

Luca Cabibbo  
Architettura  
dei Sistemi  
Software

# Esecuzione di applicazioni Spring con Docker

**dispensa asw885**  
ottobre 2024

*Composition is frozen improvisation.*  
*Igor Stravinsky*

1

Esecuzione di applicazioni Spring con Docker

Luca Cabibbo ASW



## - Riferimenti

### ❑ Docker

<https://www.docker.com/>

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

### ❑ Docker Compose

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

### ❑ Kickstart Your Spring Boot Application Development

[https://www.docker.com/blog/](https://www.docker.com/blog/kickstart-your-spring-boot-application-development/)

[kickstart-your-spring-boot-application-development/](https://www.docker.com/blog/kickstart-your-spring-boot-application-development/)

2

Esecuzione di applicazioni Spring con Docker

Luca Cabibbo ASW



## - Obiettivi e argomenti

### □ Obiettivi

- mostrare come eseguire con Docker una semplice applicazione Spring Boot
- mostrare come eseguire con Docker un sistema software distribuito composto da più applicazioni Spring Boot, mediante la composizione di container Docker

### □ Argomenti

- introduzione
- un'applicazione contenitorizzata
- esecuzione di applicazioni multi-container
- composizione con Docker
- composizione con Docker Compose
- discussione



## \* Introduzione

### □ I container sono un'opzione di rilascio per i sistemi software distribuiti, sempre più diffusa

- ogni container (“application container”) incapsula un servizio software, insieme allo stack software necessario per quel servizio
- in questa dispensa discutiamo l'esecuzione di applicazioni distribuite Spring Boot mediante container Docker
  - mostriamo prima un esempio introduttivo, di una semplice applicazione web Spring Boot, eseguita con un singolo container
  - poi illustriamo l'esecuzione di un'applicazione composta da più servizi software, eseguita mediante più container



## \* Un'applicazione contenitorizzata

- Illustriamo ora un esempio relativo all'esecuzione di una semplice applicazione web (Spring Boot) in un container Docker
  - l'applicazione **lucky-word** – per il codice e la configurazione si veda la dispensa su Spring Boot
  - non è necessario modificare né il codice né la configurazione



## Un'applicazione contenitorizzata

- Ecco il **Dockerfile** per l'applicazione – lo mettiamo nella cartella principale del progetto Spring Boot
  - utilizziamo l'immagine **eclipse-temurin:21-jdk** con Open JDK

```
# Dockerfile for the lucky-word application
```

```
FROM eclipse-temurin:21-jdk
```

```
# Install the application binary
```

```
ADD build/libs/lucky-word.jar lucky-word.jar
```

```
EXPOSE 8080
```

```
# Launch the Java application
```

```
ENTRYPOINT ["java", "-Xmx128m", "-Xms128m", "-jar", "lucky-word.jar"]
```

- un'alternativa è l'immagine **openjdk:21-jdk**



## Un'applicazione contenitorizzata

- ❑ Ecco come costruire ed eseguire questa applicazione
  - prima di tutto, bisogna effettuare la build (Java) dell'applicazione Spring Boot **lucky-word** (nell'ambiente di sviluppo)

```
gradle build
```

- questo crea, tra l'altro, il file **build/libs/lucky-word.jar** specificato nel **Dockerfile**
  - dopo di che, bisogna costruire un'immagine di container per l'applicazione (nell'ambiente per Docker)

```
# crea l'immagine del container  
docker image build --rm -t lucky-word .
```

- questo crea l'immagine Docker **lucky-word**



## Un'applicazione contenitorizzata

- ❑ Ecco come costruire ed eseguire questa applicazione
  - infine, bisogna creare e avviare il container (sempre nell'ambiente per Docker)

```
# esegue l'applicazione (con il profile di default)  
docker container run -p 8080:8080 --name=lucky-word lucky-word
```

- è possibile avviare l'applicazione con un profilo diverso utilizzando una variabile d'ambiente
    - si può impostare con l'opzione **-e** di **docker run**

```
# esegue l'applicazione con il profilo italiano  
docker container run -p 8080:8080 --name=lucky-word  
-e SPRING_PROFILES_ACTIVE=italian  
lucky-word
```



## - Spring Boot e Docker



- ❑ È utile discutere brevemente le modalità di utilizzo integrato di Spring Boot e Docker
  - un primo approccio è quello che è stato appena mostrato
    - effettuare la build dell'applicazione Spring Boot con Gradle (o Maven) e poi la costruzione dell'immagine con Docker
  - un approccio alternativo – ma non necessariamente migliore
    - effettuare la build dell'applicazione Spring Boot ed anche la costruzione dell'immagine con Gradle (o Maven) – mediante dei plugin
    - il container va poi comunque creato ed avviato con Docker
  - un altro approccio – secondo me peggiore e sconsigliato
    - copiare nel container i file sorgenti ed effettuare la build dell'applicazione con Gradle (o Maven) all'interno del container – con l'opzione **RUN** del **Dockerfile** o, peggio, quando il container viene avviato



## - Errori comuni

- ❑ Attenzione ad evitare i seguenti errori comuni (che prima o poi commettono tutti)
  - dopo aver modificato (il codice sorgente di) un'applicazione, ricordarsi (sempre!) di fare quanto segue
    - effettuare (o ripetere) la build (Java) dell'applicazione
    - effettuare (o ripetere) la build (Docker) dell'immagine Docker
    - effettuare (o ripetere) il push su Docker Hub dell'immagine Docker (se necessario)
    - effettuare (o ripetere) il pull da Docker Hub dell'immagine Docker (se necessario)
    - se non si è cambiato il numero di versione dell'immagine, potrebbe essere necessario anche rimuovere l'immagine Docker precedente dalla cache locale



## \* Esecuzione di applicazioni multi-container

- Un container Docker ha lo scopo di incapsulare ed eseguire una singola istanza di un servizio software individuale
  - tuttavia, un sistema software distribuito è in genere composto da molti servizi software
  - inoltre, per motivi di scalabilità e disponibilità, questi servizi sono spesso anche replicati
  - l'esecuzione di un sistema software distribuito, costituito da più servizi software, può essere effettuata utilizzando molti container – un container per ciascun servizio
  - la definizione e l'esecuzione di applicazioni multi-container – per supportare la realizzazione di sistemi software distribuiti complessi con la tecnologia dei container – è chiamata composizione di container



## Composizione di container

- La *composizione di container* riguarda la definizione e l'esecuzione di sistemi software multi-servizi e multi-container
  - ciascun servizio viene rappresentato da un'immagine di container
  - ogni replica di un servizio viene eseguita come un'istanza di container
  - l'intera applicazione è definita dalla composizione di questi container – tra loro distribuiti
  - le funzionalità di base di Docker Engine esaminate finora consentono di gestire le forme più semplici di composizione
    - le applicazioni multi-container possono però essere gestite in modo più semplice ed efficace mediante strumenti aggiuntivi dell'ecosistema di Docker – in particolare, questa dispensa esamina l'utilizzo di Docker Compose



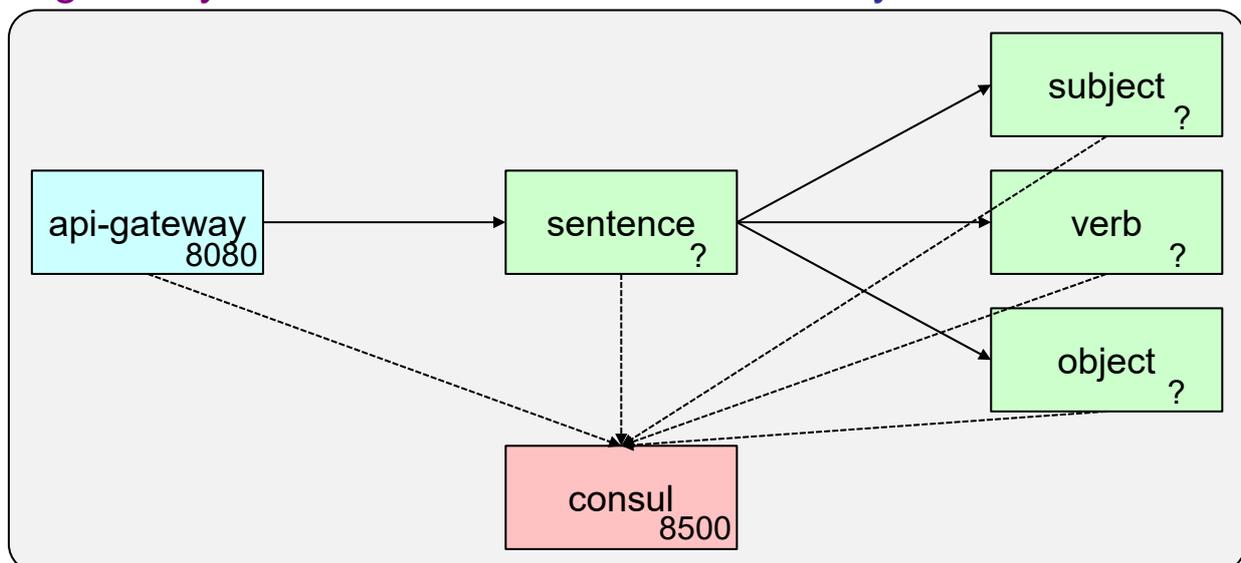
## Sulla “composizione”

- La composizione di container fa riferimento alla nozione (generale) di “composizione” nell’architettura del software – in cui un sistema software è “composto” da più elementi architetturali
  - nella composizione di container, un’applicazione è composta (in modo specifico) da più container
  - la “composizione di container” è però una nozione differente sia dalla “composizione di componenti” che dalla “composizione di servizi” – di cui si parla in altre parti del corso



## Applicazione di esempio

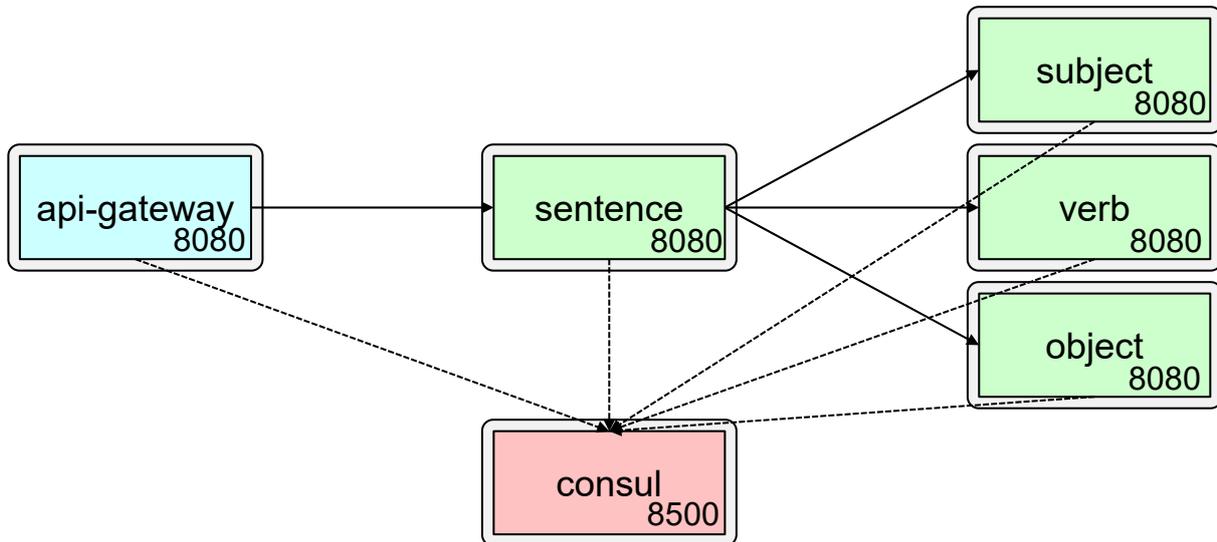
- Come applicazione di esempio, consideriamo di nuovo l’applicazione **sentence** per generare frasi in modo casuale – che è stata introdotta nella dispensa su Spring Cloud
  - consideriamo la versione basata sui servizi **sentence** e **word** (con istanze per **subject**, **verb** e **object**) e un API gateway **api-gateway** – nonché un servizio di discovery **consul**





## \* Composizione con Docker

- Le funzionalità di base di Docker consentono di gestire i casi di composizione più semplici – in cui ci sono diversi tipi di container (servizi), ciascuno con un numero prefissato di istanze
  - vediamo come realizzare l'applicazione **sentence** per generare frasi usando solo le funzionalità di base di Docker



15

Esecuzione di applicazioni Spring con Docker

Luca Cabibbo ASW



## Un'applicazione per frasi casuali

- Per realizzare l'applicazione per generare frasi casuali – con i servizi **sentence**, **subject**, **verb**, **object**, **api-gateway** e **consul** – possiamo utilizzare
  - immagini diverse per i servizi **sentence**, **word** e **api-gateway**
  - uno o più container per il servizio **sentence** – per affidabilità e scalabilità
  - uno o più container per ciascuno dei servizi **subject**, **verb** e **object** – tutti basati sull'immagine per il servizio **word**
  - un container per l'**api-gateway** ed uno **consul** – per quest'ultimo usiamo l'immagine **hashicorp/consul** disponibile su Docker Hub
  - il collegamento in rete tramite una rete definita dall'utente **sentence-net**
    - in particolare, questa rete consente ai diversi container di vedere il servizio **consul** su un nodo di nome **consul**
  - inoltre, la porta 8080 dell'**api-gateway** va esposta sull'host

16

Esecuzione di applicazioni Spring con Docker

Luca Cabibbo ASW



## Modifiche alle applicazioni

- Rispetto a quanto visto nella dispensa su Spring Cloud, bisogna modificare i servizi `sentence`, `subject`, `verb`, `object` e `api-gateway` come segue
  - non è necessario modificare in alcun modo il codice Java dei servizi `sentence`, `word` e `api-gateway`
  - la configurazione dei servizi `sentence`, `word` e `api-gateway` (file `application.yml`) va invece modificata come segue
    - `server.port` può valere 8080 per tutti i servizi – perché ogni servizio viene eseguito in modo isolato in un'istanza di container separata
    - `spring.cloud.consul.host` deve valere `consul` anziché `localhost` – così da puntare a `consul:8500` e non più a `localhost:8500` – la spiegazione è nella slide successiva



## Container e localhost

- `spring.cloud.consul.host` deve valere `consul` anziché `localhost` – così da puntare a `consul:8500` e non più a `localhost:8500`
  - nella dispensa su Spring Cloud, Consul veniva eseguito in un container Docker, e la porta 8500 di questo container veniva pubblicata sulla porta 8500 dell'host
  - le applicazioni Spring venivano eseguite nello stesso host, e per quello potevano vedere Consul all'indirizzo `localhost:8500`
  - quando invece un'applicazione Spring viene eseguita in un container Docker, `localhost` indica il nodo in cui viene eseguita l'applicazione, e quindi il suo container Docker (e non più l'host)
  - Consul viene eseguito in un diverso container, che chiamiamo `consul` – e quindi può essere visto dagli altri container all'indirizzo `consul:8500`
  - si noti che per Consul non serve più pubblicare la porta 8500 sull'host, perché viene ora acceduto mediante la rete di Docker



## Dockerfile

- Per ciascuno dei servizi bisogna definire un **Dockerfile**
  - come esempio, questo è il **Dockerfile** per il servizio **sentence**
  - l'applicazione Spring per il servizio **sentence** viene associata ad una porta nota (del container, non dell'host) – ad es., 8080

# Dockerfile per il servizio sentence

```
FROM eclipse-temurin:21-jdk
```

```
ADD build/libs/sentence.jar sentence.jar
```

```
EXPOSE 8080
```

```
ENTRYPOINT ["java", "-Xmx128m", "-Xms128m", "-jar", "sentence.jar"]
```

```
HEALTHCHECK --start-period=30s --interval=10s  
            CMD curl -f http://localhost:8080/actuator/health || exit 1
```

- i **Dockerfile** per gli altri servizi sono simili



## Immagini dei container

- Questo è lo script di creazione delle immagini Docker per l'applicazione **sentence**

```
#!/bin/bash
```

```
docker build --rm -t sentence-sentence ./sentence-service
```

```
docker build --rm -t sentence-word ./word-service
```

```
docker build --rm -t sentence-apigateway ./api-gateway
```



## Avvio dell'applicazione a container

- ❑ Questo è lo script per la creazione e l'avvio dell'applicazione

```
#!/bin/bash

docker network create sentence-net

docker run -d --network=sentence-net
    --name=consul docker.io/hashicorp/consul

docker run -d --network=sentence-net -e "SPRING_PROFILES_ACTIVE=subject"
    --name=subject sentence-word

docker run -d --network=sentence-net -e "SPRING_PROFILES_ACTIVE=verb"
    --name=verb sentence-word

docker run -d --network=sentence-net -e "SPRING_PROFILES_ACTIVE=object"
    --name=object sentence-word

docker run -d --network=sentence-net --name=sentence-1 sentence-sentence
docker run -d --network=sentence-net --name=sentence-2 sentence-sentence

docker run -d --network=sentence-net -p 8080:8080
    --name=apigateway sentence-apigateway
```



## Monitoraggio dell'applicazione

- ❑ Per conoscere lo stato dell'esecuzione di un'applicazione Spring in un container Docker è possibile usare il comando **docker ps** – in combinazione con l'istruzione **HEALTHCHECK** del Dockerfile
  - se lo stato di un container per un servizio è **health: starting** allora l'applicazione eseguita del container si sta avviando, ma non è ancora pronta
  - se invece lo stato di un container per un servizio è **healthy** allora l'applicazione eseguita del container è stata avviata ed è pronta a ricevere richieste
  - lo stato **unhealthy** indica invece un problema durante l'esecuzione dell'applicazione nel container



# Salvataggio delle immagini su Docker Hub

- Opzionalmente, è possibile salvare le immagini dei container su Docker Hub, come segue
  - effettuare il login su Docker Hub, con il comando `docker login`
    - questa dispensa usa l'utente `aswroma3` – ciascuno deve utilizzare il proprio account e le proprie credenziali
  - taggare le immagini con il nome del proprio account ed effettuare il push – ad es., mediante il seguente script

```
#!/bin/bash
```

```
docker tag sentence-sentence aswroma3/sentence-sentence:2024-10
docker tag sentence-word aswroma3/sentence-word:2024-10
docker tag sentence-apigateway aswroma3/sentence-apigateway:2024-10

docker push aswroma3/sentence-sentence:2024-10
docker push aswroma3/sentence-word:2024-10
docker push aswroma3/sentence-apigateway:2024-10
```



# Avvio dell'applicazione da Docker Hub

- Questo è lo script per avviare l'applicazione da Docker Hub

```
#!/bin/bash
```

```
docker network create sentence-net

docker run -d --network=sentence-net
    --name=consul docker.io/hashicorp/consul

docker run -d --network=sentence-net -e "SPRING_PROFILES_ACTIVE=subject"
    --name=subject aswroma3/sentence-word:2024-10
docker run -d --network=sentence-net -e "SPRING_PROFILES_ACTIVE=verb"
    --name=verb aswroma3/sentence-word:2024-10
docker run -d --network=sentence-net -e "SPRING_PROFILES_ACTIVE=object"
    --name=object aswroma3/sentence-word:2024-10

docker run -d --network=sentence-net
    --name=sentence-1 aswroma3/sentence-sentence:2024-10
docker run -d --network=sentence-net
    --name=sentence-2 aswroma3/sentence-sentence:2024-10

docker run -d --network=sentence-net -p 8080:8080
    --name=apigateway aswroma3/sentence-apigateway:2024-10
```



## Errori comuni

- ❑ Attenzione ad evitare i seguenti errori comuni
  - dopo aver modificato (il codice sorgente di) una delle applicazioni, ricordarsi (sempre!) di fare quanto segue
    - effettuare (o ripetere) la build (Java) delle applicazioni
    - effettuare (o ripetere) la build (Docker) delle immagini Docker
    - effettuare (o ripetere) il push su Docker Hub delle immagini Docker (se necessario)
    - effettuare (o ripetere) il pull da Docker Hub delle immagini Docker (se necessario)
    - se non si è cambiato il numero di versione delle immagini, potrebbe essere necessario anche rimuovere le immagini Docker precedenti dalla cache locale



## \* Composizione con Docker Compose

- ❑ **Docker Compose** è uno strumento per definire ed eseguire applicazioni Docker composte da più container
  - Docker Compose è basato sull'utilizzo di un file di configurazione **docker-compose.yml** per specificare i diversi servizi che compongono un'applicazione
  - inoltre, con Docker Compose è possibile gestire tutti i container di un'intera applicazione mediante dei singoli comandi di tipo **docker compose** (utilizzando Docker Compose v2)
    - i comandi sono analoghi se invece si utilizza Docker Compose v1 – ma sono della forma **docker-compose**



## Servizi

- In Docker Compose, un'applicazione è composta da uno o più elementi chiamati "servizi"
  - un **servizio** (in Docker Compose) corrisponde intuitivamente a un container dedicato all'esecuzione di un servizio applicativo – ad es., un microservizio in un'applicazione a microservizi
  - per ciascun servizio, nel file **docker-compose.yml** vanno specificati
    - l'immagine per il container – il suo contesto Docker e/o il nome della sua immagine di container (su Docker Hub)
    - il comando da eseguire nel container (se diverso da quello di default)
    - ulteriori informazioni e vincoli aggiuntivi



## Esempio

- Ecco la struttura (parziale) delle cartelle per l'applicazione di esempio **sentence** per generare frasi in modo casuale

```
|
+--- docker-compose.yml
+--- api-gateway/
|   +--- Dockerfile
|   +--- build.gradle
|   +--- src/
|       |   +--- ...
|       \--- build/
|           +--- ...
+--- sentence-service/
|   +--- ...
+--- word-service/
|   +--- ...
```

- rispetto a quanto visto nel precedente esempio sulla composizione con Docker, non è necessario modificare in alcun modo i servizi **sentence**, **word** e **api-gateway** – né il codice Java né la loro configurazione né i **Dockerfile**



## Esempio

- ❑ Ecco il file `docker-compose.yml` per l'applicazione di esempio `sentence` per generare frasi in modo casuale

```
name: sentence

services:
  consul:
    image: docker.io/hashicorp/consul

  subject:
    build: ./word-service
    image: aswroma3/sentence-word:2024-10-compose
    environment:
      - SPRING_PROFILES_ACTIVE=subject
    depends_on:
      - "consul"

  verb: ... simile a subject ...
  object: ... simile a subject ...
  ... segue ...
```



## Esempio

- ❑ Ecco il file `docker-compose.yml` per l'applicazione di esempio `sentence` per generare frasi in modo casuale

```
sentence:
  build: ./sentence-service
  image: aswroma3/sentence-sentence:2024-10-compose
  depends_on:
    - "consul"

apigateway:
  build: ./api-gateway
  image: aswroma3/sentence-apigateway:2024-10-compose
  ports:
    - "8080:8080"
  depends_on:
    - "consul"
```



# Composizione con Docker Compose

- Con Docker Compose, è possibile gestire tutti i container di un'intera applicazione mediante singoli comandi **docker compose**
  - le immagini per i container dell'applicazione si possono costruire con il seguente singolo comando
    - **docker compose build**
  - se necessario, le immagini dei container possono essere salvate su Docker Hub con il seguente comando (dopo il **login**)
    - **docker compose push**
  - l'applicazione può essere avviata (creando e avviando tutti i suoi container) con il comando
    - **docker compose up**
  - si possono scalare uno o più servizi con il comando
    - **docker compose up --scale subject=2 --scale object=2**
  - infine, l'applicazione può essere arrestata con il comando
    - **docker compose down**

31

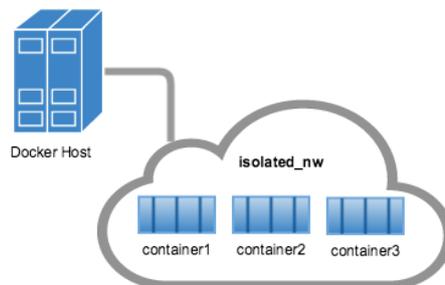
Esecuzione di applicazioni Spring con Docker

Luca Cabibbo ASW



## Ulteriori osservazioni

- Alcune ulteriori osservazioni su Docker Compose
  - Docker Compose definisce automaticamente una rete dedicata all'applicazione – tutti i suoi servizi/container sono collegati a questa rete, e possono comunicare tramite il loro nome



- è anche possibile definire altre reti e topologie di rete complesse
- in questo modo, un host Docker può ospitare più ambienti isolati, ciascuno dedicato a una diversa applicazione
  - è anche possibile avere più copie di una stessa applicazione (in ambienti separati e isolati)

32

Esecuzione di applicazioni Spring con Docker

Luca Cabibbo ASW



## Ulteriori osservazioni

- Alcune ulteriori osservazioni su Docker Compose
  - Docker Compose supporta anche la gestione di volumi
  - l'operazione di build evita di ripetere la creazione di immagini già create e aggiornate
  - l'operazione di push di immagini in un registry è molto importante anche nell'orchestrazione di container (discussa in una successiva dispensa) – in cui è necessario che le immagini Docker di interesse siano accessibili da un registry
  - consente di scalare i servizi di un'applicazione
  - consente anche di definire più configurazioni di una stessa applicazione – mediante meccanismi di estensione dei file di configurazione e l'uso di variabili
  - Docker Compose può essere usato soprattutto per sostenere lo sviluppo e il test di applicazioni multi-servizi – nel computer dello sviluppatore o in un server CI



## \* Discussione

- Abbiamo illustrato l'uso dei container Docker per l'esecuzione di applicazioni Spring Boot
  - abbiamo iniziato mostrando come eseguire una semplice applicazione web, utilizzando un singolo container
  - abbiamo poi discusso la composizione di container, mostrando come eseguire un'applicazione composta da più servizi, mediante più container



## Discussione

- ❑ La composizione di container riguarda la definizione e l'esecuzione di applicazioni multi-servizi e multi-container
  - la composizione di container Docker può essere realizzata in modo semplice ed efficace mediante Docker Compose
- ❑ La composizione di container effettuata in questo modo è utile soprattutto nell'ambiente di sviluppo, e per eseguire dei test di integrazione nell'ambiente di sviluppo
  - per eseguire un'applicazione multi-servizi e multi-container in produzione è invece preferibile utilizzare l'orchestrazione di container (discussa in un successivo capitolo ed esemplificata in una successiva dispensa)
  - in effetti, anche in quest'ultimo caso, Docker Compose è comunque utile, per semplificare alcune attività preliminari all'orchestrazione di container – in particolare, la build e il push delle immagini Docker per l'applicazione di interesse