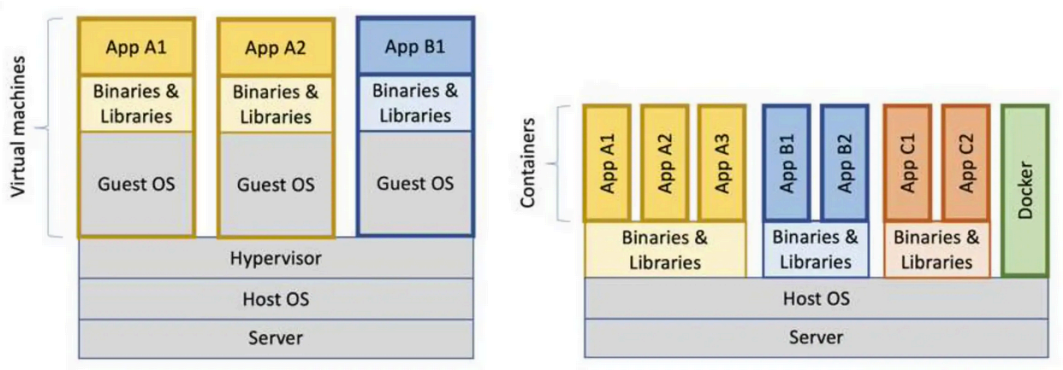
# *Introduzione alla Containerizzazione*

## Tipi di Deploy

### *Container vs VM*




**La densità del deployment** si riferisce alla misura di quante istanze di una particolare applicazione o servizio possono essere distribuite ed eseguite all’interno di un dato ambiente.

**Una alta densità del deployment** significa che ci sono più carichi di lavoro che girano su meno risorse, spesso portando a una migliore utilizzo dell’infrastruttura disponibile, ma può anche presentare sfide relative alla contendibilità delle risorse, alle prestazioni e alla scalabilità.

#### ***Distribuzione Bare Metal (Nessuna isolamento, Bassa densità)***

Inizialmente, le applicazioni venivano distribuite su server fisici in cui più applicazioni condividevano le stesse risorse hardware. Questo portava a **conflitti tra librerie, dipendenze e esigenze di prestazioni**. Una soluzione era **riservare server fisici separati per ciascuna applicazione**, ma questo comportava **alti costi, risorse sotto-utilizzate e un aumento della manutenzione**.

#### ***Distribuzione Virtualizzata (Isolamento, Media densità)***

La virtualizzazione ha introdotto un modo più efficiente per utilizzare le risorse creando più macchine virtuali (VM) su un singolo server fisico. Questo avviene attraverso un **hypervisor**, che gestisce le risorse hardware sottostanti (CPU, memoria, archiviazione) e consente a più VM di funzionare in modo indipendente.

Ogni VM contiene il proprio sistema operativo e stack applicativo. Questo approccio migliora la scalabilità, riduce i costi hardware e fornisce isolamento tra le applicazioni. Tuttavia, il sovraccarico di eseguire sistemi operativi separati per ciascuna VM rimane uno svantaggio.

#### ***Distribuzione di Container (Isolamento, Alta densità)***

I container sono unità software leggere e autonome che girano direttamente sul sistema operativo dell’host, eliminando la necessità di un sistema operativo separato per ciascuna applicazione. Essi impacchettano tutto ciò di cui un’applicazione ha bisogno per funzionare, incluse librerie, dipendenze e binari, fornendo **portabilità, efficienza e isolamento** senza il sovraccarico delle VM. Come regola generale, i container forniscono isolamento delle applicazioni mentre riducono il consumo di risorse.

## Chroot: il primo tentativo di containerizzazione

Il comando **chroot** nei sistemi operativi Unix-like cambia la directory radice apparente per un processo e i suoi figli. Questo essenzialmente “imprigiona” il processo all’interno di una directory specificata, rendendo impossibile per il processo accedere a file al di fuori di quella directory. Questo è comunemente chiamato “chroot jail”.

- **Ambiente minimale:** Quando un processo viene eseguito all’interno di un ambiente chroot, perde l’accesso alle librerie di sistema, binari e utility situati al di fuori del NEWROOT. È necessario garantire che gli strumenti essenziali (ad es. /bin/bash, librerie, /etc/passwd) siano disponibili nell’ambiente chrooted.

- **Non è una misura di sicurezza completa**: Sebbene chroot possa isolare un processo, non è infallibile. Gli utenti root all’interno di un ambiente chroot possono potenzialmente rompere la restrizione e accedere ad altre parti del sistema. Per un’adeguata isolamento, altre tecniche come container (ad es. Docker) o macchine virtuali sono più sicure.

Mostriamo le dipendenze dei comandi ls e bash:

```
↳> ldd /bin/ls
    linux-vdso.so.1 (0x0000702f32aff000)
    libcap.so.2 => /usr/lib/libcap.so.2 (0x0000702f32aa5000)
    libc.so.6 => /usr/lib/libc.so.6 (0x0000702f328b4000)
    /lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2 (0x0000702f32b01000)

↳> ldd /bin/bash
    linux-vdso.so.1 (0x000072b8f1860000)
    libreadline.so.8 => /usr/lib/libreadline.so.8 (0x000072b8f16c5000)
    libc.so.6 => /usr/lib/libc.so.6 (0x000072b8f14d4000)
    libncursesw.so.6 => /usr/lib/libncursesw.so.6 (0x000072b8f1465000)
    /lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2 (0x000072b8f1862000)
```

Possiamo costruire una jail organizzando le dipendenze necessarie come nel sistema host:

```
jail
├── bin
│   ├── bash
│   └── ls
├── lib64 -> usr/lib64
└── usr
    ├── lib
    │   ├── libcap.so.2
    │   ├── libc.so.6
    │   ├── libncursesw.so.6
    │   └── libreadline.so.8
    └── lib64
        └── ld-linux-x86-64.so.2
```

Ora possiamo eseguire entrambi i comandi (imballati con tutte le loro dipendenze ma senza un kernel) all’interno di una gabbia chroot:

```
$ sudo chroot jail /bin/ls
$ sudo chroot jail /bin/bash
```

## Tecnologie dei Container

Ecco alcune tecnologie dei container che offrono diverse capacità per la containerizzazione, ognuna adatta a requisiti diversi:

- **Docker:** La piattaforma di containerizzazione più popolare, che consente agli sviluppatori di impacchettare applicazioni e le loro dipendenze in container.

- **Podman:** Un’alternativa open-source a Docker senza daemon, focalizzata sulla sicurezza. È completamente compatibile con l’interfaccia a riga di comando di Docker, offrendo funzionalità simili senza la necessità di un demone centrale.

- **Kubernetes**: Non è una tecnologia di container ma una potente piattaforma di orchestrazione dei container che automatizza la distribuzione, la scalabilità e la gestione di applicazioni containerizzate. Kubernetes funziona senza problemi con Docker e altri runtime di container.

## Casi d’uso Chiave

### ***Microservizi e Isolamento delle Applicazioni***

I container sono ideali per sviluppare architetture basate sui microservizi, che decompongono grandi applicazioni monolitiche in servizi (più piccoli) indipendentemente distribuiti:

- **Decoupling:** Ogni servizio può essere impacchettato nel proprio container, funzionando in modo indipendente.
- **Ambientazioni Consistenti:** I container garantiscono coerenza tra gli ambienti di sviluppo, test e produzione.
- **Isolamento di Risorse e Guasti**: I container forniscono isolamento, il che significa che un guasto di un servizio non influisce direttamente sugli altri. Questo rende i container eccellenti per sistemi tolleranti ai guasti.
- **Scalabilità**: I container possono essere scalati in modo indipendente per gestire carichi variabili. Gli strumenti di orchestrazione (ad es. Kubernetes) semplificano la gestione della scalabilità e del bilanciamento del carico.
- **Rollback:** Poiché i container sono immutabili, tornare a una versione precedente è semplice in caso di errori nel deployment.

## Registri dei Container

**Le immagini dei container sono memorizzate in registri**, **che fungono da repository dove gli sviluppatori possono spingere e prelevare immagini.** I registri risparmiano tempo centralizzando la gestione delle immagini attraverso gli ambienti. Possono essere:

- Pubblici (ad es. Docker Hub): Accessibili da chiunque.
- Privati (ad es. registri aziendali): L’accesso è ristretto e include controlli di sicurezza aggiuntivi.

I registri popolari includono:

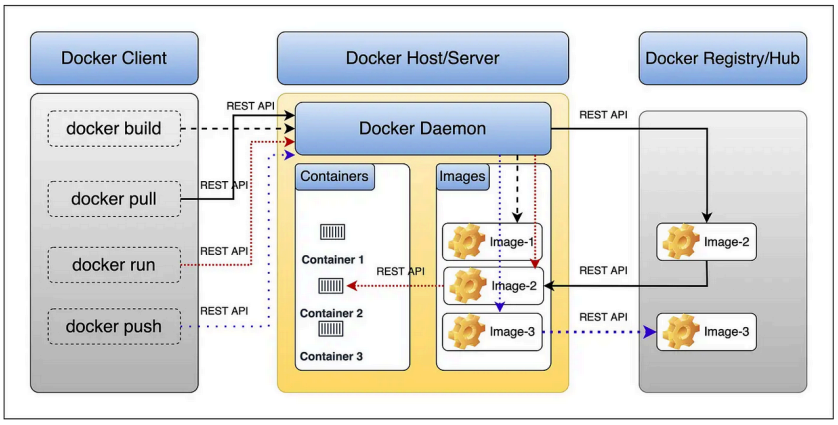
- **Docker Hub:** Il registro pubblico più ampiamente utilizzato.
- **Amazon ECR:** Integrato con AWS, offre funzionalità avanzate come la scansione delle vulnerabilità.
- **Azure Container Registry**: Fornisce geo-replica per un accesso più rapido in tutto il mondo.
- **Google Artifact Registry:** Parte di Google Cloud, ottimizzato per i servizi Google.

### ***Formati delle Immagini***

Sono in uso diversi formati di immagini dei container oggi, ciascuno con i suoi vantaggi:

- **OCI (Open Container Initiative):** Uno standard aperto ampiamente adottato supportato da Docker, Kubernetes e altre piattaforme.
- **Formato immagine di Docker**: Ancora ampiamente utilizzato, fornendo compatibilità con i sistemi legacy.
- **Formato immagine di Singularity (SIF):** Progettato per ambienti di calcolo ad alte prestazioni.
- **Formato immagine di LXD:** Utilizzato principalmente per container di sistema che necessitano di virtualizzazione leggera.

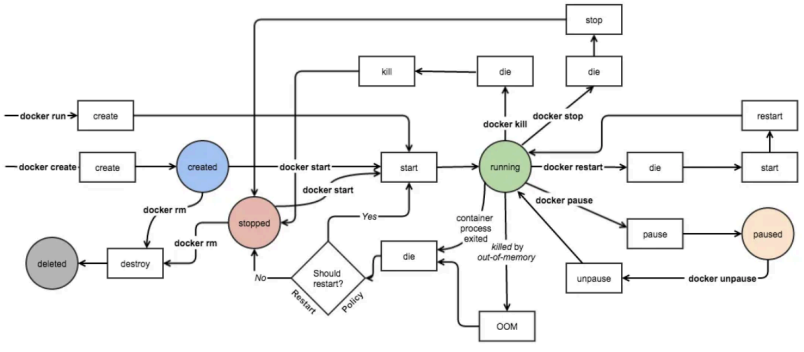
# Architettura di Docker




Docker è una piattaforma open-source che consente agli sviluppatori di automatizzare il deployment, la scalabilità e la gestione delle applicazioni all’interno dei container. L’architettura di Docker è composta da diversi componenti chiave:

- **Docker CLI:** L’interfaccia a riga di comando (CLI) è il principale modo con cui gli utenti interagiscono con Docker. Consente agli utenti di emettere comandi per gestire container, immagini, reti e volumi Docker. La CLI funge da interfaccia user-friendly per gli sviluppatori per comunicare con il demone Docker sottostante.
- **Docker Engine:** Il componente centrale di Docker che gira sulla macchina host. Il demone è responsabile dell’implementazione, esecuzione e gestione dei container. Ascolta le richieste API di Docker e gestisce oggetti Docker come immagini, container, reti e volumi.
- **Docker Registry:** Un registro Docker è un servizio per memorizzare e distribuire immagini Docker. Funziona come un repository dove gli utenti possono spingere, prelevare e gestire immagini Docker. Registri pubblici come Docker Hub sono disponibili e gli utenti possono anche impostare registri privati per uso interno all’interno delle organizzazioni.
- **Immagini:** Le immagini Docker sono pacchetti leggeri, autonomi ed eseguibili che includono tutto il necessario per eseguire un pezzo di software, come il codice dell’applicazione, runtime, librerie, variabili ambientali e file di configurazione. Le immagini sono immutabili e possono essere versionate.
- **Container:** Le applicazioni che girano all’interno dei container sono isolate l’una dall’altra. Molteplici applicazioni possono funzionare contemporaneamente all’interno dei loro container, fornendo un ambiente robusto per lo sviluppo e la produzione.

# Ciclo di Vita del Container




# File Chiave di Docker

**Dockerfile:** è un blueprint per creare un’immagine Docker. Specifica l’immagine base, i comandi per installare dipendenze, i file da copiare, le variabili ambientali da impostare e il comando da eseguire per l’applicazione.

```
FROM eclipse-temurin:21-jdk
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} application.jar
ENTRYPOINT ["java","-jar","/application.jar"]
```

**docker-compose.yml:** definisce i servizi (container) che compongono la tua applicazione, incluse le loro configurazioni, come le immagini da utilizzare, le porte da esporre, le impostazioni di rete e i volumi esterni. Con Docker Compose, puoi gestire tutti i servizi contemporaneamente, facilitando la replica di applicazioni complesse attraverso ambienti diversi

```
services:

 postgres:
    image: postgres:latest
    container_name: postgres
    restart: always
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: secret
      POSTGRES_DB: jdbc_schema
    volumes:
      - pg-data:/var/lib/postgresql/data
    healthcheck:
      test: [ "CMD-SHELL", "pg_isready -U user -d
jdbc_schema" ]
      interval: 30s
      timeout: 10s
      retries: 5

pgadmin:
    image: dpage/pgadmin4
    container_name: pgadmin
    restart: always
    ports:
      - "8888:80"
    environment:
      PGADMIN_DEFAULT_EMAIL: user@domain.com
      PGADMIN_DEFAULT_PASSWORD: password
    volumes:
      - pgadmin-data:/var/lib/pgadmin

  product-service:
    build: product-service-postgres
    image: product-service-postgres
    mem_limit: 512m
    ports:
      - 8080:8080
    environment:
      - SPRING_PROFILES_ACTIVE=docker

volumes:
  pg-data:
  pgadmin-data:
```

## CLI di Docker

La CLI di Docker fornisce vari comandi per gestire immagini e container. Ecco alcuni dei comandi più importanti:

- Scaricare un’immagine da un registro Docker: `docker pull busybox:latest`
- Mostra le immagini locali: `docker images`
- Crea e avvia un container dall’immagine specificata: `docker run -it busybox:latest`
- Elenca i container in esecuzione: `docker ps`
- Avvia e ferma un container: `docker stop <container-id> / docker start <container-id>`
- Elimina un container specificato. Il container deve essere fermato prima di poter essere rimosso: `docker rm <container-id>`

Esegui un comando in un container in esecuzione: Esegue un comando specificato all’interno di un container in esecuzione. I flag -it abilitano una sessione terminale interattiva.

```
docker exec -it <container-id> /bin/echo "Hello World!"
```

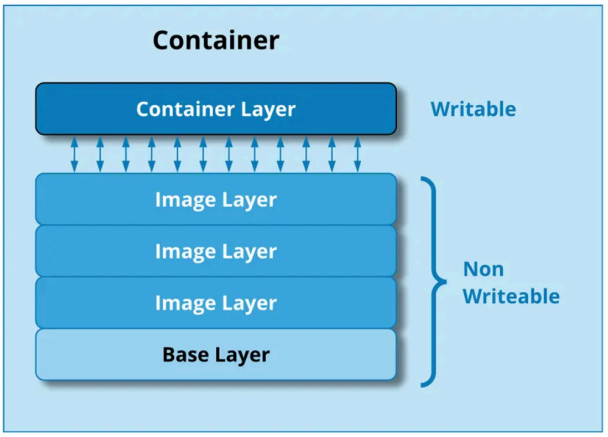
Rimuovi un’immagine dal registro locale: docker rmi -f busybox:latest

# *Creazione delle Immagini*

## Immagini e strati

Le immagini Docker:

- sono istantanee immutabili di un’applicazione e del suo ambiente.
- includono tutto il necessario per eseguire l’applicazione (codice, runtime, librerie, variabili ambientali, file di configurazione e dipendenze).
- sono costruite in un formato a strati.




Ogni strato rappresenta un insieme di modifiche (o un delta del filesystem):

- Strato base: include i componenti minimi necessari per eseguire un’applicazione (ad es., immagini Alpine o Ubuntu).
- Strati intermedi aggiungono modifiche, come l’installazione di pacchetti software, l’impostazione di variabili ambientali o la copia di file. Questi strati corrispondono alle istruzioni nel Dockerfile che descrivono come costruire l’immagine. Ogni riga o istruzione nel Dockerfile (ad es., RUN, COPY, ADD) crea un nuovo strato.
- Solo lettura: Una volta costruiti, tutti gli strati in un’immagine Docker sono in sola lettura. Quando l’immagine viene utilizzata per avviare un container, Docker combina questi strati in sola lettura in una vista unificata utilizzando un File System Unito (come OverlayFS), permettendo loro di agire come un’entità unica.
- Gli strati possono essere ispezionati con strumenti specifici come dive.

Gli strati consentono:

- Costruzioni più rapide: Docker memorizza nella cache gli strati per rendere le costruzioni più rapide. Se un’istruzione del Dockerfile non è cambiata, Docker riutilizza lo strato costruito precedentemente, migliorando le prestazioni di costruzione.
- Riutilizzo degli strati: Poiché gli strati sono indipendenti e riutilizzabili, più immagini possono condividere strati comuni. Ad esempio, se due immagini sono costruite sulla stessa immagine base (ad es. Ubuntu), possono riutilizzare lo strato base, riducendo le esigenze di archiviazione e accelerando il deployment.
- Strato scrivibile: Quando si esegue un container Docker, Docker aggiunge uno strato scrivibile sopra gli strati in sola lettura. Le modifiche apportate al container, come la modifica di file o la scrittura di log, vengono memorizzate in questo strato scrivibile superiore. Tuttavia, una volta che il container è stato eliminato, tutte le modifiche andranno perse.

***Esempio:***

```
FROM ubuntu:20.04
RUN apt-get update && apt-get install -y python3
COPY . /app
CMD ["python3", "/app/myapp.py"]
```

Questo Dockerfile crea i seguenti strati dell’immagine:

1. **Strato Base:** L’immagine ubuntu:20.04 è la base.
2. **Strato Intermedio:** Il risultato dell’esecuzione di apt-get update && apt-get install -y python3.
3. **Strato Intermedio:** Il risultato della copia dei contenuti della directory locale in /app.

4. **Strato Finale:** Lo strato risultante dall’impostazione del comando predefinito (CMD ["python3", "/app/myapp.py"]).

## Comandi Chiave

***FROM***

L’istruzione `FROM` specifica l’immagine base da cui la tua immagine sarà costruita. È di solito la prima riga in qualsiasi Dockerfile.

```
FROM ubuntu:20.04
```

Questo comando imposta l’immagine base su Ubuntu 20.04. L’immagine base fornisce l’ambiente per l’applicazione, ed è importante sceglierne una che includa la maggior parte delle dipendenze necessarie.

***ARG***

L’istruzione `ARG` definisce una variabile che gli utenti possono passare al momento della costruzione dell’immagine Docker con il flag --build-arg. Le variabili ARG sono disponibili solo durante il processo di costruzione e non possono essere accessibili dopo che l’immagine è stata costruita.

Esempio:

```
ARG NODE_VERSION=14
```

Questo comando definisce una variabile di runtime di costruzione NODE_VERSION con un valore predefinito di 14. Puoi sovrascrivere questo valore quando costruisci l’immagine utilizzando --build-arg NODE_VERSION=16.

***COPY***

L’istruzione COPY copia file o directory dalla tua macchina locale nell’immagine Docker.

```
COPY . /app
```

Questo comando copia tutto dalla directory corrente sulla tua macchina locale nella directory /app nell’immagine Docker. È comunemente usato per trasferire codice e risorse applicative.

***RUN***

L’istruzione `RUN` esegue comandi all’interno dell’immagine durante il processo di costruzione, tipicamente utilizzata per installare software o impostare l’ambiente.

```
RUN apt-get update && apt-get install -y python3
```

Questo comando aggiorna l’elenco dei pacchetti e installa Python 3 all’interno dell’immagine. Più comandi possono essere combinati in un’unica istruzione `RUN` per ridurre il numero di strati nell’immagine.

***CMD***

L’istruzione CMD specifica il comando che verrà eseguito quando il container viene avviato. È tipicamente scritto nella forma exec, che è un formato array:

```
CMD ["python3", "app.py"]
```

Questo comando esegue app.py utilizzando Python 3 quando il container si avvia.

NOTA: La principale differenza tra CMD e RUN è il momento dell’esecuzione del comando. RUN viene eseguito durante la fase di costruzione, mentre CMD viene eseguito all’avvio del container.

NOTA: Può esserci solo un’istruzione CMD in un Dockerfile. Se ne elenchi più di una, solo l’ultima avrà effetto.

***CMD-SHELL***

La forma CMD-SHELL di CMD ti permette di specificare un comando in un formato di stringa singolo piuttosto che in un array. Questa forma è utile se hai bisogno di funzioni di shell come la sostituzione delle variabili, il piping o la concatenazione dei comandi.

```
CMD python3 app.py
```

In questo esempio, il comando è interpretato dalla shell, quindi `python3 app.py` viene eseguito all’interno dell’ambiente della shell del container.

***ENTRYPOINT***

L’istruzione **`ENTRYPOINT`** specifica il comando che verrà eseguito come processo principale del container. A differenza dell’istruzione CMD, i comandi ENTRYPOINT non vengono sovrascritti quando il container viene avviato con argomenti aggiuntivi. È usato per garantire che il container esegua sempre il processo specificato.

Esempio:

```
ENTRYPOINT ["python", "app.py"]
```

Questo comando rende python app.py il processo predefinito da eseguire ogni volta che il container si avvia.

***WORKDIR***

L’istruzione WORKDIR imposta la directory di lavoro all’interno dell’immagine. Tutte le istruzioni successive verranno eseguite da questa directory.

```
WORKDIR /app
```

Questo comando imposta la directory di lavoro su /app. Se la directory non esiste, verrà creata automaticamente.

***EXPOSE***

L’istruzione EXPOSE comunica a Docker quale porta utilizzerà la tua applicazione in modo che possa essere accessibile dall’esterno del container.

```
EXPOSE 8080
```

NOTA: EXPOSE non pubblica la porta. Indica semplicemente che l’applicazione ascolterà su quella porta. La porta deve essere effettivamente esposta quando il container viene avviato.

***ENV***

L’istruzione ENV imposta variabili ambientali nel container. Queste variabili possono essere utilizzate da applicazioni o script in esecuzione all’interno del container.

```
ENV ENVIRONMENT=production
```

Questo comando imposta la variabile ENVIRONMENT su production. Le variabili ambientali possono influenzare il comportamento delle applicazioni all’interno del container.

***USER***

L’istruzione USER imposta il nome utente o l’UID che eseguirà i comandi successivi nel container. Viene utilizzata per specificare quale utente i processi all’interno del container eseguiranno.

```
USER appuser
```

## Creazione di un’immagine Docker per una semplice applicazione Python

***Passo 1: Scrivere l’applicazione Python***

```
# app.py
# print("Hello, Docker!")
```

***Passo 2: Creare il Dockerfile***

```
# Usa un runtime Python ufficiale come immagine parent
FROM python:3.9-slim

# Imposta la directory di lavoro nel container
WORKDIR /app

# Copia il contenuto della directory corrente nel container in /app
COPY . /app

# Esegui il comando per eseguire lo script Python
CMD ["python", "app.py"]
```

***Passo 3: Costruire l’immagine Docker***

```
docker buildx build -t python-app:latest .
```

***Passo 4: Eseguire il container Docker***

```
docker run python-app
```

## Creazione di un’immagine Docker per una semplice applicazione Java

***Passo 1: Scrivere l’applicazione Java***

```
// App.java
public class App {
    public static void main(String[] args) {
        System.out.println("Hello, Docker!");
    }
}
```

***Passo 2: Creare il Dockerfile***

```
# Utilizza OpenJDK runtime come immagine parent
FROM eclipse-temurin:21

# Imposta la directory di lavoro nel container
WORKDIR /app

# Copia il contenuto della directory corrente nel container in /app
COPY . /app

# Compila il programma Java
RUN javac App.java

# Esegui il programma Java
CMD ["java", "App"]
```

### ***Passo 3: Costruire l’immagine Docker***

```
docker buildx build -t java-app:latest .
```

### ***Passo 4: Eseguire il container Docker***

```
docker run java-app
```

## Creazione di un’immagine Docker con il comando `ls`

### ***Passo 1: Creare il Dockerfile***

```
# Utilizza un'immagine di Alpine Linux ufficiale come immagine parent
FROM alpine:3.18

# Esegui il comando ls
CMD ["ls", "-al"]
```

### ***Passo 2: Costruire l’immagine Docker***

```
docker buildx build -t ls-command:latest .
```

### ***Passo 3: Eseguire il container Docker***

```
docker run ls-command
# qui sovrascriviamo la direttiva CMD
docker run -it ls-command /bin/sh
```

## Creazione di un’immagine Docker per un server Echo Flask

### ***Passo 1: Creare un Dockerfile***

```
FROM python:3.9-slim
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
CMD ["python", "app.py"]
```

### ***Passo 2: Costruire l’immagine Docker***

```
docker buildx build -t echo-server-flask:latest .
```

### ***Passo 3: Eseguire il container Docker***

```
docker run -p 5000:5000 echo-server-flask
```

Questo comando mappa la porta 5000 sul tuo host alla porta 5000 nel container, permettendoti di accedere all’app Flask.

### ***Passo 4: Testare il server Echo***

```
curl -X POST http://localhost:5000/echo -H "Content-Type: application/json" -d '{"message": "Hello, Echo Server!"}'
```

# Creazione di un’immagine Docker per un server Echo Spring Boot

***Passo 1: Creare un Dockerfile***

```
FROM eclipse-temurin:21
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} application.jar
ENTRYPOINT ["java","-jar","/application.jar"]
```

***Passo 2: Costruire l’immagine Docker***

```
# Maven è necessario per creare effettivamente l'artefatto jar!
mvn clean package -Dmaven.test.skip=true
docker buildx build -t echo-server-java:latest .
```

***Passo 3: Eseguire il container Docker***

```
docker run -p 5000:5000 echo-server-java
```

***Passo 4: Testare il server Echo***

```
curl -X POST http://localhost:5000/echo -H "Content-Type: application/json" -d
'{"message": "Hello, Echo Server!"}'
```

# *Creazione dell’Immagine (Java) da Jar*

## Costruire un’Immagine tramite Dockerfile

L’approccio più standard per convertire i fat jar prodotti da Maven e Spring Boot in immagini Docker è creare un *Dockerfile* che appare come segue:

```
FROM eclipse-temurin:21
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} application.jar
ENTRYPOINT ["java","-jar","/application.jar"]
```

Per prima cosa, crea il tuo artefatto jar all’interno della directory *target/*.

```
$ cd product-service-no-db
$ mvn clean package
```

Poi, crea un’immagine Docker e inviala al tuo demone locale. Il seguente comando dà all’immagine lo stesso nome della cartella che contiene il *Dockerfile*.

```
$ docker buildx build -t $(basename $(pwd)) .
```

Se modifichiamo qualcosa nella nostra applicazione e ricostruiamo l’immagine, saremo in grado di vedere che il processo di costruzione utilizza gli strati memorizzati nella cache, eccetto per lo strato dell’applicazione!

```
$ docker images
REPOSITORY              TAG       IMAGE ID       CREATED          SIZE
product-service-no-db   latest    9883188cab8e   4 seconds ago    478MB
...
```

**NOTA:** L’immagine finale è di 478MB!

Ora eseguiamo la nostra immagine:

```
$ docker run -e SPRING_PROFILES_ACTIVE=docker -p 8080:8080 product-service-no-db
```

Ecco cosa possiamo inferire dal comando:

- **`docker run`:** Il comando `docker run` avvierà il container e mostrerà l’output dei log nel terminale. Il terminale sarà bloccato finché il container è in esecuzione.
- **`-e SPRING_PROFILES_ACTIVE=docker`** imposta la corrispondente variabile ambientale all’interno del container.
- **`-p 8080:8080`** opzione mappa la porta 8080 nel container alla porta 8080 nel host Docker, rendendo possibile chiamarla dall’esterno.

Ora elenchiamo i nostri container in esecuzione:

```
docker ps
CONTAINER ID   IMAGE                  COMMAND                  CREATED         STATUS         PORTS                    NAMES
f5c2c01a895a   product-service-no-db   "java -cp @/app/jib-…"   7 seconds ago   Up 7 seconds   0.0.0.0:8080->8080/tcp   youthful_dijkstra
```

Interroghiamo il nostro nuovo endpoint creato: http://localhost:8080/products

## Costruire un’Immagine tramite Spring Boot Maven Plugin

Spring Boot consente nativamente di costruire la nostra applicazione in un’immagine Docker:

- integrazione con layertools, per aiutarci a ispezionare ed estrarre gli strati nel nostro jar
- integrazione con Cloud Native Buildpacks, che ci aiutano a costruire l’effettiva immagine Open Container Initiative (OCI)

### ***Costruire un’Immagine dalla Linea di Comando***

Aggiungi il plugin di Spring Boot al file di configurazione pom.xml.

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

Crea il tuo artefatto jar all’interno della directory target/.

```
$ mvn clean package
```

Poi, crea un’immagine Docker e inviala al tuo demone Docker locale.

```
$ mvn spring-boot:build-image
```

Dopo che la costruzione è completata, possiamo elencare le nostre immagini Docker nel terminale per verificare che le immagini siano state create come previsto:

```
$ docker images
REPOSITORY                            TAG              IMAGE ID       CREATED         SIZE
paketobuildpacks/run-jammy-base       latest           def0c6bb7994   8 days ago      104MB
paketobuildpacks/builder-jammy-base   latest           72fa30f71c5f   44 years ago    1.43GB
product-service-no-db                 0.0.1-SNAPSHOT   526fa2f45c60   44 years ago    361MB
```

La dimensione dell’immagine è sostanzialmente ridotta rispetto a quella di Docker puro, ma il processo è piuttosto lento.

## Costruire un’Immagine (Jib Maven Plugin)

Jib è uno strumento Java open-source mantenuto da Google per costruire immagini Docker di applicazioni Java. Semplifica la containerizzazione: **non dobbiamo scrivere un Dockerfile.**

Google pubblica Jib sia come plugin Maven che come plugin Gradle. Questo ci fa risparmiare comandi di build/push Docker separati e semplifica l’aggiunta a una pipeline CI.

```
<build>
    <plugins>
        <plugin>
            <groupId>com.google.cloud.tools</groupId>
            <artifactId>jib-maven-plugin</artifactId>
            <version>3.4.3</version>
        </plugin>
    </plugins>
</build>
```

Con questa modifica, possiamo creare immagini con:

```
# (invia al DockerHub)
$ mvn compile jib:build
# (invia al demone Docker locale)
$ mvn compile jib:dockerBuild
```

Possiamo elencare le nostre immagini Docker nel terminale per verificare che le immagini siano state create come previsto:

```
$ docker images
REPOSITORY                          TAG             IMAGE ID       CREATED         SIZE
paketobuildpacks/run-jammy-base     latest          def0c6bb7994   8 days ago      104MB
paketobuildpacks/builder-jammy-base latest          72fa30f71c5f   44 years ago    1.43GB
product-service-no-db               latest          4dd75cdb8aa6   54 years ago    316MB
```

*Jib ha creato un’immagine più piccola in un tempo molto più breve!*

### ***Personalizzare Aspetti di Docker***

Per impostazione predefinita, **Jib fa una serie di ipotesi ragionevoli su ciò che vogliamo**, come il FROM e l’ENTRYPOINT. Facciamo alcune modifiche alla nostra applicazione che siano più specifiche per le nostre esigenze.

```
<build>
    <plugin>
        <groupId>com.google.cloud.tools</groupId>
        <artifactId>jib-maven-plugin</artifactId>
        <version>3.4.3</version>
        <configuration>
            <from>

<image>eclipse-temurin:21-jdk-alpine@sha256:sha256:c63d8669d87e16bcee66c0379d1deedf844152da449ad48f2c8bd7
3a3705d36b</image>
            </from>
            <to>
                <image>product-service-no-db</image>
            </to>
            <container>
                <mainClass>com.nbicocchi.product.App</mainClass>
            </container>
        </configuration>
    </plugin>
</build>
```

Configuriamo tag, volumi e diverse altre direttive Docker allo stesso modo. Jib supporta numerose configurazioni di runtime Java, anche:

- **jvmFlags** è per indicare quali flag di avvio passare alla JVM.
- **mainClass** è per indicare la classe principale, che Jib tenterà di inferire automaticamente per impostazione predefinita.
- **args** è dove specifichiamo gli argomenti del programma passati al metodo main.

## Ottimizzare le immagini Docker

La dimensione dell’immagine può avere un impatto significativo sulle tue prestazioni sia come sviluppatore che come organizzazione. Soprattutto quando si lavora in grandi progetti con molti servizi, questo potrebbe costarti molto denaro e tempo.

- *Spazio:* stai sprecando spazio su disco nel tuo registro Docker e nei tuoi server di produzione.
- *Larghezza di banda:* più grande è l’immagine, maggiore è il consumo di larghezza di banda quando si estrae e si invia l’immagine da e verso il registro.
- *Velocità:* più grande è l’immagine, più lungo è il tempo necessario per costruire e inviare l’immagine.
- Sicurezza: più grande è l’immagine, maggiori sono le dipendenze e maggiore è la superficie di attacco.

### ***Scegliere l’immagine base giusta***

Utilizzando eclipse-temurin:21, l’immagine finale è di 480M

```
FROM eclipse-temurin:21
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} application.jar
ENTRYPOINT ["java","-jar","/application.jar"]
```

Utilizzando openjdk:21-jdk-slim, l’immagine finale è di 470M

```
FROM openjdk:21-jdk-slim
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} application.jar
ENTRYPOINT ["java","-jar","/application.jar"]
```

Utilizzando eclipse-temurin:21-jdk-alpine, l’immagine finale è di 400M

```
FROM eclipse-temurin:21-jdk-alpine
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} application.jar
ENTRYPOINT ["java","-jar","/application.jar"]
```

***Costruisci la tua immagine utilizzando jlink e un Dockerfile multi-stage***

jlink è uno strumento JDK che può essere utilizzato per creare un’immagine di runtime personalizzata che contiene solo i moduli necessari per eseguire la tua applicazione. Possiamo anche utilizzare un processo multi-stage:

1. Il primo stadio viene utilizzato per costruire un’immagine JRE personalizzata utilizzando jlink.
2. Il secondo stadio viene utilizzato per impacchettare l’applicazione in un’immagine slim alpine.

Nel primo stadio, abbiamo utilizzato l’immagine eclipse-temurin:21-jdk-alpine per costruire un’immagine JRE personalizzata utilizzando jlink. Poi eseguiamo jlink per costruire una piccola immagine JRE che contiene tutti i moduli utilizzando --add-modules ALL-MODULE-PATH necessari per eseguire l’applicazione.

```
# Primo stadio, costruisci il JRE personalizzato
FROM eclipse-temurin:21-jdk-alpine AS jre-builder

RUN $JAVA_HOME/bin/jlink \
         --verbose \
         --add-modules ALL-MODULE-PATH \
         --strip-debug \
         --no-man-pages \
         --no-header-files \
         --compress=2 \
         --output /optimized-jdk-21
```

Nel secondo stadio, abbiamo utilizzato l’immagine alpine (che è piuttosto piccola, 3Mb) per impacchettare la nostra applicazione come immagine base, poi abbiamo preso il JRE personalizzato dal primo stadio e lo abbiamo utilizzato come JAVA_HOME. Eseguiamo anche l’applicazione come utente deprivilegiato per una maggiore sicurezza.

```
# Secondo stadio, utilizza il JRE personalizzato e costruisci l'immagine dell'app
FROM alpine:latest
ENV JAVA_HOME=/opt/jdk/jdk-21
ENV PATH="${JAVA_HOME}/bin:${PATH}"
COPY --from=jre-builder /optimized-jdk-21 $JAVA_HOME

ARG APPLICATION_USER=spring
RUN addgroup --system $APPLICATION_USER &&  adduser --system $APPLICATION_USER --ingroup
$APPLICATION_USER
```

```
COPY --chown=$APPLICATION_USER:$APPLICATION_USER target/*.jar /application.jar
USER $APPLICATION_USER

ENTRYPOINT [ "java", "-jar", "/application.jar" ]
```

Utilizzando questo approccio, l’immagine finale è di 170MB.

# *Panoramica di Docker Compose*

## Orchestrazione dei Contenitori

L'orchestrazione dei contenitori è la **gestione automatizzata** delle applicazioni containerizzate su più host. Garantisce che i contenitori siano distribuiti, scalati, connessi e gestiti in modo efficiente in un ambiente di produzione.

- **Distribuzione e Pianificazione Automatizzate** – Assicura che i contenitori siano posizionati sui nodi giusti.
- **Scalabilità** – Regola dinamicamente il numero di contenitori in base alla domanda.
- **Bilanciamento del Carico** – Distribuisce il traffico in modo efficiente tra i contenitori.
- **Auto-guarigione** – Riavvia automaticamente i contenitori guasti o li riprogramma.
- **Scoperta dei Servizi e Networking** – Gestisce la comunicazione tra i contenitori.
- **Sicurezza e Controllo degli Accessi** – Gestisce l'accesso basato sui ruoli e le credenziali.

**Strumenti Popolari per l'Orchestrazione dei Contenitori**

- Kubernetes (K8s)
- Docker Swarm
- Apache Mesos e Marathon
- Red Hat OpenShift

## Il file docker-compose.yml

**Docker Compose è uno strumento potente che ci consente di definire e gestire applicazioni Docker multi-contenitore**. **Docker in esecuzione su un singolo nodo non fornisce funzionalità complete di orchestrazione**. È particolarmente utile quando si lavora con ecosistemi di microservizi, poiché consente il coordinamento di più contenitori. Con Compose, possiamo configurare il networking, le risorse e soddisfare anche i requisiti di scalabilità.

Il file `docker-compose.yaml` segue una struttura gerarchica tramite l'uso di rientranze.

- **services**: Definisce i contenitori dell'applicazione; ogni servizio rappresenta un contenitore.
    - **[nome-servizio]**: Nome del singolo servizio, la scelta è a nostra discrezione.
        - **image**: Specifica l'immagine da utilizzare per il servizio.
        - **build**: Alternativa a `image`, consente di costruire un'immagine da un Dockerfile.
        - **ports**: Mappa le porte dell'host sul container.
        - **volumes**: Consente di condividere dati tra contenitore e host o tra contenitori.
        - **networks**: Definisce le reti che i contenitori utilizzeranno per comunicare.
        - **depends_on: `<servizio>`**: Definisce le dipendenze tra i servizi (quale servizio deve essere avviato per primo).
        - **environment**: Viene utilizzato per passare variabili ambientali per configurare contenitori e applicazioni.
        - **healthcheck**: Garantisce che il servizio sia sano, specificando l'intervallo e il numero di tentativi.

## Utilità da linea di comando

Gestiamo l'ecosistema di contenitori (*servizi*) grazie al comando `docker compose` (o `docker-compose` nelle versioni precedenti).

Prima di tutto, dobbiamo costruire i servizi per essere in grado di creare i contenitori:

```
docker compose build
```

Poi, possiamo avviare l'ecosistema con il comando `docker compose up` nel contesto della directory del progetto.

```
docker compose up --detach
```

**[NOTA]** Utilizziamo il flag `detach` per eseguire i contenitori in background e prevenire output verbosi.

Dato che copieremo sempre il file JAR *ultimo costruito* nel nostro contenitore microservizio, dobbiamo **ricostruire** l'immagine del microservizio **ogni** volta che cambiamo qualcosa nel codice.

Nella pratica, possiamo usare questi due comandi:

```
mvn clean package -Dmaven.test.skip=true
docker compose -f <configurazione-compose>.yml up --build --detach
```

Che è una versione abbreviata di questi:

```
export COMPOSE_FILE=<configurazione-compose>.yaml
mvn clean package -Dmaven.test.skip=true
docker compose up --build --detach
```

## Esempio semplice: esercizio ECHO-SERVER-LOGS-JAVA

Ora distribuiamo una semplice applicazione mappata sulla porta 5000. Di seguito, il file docker-compose.yml.

```
services:
  echo:
    build: .
    ports:
      - "5000:5000"
```

L'applicazione è un semplice server di echo scritto in Java con la capacità di salvare e recuperare log (code/echo-server-logs-java). Il suo controller è riportato qui di seguito:

```
@Log
@RestController
public class EchoController {
    @PostMapping(value = "/echo")
    public Map<String, Object> echo(@RequestBody String message) {
        log.info(message);
        return Map.of("echoed_data", message);
    }

    @GetMapping(value = "/logs")
    public Map<String, Object> logs() throws IOException {
        log.info("requested_logs");
        Path path = FileSystems.getDefault().getPath("/tmp", "application.log");
        return Map.of("lines", Files.readAllLines(path));
    }
}
```

Per ricevere un messaggio di echo:

```
curl -X POST http://localhost:5000/echo -H "Content-Type: application/json" -d '{"message": "Hello, Echo Server!"}'
```

Per vedere i log:

```
curl -X GET http://localhost:5000/logs
```

## Limitazione delle risorse

In Docker Compose possiamo limitare CPU e memoria per i contenitori (code/cpu-memory-meter). Il suo controller è riportato qui di seguito:

```
@RestController
public class MeterController {

    @GetMapping(value = "/")
    public Map<String, Object> echo() {
        long maxMemory = Runtime.getRuntime().maxMemory();
        long totalMemory = Runtime.getRuntime().totalMemory();
        long freeMemory = Runtime.getRuntime().freeMemory();

        return Map.of(
                "Available processors (cores)", Runtime.getRuntime().availableProcessors(),
                "Free memory (MB)", Runtime.getRuntime().freeMemory() / 1_000_000,
                "Maximum memory (MB)", maxMemory == Long.MAX_VALUE ? "no limit" : maxMemory /
1_000_000,
                "Total memory (MB)", Runtime.getRuntime().totalMemory() / 1_000_000,
                "Allocated memory (MB)", (totalMemory - freeMemory) / 1_000_000
        );
    }

    @GetMapping(value = "/allocate/{size}")
    public Map<String, Object> allocate(@PathVariable Integer size) {
        try {
            byte[] array = new byte[size * 1_000_000];
            return Map.of("array allocation", "OK");
        } catch (OutOfMemoryError e) {
            return Map.of("array allocation", "Out of memory");
        }
    }
}
```

Per applicare la limitazione delle risorse a un `servizio` specifico dobbiamo aggiungere l'attributo `resources.limits` sotto `deploy`.

```
services:
  meter:
    build: .
    ports:
      - "8080:8080"
    deploy:
      resources:
        limits:
          memory: 512M    # Limita la memoria a 512MB
          cpus: '2'       # Limita le CPU a 2 core
```

```
unset COMPOSE_FILE
```

```
mvn clean package -Dmaven.test.skip=true
docker compose up --build --detach
curl http://localhost:8080 | jq

{
  "Total memory (MB)": 37,
  "Maximum memory (MB)": 129,
  "Available processors (cores)": 2,
  "Allocated memory (MB)": 21,
  "Free memory (MB)": 16
}
```

Come ci si aspettava, il numero di core (percepito) è 2. La memoria massima (MB) rappresenta la **dimensione massima stimata** della memoria HEAP. La JVM generalmente la imposta tra il 25% e il 50% della memoria totale. In questo caso, 129MB è circa il 25% dei 512MB totali.

Come test finale, puoi provare ad allocare memoria all'interno del servizio. Questo esempio alloca 20MB e funziona bene.

```
curl http://localhost:8080/allocate/20
{"array allocation":"OK"}

Invece, questo esempio alloca 200MB e produce un errore.

curl http://localhost:8080/allocate/200
{"array allocation":"Out of memory"}
```

## Repliche

Una **Replica** in Docker si riferisce alla capacità di istanziare più istanze dello stesso contenitore, consentendoci di scalare orizzontalmente (code/echo-server-logs-java). Il campo `deploy` ci consente di gestire le repliche ed è composto da:

- `mode`: **global** o **replicated**
- `replicas`: il numero di repliche.

**NOTA:** Se non specificato, `mode` = `replicated` e `replicas` = `1`

```
services:
  echo:
    build: .
    ports:
      - "5000"
    deploy:
      mode: replicated
      replicas: 3
```

**NOTA:** Poiché il servizio che stiamo replicando mappa le porte, specificheremo solo la porta del contenitore (omettendo la porta dell'host), altrimenti si verificherà un errore.

Con la stessa procedura vista sopra, avviamo l'ecosistema.

```
export COMPOSE_FILE=docker-compose-replicas.yaml
mvn clean package -Dmaven.test.skip=true
docker compose up --build --detach
```

```
$ docker ps

CONTAINER ID   IMAGE                          COMMAND            …          PORTS
abc123         echo-server-logs-java-echo     "java -jar /applicat" ...    0.0.0.0:32768->5000/tcp
def456         echo-server-logs-java-echo     "java -jar /applicat" ...    0.0.0.0:32769->5000/tcp
ghi789         echo-server-logs-java-echo     "java -jar /applicat" ...    0.0.0.0:32770->5000/tcp
```

Possiamo invocare una replica con:

```
curl -X POST http://localhost:32768/echo -H "Content-Type: application/json" -d '{"message": "Hello, Echo Server!"}'
```

## Volumi Esterni

I contenitori Docker ci consentono di **mantenere dati persistenti che sopravvivono oltre il ciclo di vita del contenitore**, facilitando la condivisione dei dati tra contenitori, backup e ripristino. Se un servizio non specifica una sezione volumi, **nessun dato sarà persistito al di fuori del contenitore**.

Nel seguente esempio (code/echo-server-logs-java) abbiamo un server di echo configurato per salvare i log in '/tmp/application.log'.

```
@Log
@RestController
public class EchoController {
    @PostMapping(value = "/echo")
    public Map<String, Object> echo(@RequestBody String message) {
        log.info(message);
        return Map.of("echoed_data", message);
    }

    @GetMapping(value = "/logs")
    public Map<String, Object> logs() throws IOException {
        log.info("requested_logs");
        Path path = FileSystems.getDefault().getPath("/tmp", "application.log");
        return Map.of("lines", Files.readAllLines(path));
    }
}
```

Possiamo fermare il contenitore, riavviare il contenitore e vedere che i log precedenti sono scomparsi (osserva il timestamp!).

```
$ export COMPOSE_FILE=docker-compose-simple.yaml
$ mvn clean package -Dmaven.test.skip=true
$ docker compose up --build --detach
$ curl -X GET http://localhost:5000/logs | jq

{
  "lines": [
    "25-02-2025 19:07:16.968 [http-nio-5000-exec-1] INFO  c.n.echo.controller.EchoController.logs -
requested_logs",
  ]
}

$ docker compose down
$ docker compose up --detach
$ curl -X GET http://localhost:5000/logs | jq
{
  "lines": [
    "25-02-2025 19:08:33.875 [http-nio-5000-exec-1] INFO  c.n.echo.controller.EchoController.logs -
requested_logs"
```

```
  ]
}
```

Invece di salvare `application.log` all'interno del contenitore, può essere esternalizzata in tre modi differenti.

### ***Volume Nominato***

- **Definizione:** Definito e nominato manualmente dall'utente
- **Persistenza:** Persistente, esiste indipendentemente dal contenitore
- **Accesso dall'Host:** Accessibile tramite comandi Docker
- **Utilizzo Tipico:** Dati persistenti (ad esempio, database)
- **Sicurezza:** Sicuro, gestito da Docker

```
services:
  echo:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - tmp:/tmp → voglio un volume con il nome tmp che si mappa nella cartella /tmp

volumes:
  tmp:
```

- **services**: Definisce i servizi (contenitori) da creare. Qui è definito solo un servizio denominato `echo`.
- **build**: Istruisce Docker a costruire l'immagine utilizzando un Dockerfile situato nella directory corrente.
- **ports**: Mappa una porta sull'host (lato sinistro) a una porta sul contenitore (lato destro). Qui, entrambe le porte sono settate a `5000`.
- **volumes**: Specifica i volumi da montare all'interno del contenitore. Il volume `tmp` è montato su `/tmp` all'interno del contenitore, e potrebbe essere utilizzato per l'archiviazione persistente.
- **section volumes**: Definisce il volume `tmp`, il quale persiste i dati indipendentemente dal ciclo di vita del contenitore.

```
export COMPOSE_FILE=docker-compose-volume.yaml
mvn clean package -Dmaven.test.skip=true
docker compose up --build --detach
```

### ***Volume Anonimo***

- **Definizione:** Creata automaticamente da Docker senza un nome
- **Persistenza:** Temporanea, rimossa con il contenitore
- **Accesso dall'Host:** Non direttamente accessibile
- **Utilizzo Tipico:** Dati temporanei o transitori (file temporanei, log, spazio scrivibile per contenitori in sola lettura)
- **Sicurezza:** Sicuro, gestito da Docker

```
services:
  echo:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - /tmp
```

- **volumes**: Il percorso `/tmp` si riferisce a un volume anonimo. Docker creerà automaticamente un volume senza un nome specifico, e sarà montato nella directory `/tmp` all'interno del contenitore. Questo volume sarà

utilizzato per memorizzare dati ma non avrà un'identità persistente a meno che non gestito esternamente (ossia, non sarà possibile riutilizzare o fare riferimento facilmente a questo volume per nome in seguito).

### ***Montaggio Bind***

- **Definizione:** Collega una directory/file dell'host a un percorso del contenitore
- **Persistenza:** Dipende dal filesystem dell'host
- **Accesso dall'Host:** Accesso diretto dall'host
- **Utilizzo Tipico:** Sincronizza file durante lo sviluppo
- **Sicurezza:** Accesso completo all'host

```
services:
  echo:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - ./data:/tmp
```

- **volumes**: Il percorso `./data:/tmp` specifica un montaggio bind. Questo significa che la directory `./data` sulla macchina dell'host è direttamente montata all'interno del contenitore su `/tmp`. Qualsiasi modifica effettuata nel contenitore su `/tmp` sarà riflessa nella directory `./data` dell'host e viceversa.

```
mkdir data
export COMPOSE_FILE=docker-compose-bind.yaml
mvn clean package -Dmaven.test.skip=true
docker compose up --build --detach
```

## Reti

Un aspetto che rende il motore Docker uno strumento potente è la **possibilità di creare e gestire la connettività dei servizi**. Quando si utilizza Docker Compose, le reti vengono create automaticamente per i tuoi servizi, ma puoi anche definire reti personalizzate per **controllare quali servizi possono comunicare tra loro**.

Per elencare le reti create da Docker, puoi eseguire: `docker network ls`

Per ispezionare una rete specifica e vedere i suoi dettagli, inclusi i contenitori associati, utilizza: `docker network inspect <nome_rete>`

Docker supporta diversi tipi di reti:

- **Bridge**: Tipo di rete predefinito per i contenitori Docker. Isola i contenitori dalla rete host ma consente la comunicazione tra i contenitori sulla stessa rete bridge.
- **Host**: Usa direttamente lo stack di rete dell'host. I contenitori condividono le interfacce di rete dell'host, il che può portare a miglioramenti delle prestazioni ma meno isolamento.
- **Overlay**: Consente ai contenitori di comunicare attraverso diversi host Docker. È comunemente utilizzato in modalità swarm per gestire cluster di contenitori.
- **Macvlan**: Assegna un indirizzo MAC a un contenitore, consentendo che appaia come un dispositivo fisico sulla rete. Utile per applicazioni che richiedono accesso diretto alla rete fisica.

### ***Interazione con iptables***

Docker gestisce il networking utilizzando `iptables`, un'utility Linux per configurare le regole di filtraggio dei pacchetti di rete:

- Quando crei una rete (sia eseguendo `docker-compose up` sia manualmente), Docker aggiunge automaticamente regole `iptables` per consentire il traffico tra contenitori nella stessa rete.

- Ogni rete bridge è assegnata a una subnet, e le regole `iptables` vengono configurate per consentire la comunicazione tra tutti i contenitori su quella subnet.
- Puoi ispezionare le regole `iptables` applicate da Docker utilizzando `sudo iptables -L -n`. Questo comando elenca tutte le regole, incluse quelle create da Docker. Cerca le catene come `DOCKER` che contengono regole specifiche per i contenitori Docker.

### ***Primo esempio***

In questo esempio (code/network-example) definiamo i servizi `echo` e `postgres` connessi a una rete personalizzata chiamata `my_network`, utilizzando un driver `bridge`, che è il predefinito per installazioni su singolo host.

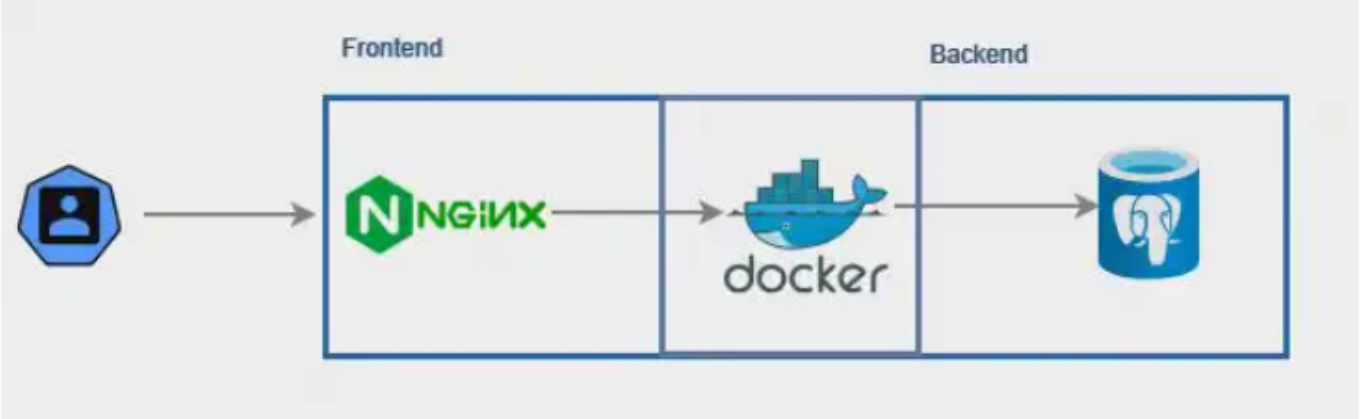
```
services:
  echo:
    build: echo-server-logs-db-java
    ports:
      - "5000:5000"
    environment:
      - SPRING_PROFILES_ACTIVE=docker
    networks:
      - my_network
    depends_on:
      postgres:
        condition: service_healthy

  postgres:
    image: postgres:latest
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: secret
      POSTGRES_DB: jdbc_schema
    volumes:
      - pg-data:/var/lib/postgresql/data
    networks:
      - my_network
    healthcheck:
      test: [ "CMD-SHELL", "pg_isready -U user -d jdbc_schema" ]
      interval: 30s
      timeout: 10s
      retries: 5

networks:
  my_network:
    driver: bridge

volumes:
  pg-data:
```

```
unset COMPOSE_FILE
cd echo-server-logs-db-java
mvn clean package -Dmaven.test.skip=true
cd ..
docker compose up --build --detach
```

### ***Secondo esempio***

Un buon modo per utilizzare le reti è per isolare una parte dell'ecosistema di microservizi che non deve essere esposta a reti esterne. In questo esempio (code/network-example), definiamo due reti `front_net` e `back_net`:

- I servizi `frontend` ed `echo` condividono la `front_net`, consentendo loro di comunicare.
- I servizi `echo` e `database` condividono la `back_net`, isolando il traffico del database dal frontend.
- `nginx` è utilizzato come proxy inverso per inoltrare richieste API.




```yaml
services:
  frontend:
    image: nginx
    volumes:
      - ./frontend/nginx.conf:/etc/nginx/nginx.conf  # Configurazione NGINX personalizzata
    networks:
      - front_net
    ports:
      - "8080:8080"
    depends_on:
      echo:
        condition: service_healthy

  echo:
    build: echo-server-logs-db-java
    environment:
      - SPRING_PROFILES_ACTIVE=docker
    networks:
      - front_net
      - back_net
    depends_on:
      postgres:
        condition: service_healthy
    healthcheck:
      test: ["CMD-SHELL", "curl -f http://localhost:5000/logs"]
      interval: 10s
      timeout: 5s
      retries: 5

  postgres:
    image: postgres:latest
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: secret
      POSTGRES_DB: jdbc_schema
    volumes:
      - pg-data:/var/lib/postgresql/data
    networks:
      - back_net
    healthcheck:
```

```
      test: [ "CMD-SHELL", "pg_isready -U user -d jdbc_schema" ]
      interval: 10s
      timeout: 5s
      retries: 5

networks:
  front_net:
    driver: bridge
  back_net:
    driver: bridge

volumes:
  pg-data:
```

```
export COMPOSE_FILE=docker-compose-isolation.yaml
cd echo-server-logs-db-java
mvn clean package -Dmaven.test.skip=true
cd ..
docker compose up --build --detach
```

## Heartbeat

Le funzioni di heartbeat — comunemente chiamate **health check** — sono meccanismi che verificano periodicamente se un servizio (contenitore) è sano e funzionante come previsto. Implementare funzioni di heartbeat assicura che i componenti della tua applicazione siano attivi e in esecuzione, e consente a Docker Compose di gestire:

- **dipendenze**: un servizio ha bisogno di un altro servizio per funzionare correttamente.
- **restart**: un servizio non funzionante può essere riavviato da Docker.

### *Implementazione dei Health Check Utilizzando `curl`*

**NOTA:** curl deve essere disponibile all'interno del contenitore

```
services:
  webapp:
    image: tuo-immagine-webapp
    ports:
      - "8080:8080"
    healthcheck:
      test: ["CMD-SHELL", "curl -f http://localhost:8080/health"]
      interval: 30s
      timeout: 10s
      retries: 3
```

**Spiegazione:**

- **`test`:** Utilizza `curl` con il flag `-f` per Fallire su errori HTTP. Richiede l'endpoint `/health`.
- **`interval`:** Tempo tra i controlli di salute (30 secondi).
- **`timeout`:** Tempo massimo da attendere per una risposta (10 secondi).
- **`retries`:** Numero di fallimenti consecutivi necessari per contrassegnare il contenitore come `non sano` (3 tentativi).

### *Implementazione dei Health Check Utilizzando `wget`*

**NOTA:** wget deve essere disponibile all'interno del contenitore

In alternativa, puoi utilizzare `wget` per il controllo di salute:

```
services:
  webapp:
    image: tuo-immagine-webapp
    ports:
      - "8080:8080"
    healthcheck:
      test: ["CMD-SHELL", "wget --spider http://localhost:8080/health"]
      interval: 30s
      timeout: 10s
      retries: 3
```

### ***Implementazione dei Health Check Utilizzando Comandi Speciali***

Per servizi come i database, comandi specializzati possono fornire controlli di salute più accurati verificando la prontezza specifica del servizio.

```
services:
  db:
    image: postgres:latest
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: mydb
    healthcheck:
      test: [ "CMD-SHELL", "pg_isready -U user -d mydb" ]
      interval: 30s
      timeout: 10s
      retries: 5
services:
  redis:
    image: redis:6
    healthcheck:
      test: ["CMD-SHELL", "redis-cli ping"]
      interval: 30s
      timeout: 10s
      retries: 3
```

### ***Integrazione dei Health Check con la Direttiva `depends_on`***

La direttiva `depends_on` in Docker Compose specifica le dipendenze dei servizi, assicurando che alcuni servizi si avviino prima di altri. Tuttavia, per impostazione predefinita, `depends_on` aspetta solo che i contenitori dipendenti vengano avviati, non che diventino sani o pronti.

Per far sì che `depends_on` attenda che un servizio diventi sano, Docker Compose supporta dipendenze basate su condizioni. Questa integrazione assicura che un servizio parta solo dopo che le sue dipendenze siano segnalate come sane.

```
services:
  db:
    image: postgres:latest
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: mydb
    healthcheck:
      test: [ "CMD-SHELL", "pg_isready -U user -d mydb" ]
      interval: 10s
      timeout: 5s
```

```
      retries: 5
      start_period: 5s

  webapp:
    image: tuo-immagine-webapp
    ports:
      - "8080:8080"
    depends_on:
      db:
        condition: service_healthy
    healthcheck:
      test: ["CMD-SHELL", "curl -f http://localhost:8080/health"]
      interval: 30s
      timeout: 10s
      retries: 3
```

# DOMANDE E ESERCIZI DEL MODULO CONTAINER:

## *Lab 1: Scrivere un Dockerfile per un Ambiente Java*

**Obiettivo**: Creare un Dockerfile che imposti un'immagine Docker contenente un'applicazione Java.

1. Scrivere un Dockerfile che utilizzi l'immagine di base `eclipse-temurin:21`.
2. Aggiungere una semplice applicazione Java (costruita con Maven come artefatto jar) che stampi "Hello, Docker!".
3. Eseguire l'applicazione Java ogni volta che il container viene avviato.
4. Testare se l'immagine creata funziona correttamente.

## Lab 2: Scrivere un File Docker Compose per un'Applicazione Web Java Spring Boot e un Database

**Obiettivo**: Scrivere un file Docker Compose per definire due container: uno per un'applicazione web Java Spring Boot e l'altro per un database PostgreSQL.

- Creare una nuova applicazione Spring Boot utilizzando Spring Initializr, selezionando le dipendenze appropriate (ad esempio, Spring Web, Spring Data JPA).
- Implementare una semplice API RESTful con operazioni CRUD per gestire una risorsa (ad esempio, User).
- L'applicazione deve girare sulla porta 9000.
- L'applicazione utilizza PostgreSQL per la persistenza (è necessario usare un volume nominato per memorizzare i dati).
- Scrivere un file docker-compose.yml per orchestrare sia l'applicazione che il database.
- Definire la rete tra i due container (sia l'applicazione che il database devono essere mappati sull'host).
- Testare sia il Plugin Maven di Spring Boot che il Plugin Jib per costruire l'immagine senza Dockerfile.

## Lab 3: Limitazione delle Risorse

**Obiettivo**: Impostare limiti di CPU e memoria su un container Java e monitorare il suo utilizzo.

1. Modificare il file `docker-compose.yml` (Lab 2) per eseguire l'applicazione, applicando limiti di CPU e memoria.
2. Modificare il file `docker-compose.yml` (Lab 2) per eseguire PostgreSQL **non** mappato sulla rete dell'host.
3. Avviare i container e utilizzare `docker stats` per monitorare il loro utilizzo delle risorse (e verificare che i limiti siano stati applicati).

### Domande

1. **Quali sono le principali differenze tra il bare-metal, le macchine virtuali e i deployment basati su container? Spiega il concetto di densità di deployment.**
2. **Cos'è un registry Docker e come facilita la distribuzione e la gestione delle immagini dei container?**
3. **Descrivi il ciclo di vita di un container, dalla creazione alla terminazione.**
4. **Commenta un (dato) Dockerfile e spiega le direttive principali.**
5. **Cosa sono Jib e BuildPacks? Perché le immagini più piccole sono preferibili?**
6. **Spiega il ruolo dello storage esternalizzato nei container Docker e in quali modi può essere implementato.**
7. **Puoi definire limiti di CPU e memoria per un container? Perché è utile?**
8. **Commenta un (dato) file `docker-compose.yml` e spiega le direttive principali.**
9. **Cos'è l'orchestrazione dei container e perché è importante nella gestione di applicazioni distribuite?**