Docker

Capitolo 0

Se volete un corso specifico fatto molto bene : Docker per comuni mortali - Morrolinux

Capitolo 1 – Cos’è Docker

Docker è una piattaforma per creare, distribuire e eseguire **container**.
I container permettono di eseguire applicazioni in ambienti isolati, leggeri e portabili, senza la necessità di macchine virtuali complete.

1.1 Differenze tra Container e VM

Caratteristica	VM	Container
Isolamento	Completo (OS separato)	Processo separato, stesso OS
Peso	Pesante (GB per VM)	Leggero (MB per container)
Avvio	Lento (minuti)	Veloce (secondi)
Portabilità	Limitata al hypervisor	Elevata, qualsiasi host Docker
Risorse	Dedicate	Condivise con host

1.2 Componenti Principali di Docker

- **Docker Engine**
  Servizio principale che esegue container e gestisce immagini.
- **Docker CLI**
  Strumento a linea di comando per interagire con Docker.
- **Docker Daemon**
  Processo in background che gestisce container e immagini.
- **Docker Hub / Registry**
  Repository pubblico o privato per salvare e distribuire immagini.

1.3 Vantaggi dell’uso di Docker

- **Portabilità**: stesso container su qualsiasi host con Docker.
- **Isolamento**: applicazioni indipendenti tra loro.
- **Efficienza**: occupa meno risorse di una VM completa.
- **Ripetibilità**: stessi container identici in sviluppo, test e produzione.
- **Integrazione CI/CD**: ideale per pipeline DevOps.

1.4 Caso d’uso tipico

1. Uno sviluppatore crea un’applicazione web.
2. Scrive un **Dockerfile** per definire ambiente e dipendenze.
3. Costruisce un’immagine e la carica su Docker Hub.
4. In produzione, il container viene eseguito senza preoccuparsi del sistema operativo host.

1.5 Conclusione

Docker permette di astrarre l’applicazione dall’infrastruttura sottostante, rendendo:

- Sviluppo più veloce
- Deploy più semplice
- Gestione di ambienti coerente e sicura

Nel prossimo capitolo entreremo nel dettaglio dell’**architettura di Docker**, spiegando come Engine, Daemon e registri lavorano insieme.

Capitolo 2 – Architettura di Docker

Per utilizzare Docker in modo efficace, è fondamentale comprendere la sua architettura e come interagiscono i vari componenti.

2.1 Componenti Principali

Docker Engine

- È il cuore di Docker.
- Si occupa di eseguire container e gestire immagini.
- Funziona come **server client-daemon**: la CLI comunica con il daemon per eseguire le operazioni.

Docker Daemon (`dockerd`)

- Processo in background che gestisce container, immagini, volumi e reti.
- Ascolta richieste dalla CLI o da API.
- Avvia container secondo le specifiche delle immagini.

Docker CLI (`docker`)

- Interfaccia a linea di comando per interagire con Docker.
- Permette operazioni come:
  - Creare, avviare e fermare container
  - Costruire e scaricare immagini
  - Gestire volumi e network

Docker Registries

- Repository per immagini Docker.

- **Docker Hub**: registry pubblico più diffuso.
- È possibile avere registry privati per sicurezza o ambienti interni.

2.2 Come Funzionano i Container

Un container è un processo isolato che condivide il kernel del sistema operativo host, ma ha:

- Filesystem isolato
- Network virtuale
- Volumi per persistenza dati
- Limiti risorse configurabili (CPU, memoria)

Vantaggi:

- Avvio immediato
- Minore overhead rispetto a una VM completa
- Facile da distribuire

2.3 Immagini Docker

- Un'immagine è un **modello di container immutabile**.
- Contiene:
  - Sistema operativo minimo
  - Applicazioni
  - Dipendenze
- Le immagini possono essere **layered**, permettendo riuso ed efficienza.
- Esempio di layer:
  - `ubuntu:22.04`
  - Installazione Python
  - Installazione librerie app
  - Codice applicativo

2.4 Networking Docker

- **Bridge network**: default per container sullo stesso host.
- **Host network**: il container usa direttamente la rete dell'host.
- **Overlay network**: per container distribuiti su più host (swarm o Kubernetes).

2.5 Volumi e Persistenza

- I container sono effimeri: i dati all’interno vengono persi se il container viene rimosso.
- **Volumi**: spazio persistente gestito da Docker.
- **Bind mount**: collegamento diretto a cartelle dell’host.

2.6 Flusso Operativo Base

1. La CLI invia comandi al Docker Daemon.
2. Il Daemon verifica immagini disponibili.
3. Se necessario, scarica l’immagine dal registry.
4. Avvia il container con filesystem, network e volumi configurati.
5. Il container esegue il processo richiesto in isolamento.

2.7 Conclusione

Comprendere l’architettura Docker permette di:

- Pianificare correttamente deployment e rete
- Gestire volumi e dati in sicurezza
- Ottimizzare immagini e risorse
- Evitare errori comuni di isolamento e rete

Nel prossimo capitolo vedremo come **installare e configurare Docker** su Linux, pronto per l’uso in produzione.

Capitolo 3 – Installazione e Configurazione Base di Docker

In questo capitolo vediamo come installare **Docker Engine** su Debian/Ubuntu usando lo script ufficiale, e come configurarlo in modalità **rootless**.

Per altre distribuzioni Linux, consultare la documentazione ufficiale: Docker Engine Install.

3.1 Prerequisiti

- Sistema operativo Debian o Ubuntu
- Utente con privilegi `sudo` (non necessario per modalità rootless, ma utile per installazione standard)
- Connessione Internet

3.2 Installazione automatica con lo script ufficiale

Docker fornisce uno script ufficiale che configura automaticamente repository, chiavi GPG e installa i pacchetti necessari.

Esegui i seguenti comandi:

```
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh ./get-docker.sh
```

- Lo script installerà Docker Engine, CLI, containerd e plugin di Docker Compose.
- Il processo include anche l’attivazione del servizio Docker.

3.3 Avvio e verifica del servizio

Dopo l’installazione:

```
sudo systemctl enable docker
sudo systemctl start docker
sudo systemctl status docker
```

Verifica che Docker funzioni correttamente eseguendo un container di test:

```
docker run hello-world
```

Se l’immagine viene scaricata e viene stampato un messaggio di conferma, Docker è correttamente installato.

3.4 Uso di Docker senza sudo

Per consentire a un utente di usare Docker senza `sudo` (modalità standard):

```
sudo usermod -aG docker $USER
```

Effettua il logout/login o esegui:

```
newgrp docker
```

Verifica:

```
docker info
```

3.5 Modalità Rootless (Docker senza privilegi root)

Docker può essere eseguito in modalità **rootless**, evitando completamente l'uso di privilegi root. Questa modalità aumenta la sicurezza riducendo l'esposizione del daemon.

Installazione rootless

```
sudo apt update
sudo apt install -y uidmap dbus-user-session
dockerd-rootless-setuptool.sh install
```

- L'installazione crea un daemon Docker gestito dall'utente corrente.
- Docker rootless esegue container con UID dell'utente, isolati dal sistema.

Avvio e utilizzo

```
systemctl --user start docker
systemctl --user enable docker
```

Verifica:

```
docker info
```

> Per dettagli avanzati, limiti e configurazioni aggiuntive, consultare la documentazione ufficiale:
>
> Rootless Docker

3.6 Altre distribuzioni Linux

Se si utilizza un sistema diverso da Debian/Ubuntu (RHEL, CentOS, Fedora, SLES, Raspberry Pi OS, ecc.), seguire la guida ufficiale:

Install Docker Engine – Docker Docs

3.7 Buone pratiche post-installazione

- Tenere Docker aggiornato con il gestore pacchetti o lo script ufficiale
- Limitare l’accesso al gruppo `docker` a utenti trusted
- Verificare lo stato del daemon con `systemctl status docker`
- Testare sempre con `docker run hello-world` dopo aggiornamenti
- Considerare la modalità rootless per ambienti multi-utente o produzione più sicura

---

Conclusione

Docker Engine è ora pronto all’uso, sia in modalità standard che rootless.
Nel prossimo capitolo vedremo come **gestire immagini Docker**, costruirle e personalizzarle con Dockerfile.

Capitolo 4 – Gestione delle Immagini Docker e Dockerfile

In questo capitolo vedremo come:

- Cercare immagini Docker
- Scaricare e gestire immagini locali
- Creare immagini personalizzate
- Scrivere un Dockerfile
- Ottimizzare le immagini

4.1 Cosa sono le immagini Docker

Un'immagine Docker è un template immutabile utilizzato per creare container.

Le immagini:

- Sono composte da layer (livelli)
- Sono versionate tramite tag
- Possono essere archiviate in registry pubblici o privati

Il registry pubblico predefinito è Docker Hub:
https://hub.docker.com/

4.2 Ricerca di un'immagine

Per cercare un'immagine nel registry:

```
docker search nginx
```

L'output mostrerà:

- NAME
- DESCRIPTION
- STARS
- OFFICIAL

Le immagini ufficiali sono contrassegnate come `OFFICIAL`.

4.3 Scaricare un'immagine

Per scaricare un'immagine:

```
docker pull nginx
```

Per scaricare una versione specifica:

```
docker pull nginx:1.25
```

Se non si specifica un tag, Docker utilizza automaticamente `latest`.

4.4 Visualizzare le immagini locali

Per vedere le immagini presenti nel sistema:

```
docker images
```

Oppure:

```
docker image ls
```

L'output mostra:

- REPOSITORY
- TAG
- IMAGE ID
- CREATED
- SIZE

4.5 Eliminare un'immagine

Per rimuovere un'immagine:

```
docker rmi nginx
```

Oppure tramite ID:

```
docker rmi <image_id>
```

Per forzare la rimozione:

```
docker rmi -f nginx
```

4.6 Creare un'immagine personalizzata con Dockerfile

Un Dockerfile è un file di testo che contiene le istruzioni per costruire un'immagine personalizzata.

Esempio base:

```
FROM debian:12

RUN apt update && apt install -y nginx

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

Salvare il file con nome:

```
Dockerfile
```

4.7 Costruire un'immagine

Dalla directory dove si trova il Dockerfile:

```
docker build -t mio-nginx:1.0 .
```

Spiegazione:

- `-t` assegna nome e tag
- `.` indica il contesto di build (directory corrente)

Verifica:

```
docker images
```

4.8 Avviare un container dalla propria immagine

```
docker run -d -p 8080:80 --name webtest mio-nginx:1.0
```

Parametri utilizzati:

- `-d` esecuzione in background
- `-p 8080:80` mappatura porta host → container
- `--name` assegna un nome al container

Aprire nel browser:

```
http://localhost:8080
```

4.9 Comprendere i layer delle immagini

Ogni istruzione come:

- RUN
- COPY
- ADD

genera un layer.

Per visualizzare la cronologia:

```
docker history mio-nginx:1.0
```

Meno layer inutili significano immagini più leggere ed efficienti.

4.10 Ottimizzazione delle immagini

Unire i comandi RUN

Meglio scrivere:

```
RUN apt update && \
    apt install -y nginx && \
    apt clean && \
    rm -rf /var/lib/apt/lists/*
```

Usare immagini leggere

Esempi:

- alpine
- debian:slim

Multi-stage build (concetto base)

Permette di costruire in uno stage e copiare solo il necessario nell'immagine finale.

Esempio:

```
FROM golang:1.22 AS builder
WORKDIR /app
COPY . .
RUN go build -o app

FROM debian:12-slim
WORKDIR /app
COPY --from=builder /app/app .
CMD ["./app"]
```

Questo riduce la dimensione finale dell'immagine.

4.11 File .dockerignore

Permette di escludere file dal contesto di build.

Esempio:

```
.git
node_modules
*.log
```

4.12 Best Practice

- Specificare sempre il tag dell'immagine
- Evitare `latest` in produzione
- Ridurre il numero di layer
- Eliminare file temporanei durante la build
- Utilizzare immagini ufficiali quando possibile

Conclusione

Ora sai:

- Gestire immagini Docker
- Creare Dockerfile
- Costruire immagini personalizzate
- Ottimizzare le build

Nel prossimo capitolo vedremo la gestione avanzata dei container, inclusi volumi, reti e limiti di risorse.

Capitolo 5 – Gestione Avanzata dei Container

In questo capitolo vedremo:

- Gestione avanzata dei container
- Volumi Docker
- Reti Docker
- Variabili d’ambiente
- Limiti di risorse (CPU e RAM)
- Policy di riavvio

5.1 Gestione dei container

Elencare i container

Container attivi:

```
docker ps
```

Tutti i container (anche fermati):

```
docker ps -a
```

Avviare un container fermo

```
docker start nome_container
```

Fermare un container

```
docker stop nome_container
```

Riavviare un container

```
docker restart nome_container
```

Eliminare un container

```
docker rm nome_container
```

Forzare la rimozione:

```
docker rm -f nome_container
```

5.2 Log dei container

Visualizzare i log:

```
docker logs nome_container
```

Seguire i log in tempo reale:

```
docker logs -f nome_container
```

5.3 Accesso alla shell di un container

Entrare in un container in esecuzione:

```
docker exec -it nome_container /bin/bash
```

Se bash non è disponibile:

```
docker exec -it nome_container /bin/sh
```

5.4 Variabili d’ambiente

Passare variabili al container:

```
docker run -d -e MYSQL_ROOT_PASSWORD=secret mysql
```

Oppure tramite file `.env`:

```
docker run --env-file .env nginx
```

5.5 Volumi Docker

I volumi permettono di salvare dati persistenti fuori dal container.

Creare un volume

```
docker volume create mio_volume
```

Elencare i volumi:

```
docker volume ls
```

Utilizzare un volume

```
docker run -d \
  -v mio_volume:/var/lib/mysql \
  mysql
```

In questo modo i dati non vengono persi se il container viene eliminato.

Rimuovere un volume

```
docker volume rm mio_volume
```

5.6 Bind Mount (cartella host)

Montare una cartella del sistema host:

```
docker run -d \
  -v /home/user/dati:/app/dati \
  nginx
```

Differenza:

- Volume → gestito da Docker
- Bind mount → directory reale del sistema host

5.7 Reti Docker

Elencare le reti

```
docker network ls
```

Creare una rete personalizzata

```
docker network create mia_rete
```

Avviare container nella stessa rete

```
docker run -d --name app --network mia_rete nginx
docker run -d --name db --network mia_rete mysql
```

I container nella stessa rete possono comunicare usando il nome del container come hostname.

5.8 Limiti di risorse

Limitare la memoria:

```
docker run -d --memory="512m" nginx
```

Limitare la CPU:

```
docker run -d --cpus="1.0" nginx
```

Questo evita che un container utilizzi tutte le risorse del sistema.

5.9 Policy di riavvio

Riavviare automaticamente il container:

```
docker run -d --restart unless-stopped nginx
```

Opzioni disponibili:

- no
- always
- on-failure
- unless-stopped

5.10 Ispezione e statistiche

Dettagli completi di un container:

```
docker inspect nome_container
```

Utilizzo risorse in tempo reale:

```
docker stats
```

Best Practice

- Usare volumi per dati persistenti
- Limitare CPU e RAM in ambienti di produzione
- Utilizzare reti personalizzate per isolamento
- Impostare policy di riavvio per servizi critici
- Monitorare log e risorse regolarmente

Conclusione

Ora sai:

- Gestire container in modo avanzato
- Usare volumi e bind mount
- Configurare reti personalizzate
- Limitare risorse
- Configurare riavvii automatici

Nel prossimo capitolo vedremo Docker Compose per gestire applicazioni multi-container.

Capitolo 6 – Docker Compose

In questo capitolo vedremo:

- Cos’è Docker Compose
- Installazione del plugin Compose
- Struttura di un file docker-compose.yml
- Gestione di applicazioni multi-container
- Comandi principali
- Variabili d’ambiente
- Best practice

6.1 Cos’è Docker Compose

Docker Compose è uno strumento che permette di definire e gestire applicazioni multi-container tramite un file YAML.

Con un solo comando è possibile:

- Creare reti
- Creare volumi
- Avviare più container
- Gestire dipendenze tra servizi

È ideale per ambienti di sviluppo e piccole/medie infrastrutture.

6.2 Verifica installazione

Docker Compose è incluso come plugin nelle versioni recenti di Docker.

Verifica:

```
docker compose version
```

Se il comando risponde con una versione, Compose è correttamente installato.

6.3 Struttura base di docker-compose.yml

Esempio semplice con Nginx:

```
services:
  web:
    image: nginx:latest
    ports:
      - "8080:80"
```

Salvare il file come:

```
docker-compose.yml
```

6.4 Avviare un progetto Compose

Dalla directory dove si trova il file:

```
docker compose up -d
```

Parametri:

- `up` → crea e avvia i servizi
- `-d` → modalità detached (background)

Verifica container attivi:

```
docker compose ps
```

6.5 Fermare e rimuovere i servizi

Fermare i container:

```
docker compose stop
```

Fermare e rimuovere container, reti e configurazioni:

```
docker compose down
```

Rimuovere anche i volumi:

```
docker compose down -v
```

6.6 Esempio applicazione multi-container (Web + Database)

Esempio con Nginx e MySQL:

```
services:
  web:
    image: nginx:latest
    ports:
      - "8080:80"
    depends_on:
      - db

  db:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: esempio
    volumes:
      - db_data:/var/lib/mysql

volumes:
  db_data:
```

In questo esempio:

- Viene creata automaticamente una rete
- I servizi possono comunicare usando il nome del servizio (es. `db`)
- Il volume `db_data` salva i dati del database

6.7 Variabili d’ambiente con file .env

Creare un file `.env` nella stessa directory:

```
MYSQL_ROOT_PASSWORD=superpassword
```

Nel file docker-compose.yml:

```
environment:
  MYSQL_ROOT_PASSWORD: ${MYSQL_ROOT_PASSWORD}
```

Compose caricherà automaticamente le variabili dal file `.env`.

6.8 Ricostruire i servizi

Se modifichi il Dockerfile:

```
docker compose up -d --build
```

Forzare la ricreazione dei container:

```
docker compose up -d --force-recreate
```

6.9 Visualizzare i log

Log di tutti i servizi:

```
docker compose logs
```

Seguire i log in tempo reale:

```
docker compose logs -f
```

Log di un singolo servizio:

```
docker compose logs web
```

6.10 Scalare un servizio

Esempio:

```
docker compose up -d --scale web=3
```

Questo avvia 3 istanze del servizio web.

Best Practice

- Non usare `latest` in produzione
- Separare ambienti (dev, staging, prod)
- Usare file `.env` per password e configurazioni
- Versionare sempre il file docker-compose.yml
- Utilizzare volumi per dati persistenti
- Limitare CPU e RAM nei servizi critici

Conclusione

Ora sai:

- Cos’è Docker Compose
- Gestire applicazioni multi-container
- Usare variabili d’ambiente
- Scalare servizi

- Gestire reti e volumi automaticamente

Nel prossimo capitolo vedremo come mettere in produzione un'applicazione Docker con reverse proxy e HTTPS.