```

Example: `docker logs -f -n 10 -t <container id>`

Dove `-f` sta per `-follow`, `-n 10` il numero di righe da visualizzare a partire dal fondo del log, `-t` mostra i timestamps

## Interactive mode

Infine potremmo voler eseguire un container in modalità interattiva, ad esempio eseguendo una immagine di Alpine potremmo volere una shell per lavorare in un ambiente linux completamente isolato

```
docker run -it alpine
```

Dove `-i` sta per interactive, `-t` per tty (pseudo shell)

```
FNASSANO20M~/AAA_LOCAL/DOCKER/DOCKER-COURSE-1(main|✓) % docker run -it alpine
/ # uname -a
Linux 7c45885abfc 5.10.76-linuxkit #1 SMP Mon Nov 8 10:21:19 UTC 2021 x86_64 Linux
/ # █
```



# Docker CLI to manage CONTAINERS

## How to run commands in a container

Possiamo voler eseguire comandi all'interno di un running container

`docker exec -d <image id> <command>` per eseguire un comando in background e tornare alla shell dell'host

`docker exec -it <image id> <command>` uso tipico per aprire una shell dentro il container

Example: `docker exec -it alpine bash`



# Docker CLI

## Nota

L'azienda Docker ha intrapreso l'iniziativa di raggruppare i comandi della CLI per feature e concetto, nel tentativo di razionalizzarli.

Pertanto quelli che abbiamo visto sono le loro versioni classiche, ma è possibile che in futuro vadano per la maggiore le loro alternative più recenti.

Alcuni esempi sufficienti a comprendere la ratio di questo cambiamento:

`docker ps` → `docker container ls`

`docker run` → `docker container run`

`docker images` → `docker image ls`

`docker build` → `docker image build`





# Agenda

07 **Build with Github Actions**

# Build with a pipeline

**GitHub Actions** è una popolare piattaforma CI/CD per build, test and deployment, ed è un ottimo modo per automatizzare la generazione delle vostre immagini, e facendo sì che la fase di build sia parte di una pipeline più complessa (e completa).

Segnaliamo due pagine di documentazione come possibile riferimento:

- <https://docs.github.com/en/actions/publishing-packages/publishing-docker-images>
- <https://docs.docker.com/build/ci/github-actions/>

Le Actions fornite da Docker e che utilizziamo nel nostro workflow sono documentate a questi indirizzi:

- **Docker Login**: sign in to a Docker registry.
- **Docker Metadata action**: extracts metadata from Git reference and GitHub events.
- **Build and push Docker images**: build and push Docker images with BuildKit.

```
sample_page.html  JS app.js  ! docker-image.yml  Dockerfile

.github > workflows > ! docker-image.yml > ...
You, 50 minutes ago | 1 author (You) | GitHub Workflow (github-workflow.json)
# Refer to: https://docs.github.com/en/actions/publishing-packages/publishing-docker-images

1
2
3 name: Docker Image CI
4
5 on:
6   release:
7     types: [published]
8
9 # Customization: we add these env variables
10 env:
11   DOCKER_HUB_NAMESPACE: vengomatto
12   DOCKER_HUB_REPOSITORY: docker-course-1
13
14 jobs:
15
16   push_to_registry:
17     name: Build/Push Docker image to Docker Hub
18     runs-on: ubuntu-latest
19     steps:
20       - name: Check out the repo
21         uses: actions/checkout@v3
22
23       - name: Log in to Docker Hub
24         uses: docker/login-action@f054a8b539a109f9f41c372932f1ae047eff08c9
25         with:
26           username: ${ secrets.DOCKER_USERNAME }
27           password: ${ secrets.DOCKER_PASSWORD }
28
29       # Customization: we specify explicitly which account and repository
30       - name: Extract metadata (tags, labels) for Docker
31         id: meta
32         uses: docker/metadata-action@98669ae865ea3cfffbcbaa878cf57c20bbf1c6c38
33         with:
34           images: ${ env.DOCKER_HUB_NAMESPACE }/${ env.DOCKER_HUB_REPOSITORY }
35
36       - name: Build and push Docker image
37         uses: docker/build-push-action@ad44023a93711e3deb337508980b4b5e9bcdec5dc
38         with:
39           context: .
40           push: true
41           tags: ${ steps.meta.outputs.tags }
42           labels: ${ steps.meta.outputs.labels }
43
```



# Assignment

Il nostro obiettivo è mettere alla prova tutto quanto abbiamo imparato

- Crea un nuovo repo su Github
  - [Create a repo](#)
  - [Import your project to github](#)
- **Seleziona una applicazione di tua scelta**
  - Esempio1: [docker-course-2](#)
  - Esempio2: [scheduling-app-with-react-nodejs-emailjs](#) - con approfondimenti sull'app in questo [blog](#)
- Scrivi il Dockerfile
- Scrivi il GitHub Workflow per pubblicare l'immagine su Github Packages
  - [Publishing images to GitHub Packages](#)
- Esegui git commit e push dei tuoi files
- Segui il processo di build su Github Actions
- Apri una sessione/istanza su Docker Playground
- Lancia un container a partire dalla tua immagine
- Esponi la porta relativa e verifica il suo funzionamento



# Thank you!



@LifeAtHudl



Hudl

# Template



