2. Docker

Cosa è Docker

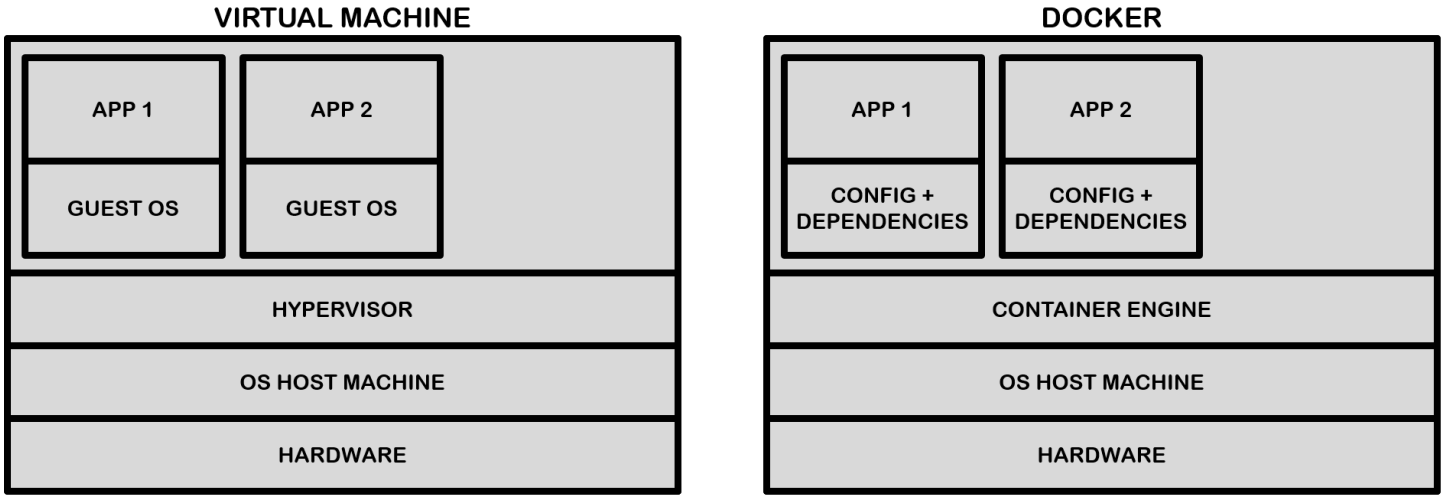
Docker è una piattaforma *open source* che permette di distribuire applicazioni attraverso dei container. Consiste in due servizi di base: **Docker Engine**, la runtime che si occupa della gestione dei container, e il **Docker Hub**, un repository dov'è possibile reperire le immagini dei sistemi operativi dalle quali si andranno a creare successivamente i container.

Gestire applicazioni in container comporta innumerevoli vantaggi, primo fra tutti, la possibilità di eseguire tale applicativo col sistema operativo più compatibile, eliminando qualsiasi conflitto con altri programmi. È possibile creare tanti container quante sono le applicazioni che si intendano utilizzare (detta **architettura a microservizi**), ciascuno con sistemi operativi diversi. Questi inoltre andranno ad occupare molto meno spazio rispetto a una generica macchina virtuale, in quanto di un sistema operativo sarà presente solo il livello superiore al kernel, mentre l'astrazione dell'hardware sarà fornita sempre dal kernel del sistema operativo host.

Per i container si utilizzano tipicamente sistemi operativi basati su kernel Linux, per quanto siano disponibili altre piattaforme. Essi sono distribuiti sotto forma di **immagini** immutabili, la quale modifica può avvenire solo una volta istanziato un container. In pratica:

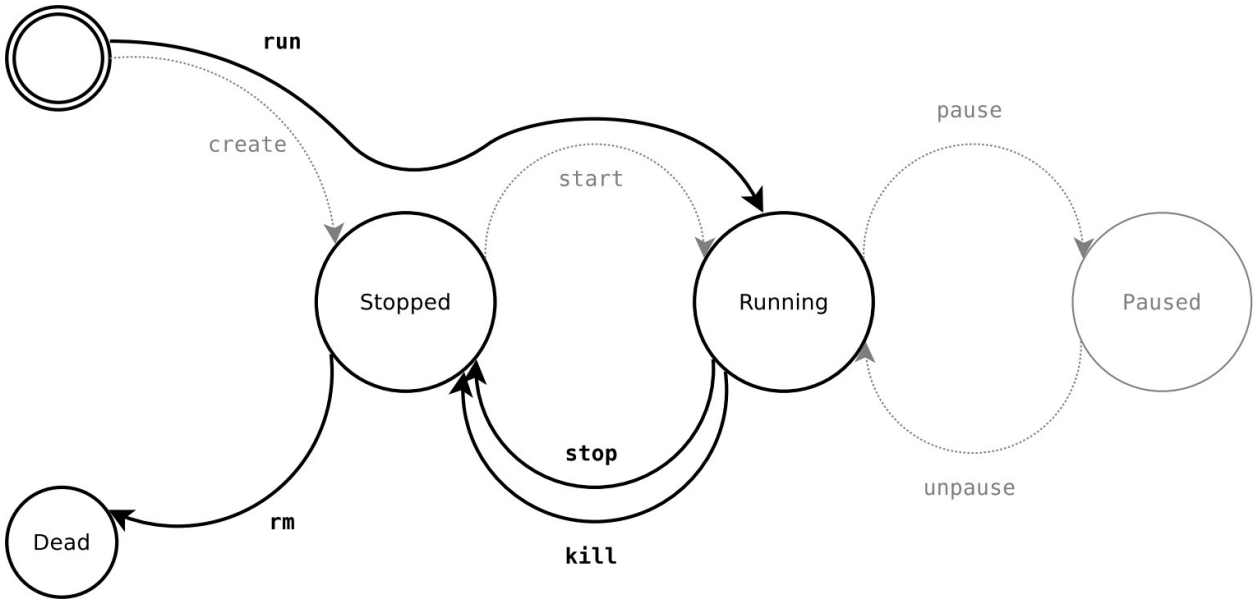
- Un'**immagine Docker** è un'immagine **immutabile** del file system di un sistema operativo, tipicamente Linux. NON può quindi essere modificato;

- Un **container** (struttura - ambiente) è un "file system temporaneo" derivato dall'immagine Docker, sul quale è possibile effettuare delle modifiche. Qualunque modifica viene perduta qualora il container venisse eliminato, a meno che da tale container non si generi una nuova immagine Docker. Il container è costituito inoltre da un ***network stack***, con associato un indirizzo IP, nonché un gruppo di processi, gestito da un ***processo principale*** che termina forzatamente tutti gli altri qualora quest'ultimo venisse terminato.



IVAN SOLLAZZO - ANDREA PISCOPO - NICOLAS BRUNO
PROFESSORE PIERLUIGI GALLO - INGEGNERIA CIBERNETICA
UNIVERSITA' DEGLI STUDI DI PALERMO

Ciclo di vita di un container



Tutti i container in Docker hanno un ciclo di vita dipendente da come lo sviluppatore interagisce con Docker Engine. Un container, nel momento in cui viene creato, può:

- **Essere eseguito direttamente**, evitando di giungere allo stato *Stopped*, e saltare direttamente allo stato *Running*, mediante il comando ***run***;

- **Essere creato, ma non eseguito**, passando subito allo stato *Stopped*, mediante il comando ***create***.

Se il container si trova in *Stopped*, può essere:

- **Eseguito** mediante il comando ***start***;
- **Eliminato** mediante il comando ***rm***.

Nel momento in cui il container si trova in *Running*, può:

- Essere **terminato** mediante il comando ***stop***;
- Essere **ucciso**, che è un'azione di stop forzata, mediante il comando ***kill***;
- Essere messo **in pausa** mediante il comando ***pause***.

Nel momento in cui il container si trova nello stato *Paused*, può essere ripristinato mediante il comando ***unpause***. Qualsiasi container eliminato mediante il comando ***rm*** passa allo stato *Dead***.** Per eliminare un container, è opportuno prima portarlo allo stato *Stopped*.

Il comando ***restart*** fa passare il container allo stato *Stopped* e successivamente allo stato *Running*.

IVAN SOLLAZZO - ANDREA PISCOPO - NICOLAS BRUNO
PROFESSORE PIERLUIGI GALLO - INGEGNERIA CIBERNETICA
UNIVERSITA' DEGLI STUDI DI PALERMO

Comandi gestione dei container Docker

Comando	Descrizione
docker create *immagine*	**Crea un container** partendo da un'immagine Docker, **senza eseguirlo**.
docker run *immagine*	**Crea un container** partendo da un'immagine e **lo esegue**.
docker rename *container nuovo_nome*	**Rinomina** un container.
docker update *container*	**Aggiorna** la configurazione di un container. Alcuni parametri richiedono la ricreazione del container.
docker start *container*	**Esegue** un container.
docker stop *container*	**Termina** il container in modo standard (SIGTERM).
docker kill *container*	**Termina** il container in maniera forzata (SIGKILL, utile quando il container non risponde).
docker restart *container*	**Riavvia** il container.
docker pause *container*	**Mette in pausa** il container.
docker unpause *container*	**Ripristina** il container.
docker rm [opzioni] *container*	**Elimina** il container. Aggiungendo l'opzione *-f*, vengono eliminati anche i container in esecuzione, di fatto è una rimozione forzata.
docker attach *container*	Permette di **collegarsi alla shell testuale** di un container già in esecuzione.

Il file system di un container persiste dunque finchè la macchina a stati si alterna tra *Stopped* e *Running*, mentre viene eliminato quando si passa a *Dead*.

Creazione e avvio diretto di un container

Per avviare direttamente un container, si usa dunque il comando:

```
docker run [opzioni] immagine [argomenti]
```

Il container viene dunque **creato** e passato immediatamente allo stato *Running*. Il file system temporaneo è ereditato dall'immagine specificata. Gli *argomenti* sono i comandi che si intendono passare al container.

Esempio
Il comando *docker run ubuntu /etc/hostname*, crea al volo un container basato sull'immagine *ubuntu* e viene restituito l'output del file */etc/hostname*.

IVAN SOLLAZZO - ANDREA PISCOPO - NICOLAS BRUNO
PROFESSORE PIERLUIGI GALLO - INGEGNERIA CIBERNETICA
UNIVERSITA' DEGLI STUDI DI PALERMO

Di fatto, *docker run* restituisce un codice d'uscita, ovvero, il container viene terminato una volta che i comandi passati vengono terminati. Ciascun argomento può essere separato dal *punto e virgola (;)*.

Foreground mode

L'output restituito in uscita deriva direttamente dai buffer *Stdout* e *Stderr*. Questa modalità è chiamata **Foreground mode**, ed è quella predefinita.

Detached mode

Il container viene avviato senza mostrare l'output di *Stdout* e *Stderr*, al suo posto viene ritornato l'ID del container. Questa modalità può essere invocata aggiungendo l'opzione *-d* al comando *docker run*.

Esempio
docker run -d ubuntu crea al volo un container basato sull'immagine *ubuntu*, senza restituire gli output dei buffer.

Modalità interattiva

È possibile interagire con i container attivando la **modalità interattiva** mediante l'uso delle opzioni *-t* (avvia una TTY) e *-i* (interagisci col container).

```
docker run -ti immagine
```

In tal modo, il container potrà leggere l'*stdin* e quindi sarà possibile inviare comandi alla shell testuale. **Si osservi** che ciò infatti non funziona con la sola opzione *-t*.

Cambio utente

L'utente predefinito è **root**, ma è possibile tuttavia utilizzare un container con un utente diverso mediante il l'opzione *-u*.

```
docker run -u utente immagine [argomenti]
```

Esempio

```
docker run -u provoletta ubuntu whoami
```

Esegue un container derivato dall'immagine ubuntu. Supponendo che al suo interno sia configurato l'utente *provoletta*, viene restituito l'output del comando *whoami* sotto l'azione di tale utente.

Directory di lavoro predefinita

La directory predefinita è **/**, tuttavia è possibile utilizzare un container partendo da una directory di lavoro differente mediante l'opzione *-w*.

```
docker run -w directory immagine [argomenti]
```

Variabili d'ambiente

È possibile configurare delle variabili d'ambiente mediante l'opzione *-e*, può essere ripetuta più volte.

```
docker run -e FOO=foo immagine [argomenti]
```

Hostname

È possibile configurare anche l'hostname del container, mediante l'opzione *-h*.

```
docker run -h hostname immagine [argomenti]
```

Impostazione del nome di un container

Di default, Docker genera un nome a caso per i container creati. Ad ogni modo, mediante l'opzione *--name* è possibile impostare un nome personalizzato.

```
docker run --name nome_container immagine [argomenti]
```

I nomi devono essere **univoci** per ciascun container. NON è possibile avere due container col medesimo nome, anche se hanno ID diversi (da non confondere, appunto, col nome).

Rimozione automatica di un container

Di default, un container Docker rimane memorizzato anche dopo l'esecuzione del comando *docker run*. Aggiungendo l'opzione *--rm*, è possibile eliminare automaticamente esso una volta utilizzato.

```
docker run --rm immagine [argomenti]
```

Rimozione container zombie

Tutti i container inutilizzati possono essere rimossi utilizzando il comando:

```
docker container prune
```

Lista container in esecuzione

Tutti i container in esecuzione possono essere visualizzati in panoramica mediante il comando:

```
docker ps
```

Aggiungendo l'opzione *-a*, è possibile visualizzare anche quelli che NON sono in esecuzione.

```
docker ps -a
```

Creazione di un container senza eseguirlo immediatamente

Un container può essere creato senza eseguirlo immediatamente utilizzando il comando:

```
docker create immagine
```

Può essere avviato successivamente, con tanto di shell testuale, utilizzando il comando:

```
docker start -ia container
```

Dove *-i* sta per ***interactive*** e *-a* per ***attach***.

Interagire con un container già esistente

È possibile avviare una shell testuale di un container già esistente utilizzando il comando.

```
docker start -i container
```

Eseguire comando in un container

Prima di eseguire un comando, è opportuno avviare il container.

```
docker start container
```

Dopodichè, si può utilizzare:

```
docker exec container comando
```

Si osservi che in tal modo non sarà possibile interagire col container, ma una volta terminato quest'ultimo, il container sarà stoppato. A meno che non si usino le opzioni *-t* e *-i*.

Esempio

```
docker exec -it ubuntu bash -c "nano /etc/hostname"
```

Esegue l'editor *nano* nel container Ubuntu. Prima viene invocato il processo *bash* che si occuperà di eseguire una shell dentro la quale avviare *nano* appunto.

Comandi ispezione dei container Docker

Comando	Descrizione
docker ps	Visualizza lista container in esecuzione.
docker ps -a	Visualizza lista di tutti i container, anche di quelli non in esecuzione.
docker logs [opzioni] *container*	Visualizza l'output del container. Aggiungendo l'opzione *-f*, l'output viene seguito in tempo reale.
docker top *container* [opzioni ps]	Visualizza i processi in esecuzione all'interno del container.
docker stats *container*	Visualizza le statistiche di utilizzo di un container.
docker diff *container*	Confronta le variazioni del file system del container rispetto all'immagine da cui esso deriva.
docker port *container*	Visualizza la mappatura delle porte del network stack del container.
docker inspect *container*	Restituisce in output la configurazione a *basso livello* del container, sotto forma di file json.
docker cp *container:percorso percorso_host*	Permette di copiare file dal container al sistema host.
docker cp *percorso_host container:percorso*	Permette di copiare file dal sistema host al container.
docker export *container*	Permette di esportare il contenuto del container sotto forma di archivio *tar*.
docker exec *container* [argomenti]	Permette di eseguire un comando all'interno di un container in esecuzione. Ad esempio, *docker exec -i ubuntu bash* permette di eseguire una shell bash del container ubuntu.
docker wait container	Aspetta che il container termini e ritorna il codice d'uscita. Se viene ritornato 0, significa che non ci sono problemi.
docker commit container	Crea un'immagine del file system del container, utilizzabile per creare altri container.

IVAN SOLLAZZO - ANDREA PISCOPO - NICOLAS BRUNO
PROFESSORE PIERLUIGI GALLO - INGEGNERIA CIBERNETICA
UNIVERSITA' DEGLI STUDI DI PALERMO

Montare volumi con Docker

Il montaggio di un volume su un container Docker può avvenire **SOLO** durante la creazione di un container. Per montare un volume su un container esistente, è opportuno eseguire un **commit** del container, eliminarlo e crearne uno nuovo dall'immagine generata in cui si effettua il montaggio.

Volumi bind

I volumi di tipo **bind** sono dipendenti dalla struttura del file system del sistema operativo host, e può essere messo a disposizione qualsiasi percorso per il montaggio. Il loro ciclo di vita **coincide** con quello del **container**. All'eliminazione di quest'ultimo, i file salvati rimarranno comunque, ma Docker revocherà la specifica di volume fino al prossimo montaggio. Il volume può essere montato con:

```
docker run -ti -v <percorsoHost>:<percorsoContainer> <immagine>
```

- Sia il **percorso dell'host**, che il **percorso del container** devono essere **assoluti**;
- Aggiungendo *:ro* in seguito al percorso del container, il volume viene montato in **sola lettura**;

Lo scopo è quello di rendere i dati persistenti anche in seguito all'eliminazione del container, oppure per rendere possibili comunicazioni interprocesso tra container e sistema host.

Volumi named

Sono volumi gestibili direttamente da Docker e indipendenti dalla struttura del file system del sistema operativo host. Il loro ciclo di vita non coincide con quello del container, ma si gestisce mediante il comando *docker volume*.

Per creare un volume named, si utilizza il comando:

```
docker volume create <nomeVolume>
```

L'output restituito sarà l'identificativo univoco associato al volume. Un elenco dei volumi creati è consultabile mediante il comando:

```
docker volume list
```

Per montare il volume, si utilizza una sintassi simile a quella vista per i volumi bind:

```
docker run -ti -v <nomeVolume>:<percorsoContainer> <immagine>
```

Ancora una volta, i volumi named sono montabili SOLO in fase di creazione del container.

Per eliminare un volume, si usa il comando:

```
docker volume rm <nomeVolume>
```

Montaggio diretto di dispositivi I/O

Montare direttamente dispositivi I/O NON è possibile senza prima inserirli in un'opportuna whitelist, usando il comando:

```
docker --device <percorsoDispositivo> -ti <immagine>
```

In alternativa, è possibile montare device sul sistema host e poi eseguire un montaggio bind sul container.

Gestione della rete

Con Docker è possibile definire delle **reti** per interconnettere le macchine virtuali create con Docker. Di default, a ciascun container è associata la rete **bridge**, costituita tipicamente dalla subnet 172.17.0.0/16, alla quale è possibile associare fino a un massimo di 2^{16} container.

È possibile creare delle reti personalizzate, utilizzando il comando:

```
docker network create <nomeInterfaccia>
```

Di default, muta di una cifra il secondo ottetto. La prima rete personalizzata sarà *172.18.x.x*, ecc. È possibile selezionare in maniera personalizzata la subnet da utilizzare per la rete:

```
docker network create --subnet <IPSubnet>/<mascheraDiRete> <nomeInterfaccia>
```

È possibile scegliere quale rete associare al momento della creazione del container utilizzando il comando:

```
docker run --net <interfaccia> <immagine>
```

Per visualizzare un elenco delle reti è possibile utilizzare il comando:

```
docker network ls
```

Inoltre, è possibile connettere reti a container già esistenti:

```
docker network connect <rete> <contaner>
```

Analogamente, è possibile disconnetterle:

```
docker network disconnect <rete> <container>
```

Tutte le configurazioni di rete vengono mantenute anche dopo aver arrestato e riavviato il container.

Interconnettere più subnet

Tutte le subnet sono isolate tra loro, per principio. Ad ogni modo è possibile interconnettere più subnet collegando un container ad esse. Ci si attenga alla seguente procedura:

1. Creare due reti:

```
$ docker network create pippoNet

$ docker network create plutoNet
```

2. Creare un container per ciascuna rete, aggiungendo l'opzione *--cap-add=NET_ADMIN*. Questa opzione consente di avere i privilegi massimi forniti dal kernel Linux per quanto concerne la gestione della rete.

```
$ docker run -ti --net pippoNet --cap-add=NET_ADMIN --name pippo ubuntu
$ docker run -ti --net plutoNet --cap-add=NET_ADMIN --name pluto ubuntu
```

3. Creare un terzo container da connettere ad ambo le reti:

```
$ docker run -ti --net pippoNet --cap-add=NET_ADMIN --name paperino ubuntu
$ docker network connect plutoNet paperino
```

Assicurarsi che tutti i container siano in esecuzione:

```
$ docker start pippo
$ docker start pluto
$ docker start paperino
```

4. Eseguire su tutti i container il seguente comando:

```
# apt update; apt install iproute2 iputils-ping
```

5. Determinare gli indirizzi IP delle subnet e del container *paperino*:

```
# ip addr
```

6. Aggiungere alla tabella di routing dei container *pippo* e *pluto* il container *paperino* come gateway per le

reti reciproche:

```
# ip route add <subnet pippoNet> via <ip paperino su plutoNet> # container
pluto

# ip route add <subnet plutoNet> via <ip paperino su pippoNet> # container
pippo
```

7. Provare a pingare i container *pippo* e *pluto* a vicenda:

```
# ping <ip pluto> // pippo
# ping <ip pippo> // pluto
```

Port forwarding

È possibile reindirizzare le connessioni di un porta in ascolto sull'host verso una porta locale del container, utilizzando il comando:

```
docker run -d -p [indirizzoIP:]portaHost:portaContainer immagine
```

indirizzoIP è l'IP dell'host, *-d* permette di avviare il container in **detached mode**, utilizzata per non avviare una shell. Il port forwarding viene mantenuto anche dopo il riavvio del container.

Gestione immagini Docker

Un'immagine Docker è appunto uno **snapshot** del file system del container. Utilizza un meccanismo di memorizzazione *copy-on-write* per:

- Istanziare nuovi container;
- Creare nuove versioni di un'immagine.

Ciascuna immagine è identificata da ID univoci, in particolare:

- L'**ID del container**, generato casualmente;
- Il **digest**, un hash creato dal contenuto del file system.

Può inoltre essere aggiunto un nome, detto **tag**, per rendere il container più semplice da riconoscere.

Lista comandi

Comando	Descrizione
docker images	Visualizza lista delle immagini.

docker history *immagine*	Visualizza la cronologia di un'immagine.
docker inspect *immagine*	Visualizza informazioni circa la configurazione del container, in formato *.json*.
docker tag *immagine tag*	Permette di associare un tag a un'immagine.
docker commit *container immagine*	Permette di creare un'immagine da un container.
docker import *percorso*	Permette di importare del contenuto da un file tar per creare un'immagine docker.
docker save *repo[:tag]*	Permette di esportare un'immagine in formato tar.
docker load	Permette di importare uno o più immagini da un file tar.
docker rmi *immagine*	Permette di eliminare un'immagine.
docker pull *immagine*	Permette di scaricare un'immagine dal Docker Hub.
docker search *campo*	Permette di cercare un'immagine nel Docker Hub.
docker login	Permette di collegarsi a un registro dove reperire immagini.
docker logout	Permette di scollegarsi da un registro dove reperire immagini.
docker-ssh	Permette di trasferire immagini tramite SSH.

Osservazione: i comandi *push, pull, search, login* e *logout* permettono di importare ed esportare immagini mediante le **registry API** e dunque in remoto. I comandi *save* e *load* permettono di importare ed esportare immagini manualmente e offline.

Struttura di un Docker tag

Un Docker tag è sostanzialmente strutturato da due parti: il **repository** e il **tag**. In particolare:

- Il **repository** è il nome del container;
- Il **tag** è la versione.

Per esempio, il repository *debian* può avere tag *latest*, per scaricare l'ultima versione dell'immagine o *oldest*, per scaricare la versione più vecchia.

Convenzioni dei Docker tag

Per quanto localmente il tag possa essere arbitrario, i comandi *docker push* e *docker pull* rispettano alcune convenzioni. Il **repository** può essere: Il nome associato nel Docker Hub oppure l'insieme di *hostname + nome*.

Docker builder

Il processo di building consente di costruire un'immagine Docker partendo da:

- Una descrizione **DSL** (Domain Specific Language) che fornisce delle specifiche su come deve essere costruita l'immagine;
- Una cache per memorizzare lo storico delle build dell'immagine, al fine di poter interagire velocemente con esse.

L'input per il builder è tipicamente costituito da:

- Il **Dockerfile** che fornisce delle specifiche su come deve essere costruito il container dal quale si andrà a creare dopo l'immagine;
- Possibili altri file da includere nel processo di building.

Per costruire un'immagine, si usa il comando:

```
docker build [-t tag] percorso
```

I passaggi eseguiti dal comando build sono i seguenti:

1. Viene creato un file *tar* del contenuto incluso col Dockerfile. I file da omettere possono essere specificati in un apposito file chiamato *.dockerignore*;
2. Il daemon di Docker riceverà il *tar* e provvederà a:
    a. Eseguire il **Dockerfile**, effettuando un *commit* **per ogni specifica** indicata nello stesso;
    b. Se specificato come opzione, associare un tag una volta creata l'immagine.

Esempio di un Dockerfile

```
# Con riferimento all'immagine debian:wheezy
FROM debian:wheezy

# Installa gli ultimi aggiornamenti disponibili
RUN apt-get update && apt-get -y dist-upgrade

# Installa il programma nginx
RUN apt-get -y install nginx

# Imposta il comando predefinito del container. Ovvero, avvia nginx in modalità foreground.
CMD ["nginx", "-g", "daemon off;"]

# Indica al Docker engine che tale programma richiede in ascolto la porta 80
EXPOSE 80
```

Sintassi di un Dockerfile

- I **commenti** devono essere preceduti dal *cancelletto* #;
- I **comandi** devono SEMPRE rientrare in una singola riga e possibilmente separati dal *backslash* \;
- Il **primo comando** del Dockerfile deve essere SEMPRE *FROM*, cioè per indicare l'immagine sorgente oppure *scratch* per iniziare da zero.

Elenco comandi

Comando	Descrizione
FROM *immagine/scratch*	Indica se partire da un'immagine sorgente oppure da zero.
COPY *percorso destinazione*	Permette di copiare del contenuto da una sorgente alla destinazione.
ADD *sorgente destinazione*	Analogo a COPY, ma supporta anche gli URL e scompatta gli archivi tar.
RUN *comando*	Permette di avviare un comando arbitrario all'interno del container.
CMD *comando*	Permette di avviare un determinato comando all'avvio del container.
ENTRYPOINT *comando*	*Da confermare*
USER *utente[:gruppo]*	Indica quale utente, arbitrariamente gruppo, deve eseguire il comando.
WORKDIR *percorso*	Permette di indicare la directory di lavoro predefinita.
ENV *nome="variabile"*	Permette di definire una variabile d'ambiente.
STOPSIGNAL *segnale*	Permette di inviare un segnale al fine di terminare il container.
HEALTHCHECK CMD *comando*	Permette di verificare se il container sta funzionando correttamente.
EXPOSE *porta*	Permette di mettere in ascolto una data porta TCP/UDP.
VOLUME *volume*	Permette di impostare un punto di montaggio per un determinato volume esterno.
LABEL *nome="valore"*	Permette di impostare un metadata arbitrario.
ARG *nome[=valore]*	Permette di definire delle variabili a tempo di building.

ONBUILD *istruzione*	Permette di indicare quali istruzioni eseguire **DOPO** il building del Dockerfile. Può essere utile per creare in seguito immagini derivate.

Le variabili a tempo di costruzione possono essere indicate, anziché nel Dockerfile, nel comando *build*:

```
docker build --build-arg nome=valore
```