# Cloud Computing Lab 2 – Docker

# Checklist

*Rispondere alle domande **IN SEQUENZA***

*L'output della domanda X dipende dalle precedenti X-1 domande*

1. **Eseguire il comando** `docker run hello-world` **e riportare l'output ottenuto:**

```
> docker run hello-world

Unable to find image 'hello-world:latest' locally

latest: Pulling from library/hello-world

e58f9ad3b15a: Pull complete

432369badaaa: Pull complete

d1e4ddcdf7f7: Pull complete

Digest: sha256:aa0cc8055b82dc2509bed2e19b275c8f463506616377219d9642221ab53cf9fe

Status: Downloaded newer image for hello-world:latest


Hello from Docker!

This message shows that your installation appears to be working correctly.


To generate this message, Docker took the following steps:

 1. The Docker client contacted the Docker daemon.

 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.

    (windows-amd64, nanoserver-1809)

 3. The Docker daemon created a new container from that image which runs the

    executable that produces the output you are currently reading.

 4. The Docker daemon streamed that output to the Docker client, which sent it

    to your terminal.
```

To try something more ambitious, you can run a Windows Server container with:

PS C:\> docker run -it mcr.microsoft.com/windows/servercore:1809 powershell

Share images, automate workflows, and more with a free Docker ID:

https://hub.docker.com/

For more examples and ideas, visit:

https://docs.docker.com/get-started/

**2. Se si riesegue il comando precedente, la sua esecuzione è più veloce? Se sì, perché?**

Sì, perché docker riesce a trovare l'immagine in locale. Infatti durante la seconda esecuzione non verrà stampata a video la scritta "*Unable to find image 'hello-world:latest' locally*"

**3. Spiegare cosa fa il seguente comando, compresi i flags:**

```
docker run -i -t debian /bin/bash
```

**docker** – lancia docker

**run** - prima crea un nuovo livello scrivibile del container sopra l'immagine specificata e poi manda in esecuzione il container con un'istanza dell'immagine "debian"

**-t** –permette di allocare una sessione virtuale del terminale all'interno del container

**-i** –specifica la modalità interattiva

**/bin/bash** –manda in esecuzione una bash all'interno del container su cui gira l'immagine debian

Docs: https://docs.docker.com/engine/reference/commandline/run/

4. **Dopo aver eseguito il comando precedente, cosa succede eseguendo il comando `exit`? Perché?**

Il comando exit permette di effettuare il logout dalla shell collegata al container creato in precedenza (equivalente a CTRL-D / CTRL-C). Nel caso in questione con il comando exit viene terminato anche il container, perché /bin/bash è l'unico eseguibile in esecuzione sul container.

5. **Eseguire il comando che permette di ottenere la lista delle immagini al momento presenti sulla propria macchina:**

**>docker image ls**

Docs: https://docs.docker.com/engine/reference/commandline/images/

6. **Eseguire il comando che permette di ottenere la lista sia dei container in esecuzione che di quelli eseguiti:**

**>docker ps -a**

Questo comando mostra tutti i container eseguiti ed in esecuzione. (flag -a = all, tutti i container)

Docs: https://docs.docker.com/engine/reference/commandline/ps/

7. **Qual è la differenza in un Dockerfile tra i comandi RUN e CMD?**

**RUN:** viene usato durante la build. Ogni run crea un nuovo layer intermedio. In un Dockerfile possono essere presenti molti comandi RUN.

**CMD:** viene usato per eseguire un comando quando ci lancia un container. In un Dockerfile deve essere presente un solo CMD.

8. **Scaricare da moodle il file DockerApp.zip, estrarlo ed eseguire in locale i seguenti comandi all'interno della cartella per lanciare l'applicazione:**

   a. `pip install -r requirements.txt`

   b. `python3 app.py`

   **Andare quindi su http://127.0.0.1:5000/ e verificare che l'applicazione sia in esecuzione. Dopo averla chiusa containerizzarla ed eseguirla su Docker. Descrivere tutti gli step ed i comandi necessari e gli eventuali flag utilizzati. Verificare che anche l'applicazione containerizzata sia disponibile su http://127.0.0.1:5000/**

   **Suggerimento 1: come immagine di partenza usare** `python:3.8-slim-buster`

   **Suggerimento 2: per lanciare la nostra applicazione in un container è necessario eseguire il comando:** `python3 -m flask run -host=0.0.0.0`

1. Creare il Dockerfile
2. **docker build -t myapp .**
3. **docker run -p 5000:5000 myapp**

Dockerfile:

```
FROM python:3.8-slim-buster

ADD . /app

WORKDIR /app

RUN pip install -r requirements.txt

EXPOSE 5000
```

```
CMD [ "python3", "-m" , "flask", "run", "--host=0.0.0.0"]
```

Il comando build costruisce un'immagine Docker a partire dal Dockerfile e da un contesto, ovvero l'insieme di files specificati nel PATH o URL passato come parametro. Il processo di build può riferirsi ad ogni file presente nel contesto. Il flag -t è diverso da quello che utilizziamo nella run, qui ha il significato di —tag, ovvero taggare, nominare l'immagine che sto andando a creare.

Il flag -p (—publish) serve a "pubblicare" una porta del container all'host, ovvero a renderla visibile ed usabile nei confronti dell'host. In questo la porta dell'host e del container sono la stessa, la numero 5000. A sinistra del carattere ":" è specifica la porta dell'host, a destra quella del container.

**9. Ridigitare il comando del passo 5 e verificare cosa è cambiato**

Rieseguendo il commando `docker image ls` risulta una nuova immagine chiamata myapp

**10. Seguire la seguente guida per provare `docker-compose`:**

`https://docs.docker.com/compose/gettingstarted/`