DOCKER

I docker file sono file di istruzioni che indicano a Docker come deve generare l’immagine di un container.  
Sono file testuali.  
Il giro è il seguente:  
  
docker file -> [docker build] -> docker image -> [docker run] -> startato container docker con immagine che gira  
  
I docker file devono iniziare con una **FROM** instruction (indica l’immagine base da cui partire).  
Es

**FROM** openjdk:8-jdk-alpine   
**MAINTAINER** baeldung.com   
**COPY** target/docker-message-server-1.0.0.jar message-server-1.0.0.jar **ENTRYPOINT** ["java","-jar","/message-server-1.0.0.jar"]

Text

Description automatically generated

NB: Dockerfile deve essere il nome (senza estensione) e poi si crea un .dockerignore (senza estensione) per evitare cosa va messo nell’immagine che non ci serve (es .git etc).

Quando si ha il docker file, si usa **docker build** per creare un immagine docker dal docker file.  
Es  
**docker build -t [TAG\_SCELTO\_IMMAGINE] .**

**NB: il tag in genere è la versione**

(il punto indica il path dove si trova il Dockerfile, e il tag serve per identificare versione dell’immagine)

(una build o la si pusha su un registro online, o si lascia locale, in entrambi i casi la run la riconosce usando i tag.  
Se la lasci in locale, accedendo a docker desktop puoi vedere le immagini buildate presenti nel registro locale, e queste saranno richiamabili per startare i container anche in kubernetes per esempio usando il tag)

A quel punto si puo’ lanciare **docker run** per far startare il container.  
Es:  
  
**docker image ls** -> lista tutte le immagini buildate

**docker run -p[CONTAINER\_PORT]:[TARGET\_PORT\_SPRINGBOOTTOMCAT] [TAG\_SCELTO\_IMMAGINE]**(NB per esempio come arg -p8082:8081 vuol dire che l’applicativo sarà raggiungibile contattando a 8082 nell’url, poiché dobbiamo sempre contattare il container, e questo la gira alla porta applicativa configurata nel properties di spring per il tomcat sull’8081)

**docker ps** -> per visualizzare i container startati con successo  
**docker ps -a -> visualizzazione anche container falliti**  
**docker stop ID\_CONTAINER ->** per stoppare container   
**docker restart ID\_CONTAINER -> restarta**

Nello specifico il giro per applicazioni spring boot, è che si crea il docker file in cui si indica che il jar (precedentemente ottenuto) dell’applicazione deve essere copiato nell’immagine, e che sarà l’entry point (cioè che viene lanciato il .jar quando parte il container).  
In questo modo quando si lancia il container con il docker run parte l’applicativo spring boot.

**NB: nell’esempio scritto prima di Dockerfile, viene copiato tutto il fat jar dell’applicativo (comprensivo di resources) nell’immagine del container. Questo vuol dire che anche se cambiamo le .properties del progetto spring, dobbiamo ribuildare maven (affinchè vengano reinserite nel nuovo jar) e ribuildare l’immagine.**

**NB2: tutte le dipendenze dell’applicativo spring vengono scaricate da maven e copiate nel fat jar che poi viene copiato nell’immagine del container (che conterrà la jdk proveniente da alpine nell’esempio).  
Quindi non c’e’ bisogno di altro per far girare l’applicativo nel container docker.**

Docker compose è un sistema di gestione di container Docker.  
A differenza di Kubernetes non offre autoscaling etc, pero’ comunque permette di fare molte delle cose viste con K8.  
  
Docker Compose, a differenza di docker plain, si usa quando abbiamo piu’ docker da tirare su, ciascuno con le sue immagini e configurazioni.   
Docker compose permette anche di creare una network interna per i docker in maniera tale che questi possano dialogare tra loro usando i nomi logici (dei servizi).

DOCKER COMPOSE

Quando Si usa docker compose, ciascun applicativo deve comunque avere il suo docker file.  
Si puo’ prevedere di lanciare manualmente prima la docker build per creare l’immagine di ciascun applicativo prima di lanciare il docker-compose up, che sfrutta le immagini precedentemente prodotte per gli applicativi, oppure per semplicità si puo’ lanciare   
  
docker-compose -f file\_confdockercompose.yml up (up container conf)  
docker-compose -f file\_confdockercompose.yml down (down dei container della conf)   
docker-compose -f file\_confdockercompose.yml up –build (up container buildando le immagini dei docker linkati)

Docker-compose up –build per far si che vengano generate le immagini docker dei docker file puntati nel docker compose.  
  
Es di **docker** file di progetto  
  
A screen shot of a computer

Description automatically generated  
  
Questo file va chiamato Dockerfile (senza estensione) e messo nella cartella del progetto.  
Poi nella cartella padre (es del repository) si crea **il docker-compose.yml** file  
  
A screen shot of a computer program

Description automatically generated

Spiegazione:  
nei **services** dichiariamo i container docker che docker-compose deve gestire.  
Il nome del service sarà anche il nome logico con il quale i container potranno essere contattati da altri container nella stessa sottorete.  
La sottorete creata è sotto il campo **networks**(in fondo) ed è etl-network.  
Stiamo creando un container maria\_db\_container, usando l’immagine definita (non la trova nel docker hub locale quindi la scarica dal docker hub registry online).

Ci sono 2 modi in cui si danno le variabili d’ambiente al container (quelle poi riferite negli applicativi). Tramite file esterno (**che deve chiamarsi .env per forza, altrimenti non funziona**) linkato con la property **env\_file:** e tramite la property **environment** con cui si listano e si danno direttamente le chiavi / valore. Qui stiamo usando entrambi gli approcci (il file .env che stiamo fornendo al docker-compose è unico per entrambi i docker che stiamo configurando e si trova nella stessa folder dove sta il docker compose).  
  
**NB:** mentre per il primo container, quello del db, puntiamo direttamente all’immagine nel docker registry , per il container applicativo vogliamo che sia docker-compose a mandare la generazione dell’immagine (usando il docker file dell’applicativo ) visto che stiamo usando il flag – build .  
Dobbiamo quindi definire qual è la cartella dell’applicativo e dove si trova il suo docker file.  
Visto che il file docker-compose.yml si trova nella cartella padre (es del repository) al cui interno c’e’ la cartella dell’applicazione etl-importer-updater (al cui interno c’e’ il docker file di etl-importer-updater) queste info le diamo con i campi  
**A screen shot of a computer

Description automatically generated**  
Anche per il container applicativo etl-importer-updater stiamo fornendo l’.env file e stiamo definendo delle variabili anche direttamente nel campo **environment** (**nello specifico stiamo settando anche il profilo spring di start dell’applicativo**  ).

(NB: $SPRING\_PROFILES\_ACTIVE è il nome della variabile da usare quando stiamo startando con docker e vogliamo settare quella che altrimenti avremmo indicato con

java -jar -Dspring.profiles.active=prod (se avessimo lanciato il jar manualmente)  
  
o con

mvn spring-boot:run -Dspring-boot.run.profiles=prod (se avessimo lanciato con mvn spring-boot)

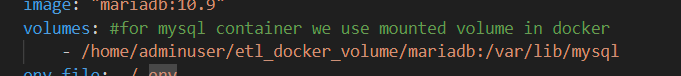
)  
  
Come ultima istanza si noti che alla fine del docker-compose stiamo creando 2 volumes, uno che montiamo (richiamandolo) nel docker del db e uno nel docker applicativo (specificando quando montiamo dove vengono montati)  
  
A questo punto per startare il tutto si fa:  
  
Step1) builda applicativo etl-importer-updater   
 vai dove sta il pom.xml e chiama mvn clean install

Step2) vai dove sta il docker-compose.yml e lancia  
 docker-compose -f docker-compose.yml up –build   
  
Questo equivale ad aver lanciato 2 docker build sui docker file dei 2 container e ad averli runnati con docker run [immagine]

Per resettare completamente docker stack , da shell :  
docker compose -f [NOME FILE CONF DOCKER COMPOSE .yml]

docker rm -f $(docker ps -a -q)

docker volume rm $(docker volume ls -q)

Networks:  
Puoi dichiarare alla fine del docker compose file yml i networks, come bridge e richiamarli nel network di ogni container che dichiari come servizio.  
In questo caso i container potranno parlare tra di loro puntando ai nomi dei servizi come host.  
Questo perché il docker engine crea un bridge network interno.  
ATTENZIONE: in questo caso non puoi usare il localhost per puntare alla macchina host poiché localhost è interpretato da ogni container come il container stesso (non viene usato host file della macchina host su cui gira il docker engine).  
Se vuoi che un docker container usi la network della macchina stessa, devi usare network\_mode:host nel container -> così viene usato il file host della macchina host.  
In questo caso pero’ attenzioen non è possibile usare il port mapping in ciascun container.  
  
Volumi:  
Se dichiari con **volumes** a fine file il nome di un volume, e lo richiami in un container, questo è un volume (persistente anche al riavvio del container) completamente gestito da docker.  
Questi si chiamano **named volumes** e sono quelli che vedi se fai **docker volume ls**.  
I named volumes non esistono nel file system della macchina host.  
Se invece vuoi che docker “veda” una cartella esistente sul file system host allora non dichiari nei **volumes** a fine file, ma fai direttamente in un container ad esempio:  
  
  
Qui stiamo dicendo che nella macchina host a /home/admin/user/etl\_docker\_volume/mariadb c’e’ la cartella che deve finire nel file system del container a /var/lib/mysql

**Questi si chiamano bind volumes**

Questo è il modo in cui si inizializzano i container spesso per i db.  
Ad esempio per mysql quello che finisce in /var/lib/mysql sono i dati del db (tabelle etc).  
Quindi se montiamo un bind volume dal filesystem host nella cartella che l’immagine mysql cerca allo startup per caricare il db, avremo nel filesystem (che non viene cancellato anche se puliamo tutto il docker) i file del db e quindi non lo perdiamo.  
Spesso si fa binding volume di una cartella contenente script di inizializzazione, in un volume del container ad un path specifico perché in quel path il db allo startup cerca gli script di inizializzazione.  
E’ il caso con mongodb ad esempio   
A screen shot of a computer

Description automatically generated  
qui stiamo dicendo di fare un bind mount del file presente in /init\_scripts/ chiamato init\_mongo.sh (il path è relativo al docker compose file yml) per metterlo nel volume del docker di mongodb al path docker-entrypoint-initdb.d che è la cartella che mongodb cerca per startare i file .sh e js presenti in esso (in cui si puo’ fare inizializzazione delle collection user etc).  
Attenzione che per molti db (es mongodb) gli script di inizializzazione vengono lanciati SOLO SE NON ESISTONO GIA’ CARTELLE DI VOLUME ALLOCATE (quindi solo al primo start del docker compose se per i successivi non si rimuovono i volumi) .  
Quindi un alternativa, ad es per mongo db, è non usare script di inizializzazione, ma associare un bind volume (quindi esterno, nel file system dell’host machine) per la sua cartella delle collection, startare il docker compose, collegarsi al mongodb , caricare con client le collection, e a quel punto anche se cancelliamo il docker/container/volumi del docker questo essendo una bind mount non viene cancellata e quindi ai successivi restart del mongodb questo si caricherà le collectiong già presenti nella cartella.   
A screenshot of a computer

Description automatically generated