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 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  
**docker stop ID\_CONTAINER ->** per stoppare container

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.**

Spring Boot da 2.3 evita di dover creare il dockerfile per arrivare ad avere un’immagine startabile su docker della nostra applicazione.  
Questo perché include il buildpack (un applicazione che intelligentemente capisce che tipo di eseguibile si vuole far girare, e crea il container corretto per farlo girare, sostanzialmente quindi un generatore di Dockerfile). In questo modo basta richiedere la build senza passare per il Dockerfile, e il buildpack genera direttamente l’immagine.  
Basta fare **mvn spring-boot:build-image (bisogna avere docker installato e running e l’immagine viene buildata e lasciata in locale)  
  
mvn spring-boot:build-image**

Il nome sarà l’artifact e il tag la versione nel pom

Ma il processo docker file -> docker build -> docker run, è tedioso visto che va fatto per ciascun applicativo / container.  
Per questo esiste **docker-compose che tramite un unico file, docker-compose.yml** permette di configurare differenti container, tutti nello stesso file, e startarli tutti assieme.  
  
**docker-compose up –-build**

**Docker-compose down**

KUBERNETES

A picture containing text, sign, screenshot, vector graphics

Description automatically generated

Kubernetes è un **orchestratore** di container.  
E’ agnostico rispetto alla tecnologia del container (di solito Docker).  
Non necessariamente deve girare nel **cloud**, puo’ girare anche **su macchine fisiche**.  
Kubernetes prevede differenti nodi replica: **nodes**.  
Ogni node **sta su una macchina fisica differente, oppure in una macchina virtuale**.  
In ciascun nodo girano uno o piu’ **POD**.   
Un pod è il **container**.  
Gli stessi pod tipicamente vengono **replicati** **in nodi differenti**.  
I pod **non sono creati direttamente**, ma definendo il **blueprint** , cioè l’immagine (es immagine docker dipendenze) del container. In questo modo il dev ops tramite il blueprint definisce direttamente quante repliche il pod deve avere (e kubernetes distribuirà i pod replicati sui differenti nodes, in maniera tale che se un nodo cade automaticamente aggancia il pod su un altro nodo).  
Una blueprint di una tipologia di POD è detto **DEPLOYMENT.  
In kubernetes non si lavora direttamente con i container, se ne occupa kubernetes.  
Tutto quello che dobbiamo fornire sono le immagini usate (o presenti localmente o su docker hub)**  
L’insieme dei nodi di kubernetes è detto **cluster** kubernetes.  
Un cluster kubernetes ha una sua private network a tutti gli effetti, e ciascun pod ha un suo ip.  
Tuttavia non c’e’ modo di conoscere a priori l’ip che verrà assegnato ad un POD . Inoltre se un pod cade, ed è sostituito da una replica su di un altro nodo, il suo ip interno cambierà. L’ip di un’istanza di pod è quindi **altamente dinamico**  
E’ per questo che i riferimenti ai POD (es applicazioni) si fanno tramite **SERVICES**.  
Un service ha sempre lo stesso IP (**statico)**, ed è assegnato ad una blueprint di pod.  
Così che kubernetes se un POD cade, aggancia un nuovo POD con un nuovo ip al nome del service, ma l’ip del service non cambia.  
Tipicamente ad ogni POD si assegna una sola applicazione (anche se un container docker potrebbe farne girare di piu’ per esempio).  
Le applicazioni tipicamente (es spring boot) si riferiscono ai riferimenti (puntamenti) verso gli altri applicativi nel cluster (gli altri pod)usando le entries delle **ConfigMaps** come fossero variabili di sistema(e infatti in spring finiscono nell’environment).  
Ad esempio se abbiamo un applicativo che deve puntare al DB, l’applicativo spring boot nel suo application.yml o properties avrà per esempio

Hibernate.jpa.connect=${SERVIZIO\_DB}

E nel cluster kubernetes avremo per esempio un service chiamato DB\_SERVICE, che fa riferimento ad un deployment (blueprint POD e insieme di istanze repliche di POD) nel cui container sta il db.  
Inoltre nel ConfigMaps del cluster inseriremo l’entry SERVIZIO\_DB : DB\_SERVICE per far si che la property di environment SERVIZIO\_DB venga iniettata negli applicativi.  
(NB: di solito le password /utenze non vengono salvate in chiaro, ma codificate nell’equivalente delle ConfigMaps chiamato **Secrets**).  
NB: i services quindi rappresentano il dns lookup statico riferito ai POD che cambiano continuamente IP.   
I services di default non sono accessibili da fuori al cluster kubernetes.  
Per abilitare l’accesso ad un service si creano gli **INGRESS**.  
Attenzione: Kubernetes non fornisce supporto alla memorizzazione fisica.  
Le componenti **STATEFUL** (cioè che necessitano di persistere fisicamente informazioni, ad es i DB)vengono si contenuti nei POD, ma il supporto fisico è demandato facendoli puntare a network storage esterni al cluster(ecco perché di solito si evita proprio di creare DB nei POD di kubernetes, ma il db si mette direttamente all’esterno).  
La creazione di blueprint per POD di componenti STATEFUL (es DB) è fatta non con il deployment, ma con lo strumento chiamato **StatefulSets**

Ogni pod ha installato al suo interno 3 componenti , sempre:  
**Kubelet** -> è il runtime di kubernetes ed è quello che si occupa di far salire il POD con la tecnologia di container scelta(es Docker)  
-**Runtime del container** (ES Docker) scelto  
-**Kubernetes Proxy** (proxy interno che fa sì che se un POD cerca di contattare un altro POD, e questo ha una replica che sta girando sullo stesso nodo, evita di far uscire la richiesta dal NODO e la dirotta internamente verso il POD sullo stesso nodo).  
I services in kubernetes si occupano di fare il **load balancing** sui vari POD che replicano la stessa blueprint (dirottando il carico sulle repliche dei POD che si trovano sui nodi meno carichi).  
  
I nodi in kubernetes di dividono in 2 tipi:  
**Master e Worker.**Tipicamente i worker sono > dei master.  
I master hanno sempre 4 componenti:  
**Api Manager** -> le api con le quali il dev op interagisce con il cluster kubernetes (rest api, interfaccia GUI o **kubectl**)  
**Controller Manager** -> demone che si occupa di controllare periodicamente se un pod è giu, e in quel caso lo restarta.  
**Scheduler** -> componente che si occupa di startare nuovi pod, contattando le kubelets che si trovano nelle varie repliche distribuite sui nodi.  
**Etcd** -> è la memoria cache usata dal controller per conoscere lo stato dei nodi, dei pod, cosa è da startare etc.

Esiste una versione di kubernetes che si puo’ configurare in locale e piu’ leggera.  
E’ detta **minikube**.  
Minikube prevede un unico nodo, che è sia master che worker.  
Ha quindi al suo interno la parte del nodo master (api manager, controller manager, etcd e scheduler) e la parte del nodo worker (al cui interno ci sono i pod, ciascuno con la sua kubelet, runtime del container e kubernetes proxy), services, ingress, configmaps, secrets etc…  
NB:minicube viene fatto tipicamente girare in una macchina virtuale.  
(Ironico che si abbia quindi una macchina virtuale, con l’hypervisor, al cui interno sta installato un nodo kubernetes che al suo interno ha dei pod con ad esempio al loro interno docker) .

Per interagire col cluster si possono usare api rest/gui o **kubectl** (va installato separatamente e aggiunto al path, come minikube)  
**minikube start -> starta il cluster** (singolo nodo master e worker)

**kubectl cluster-info** -> info sul cluster  
**minikube dashboard** -> apre la gui di interazione col cluster  
**minikube ssh** -> si collega alla vm del nodo di minikube   
**kubectl get nodes** -> da lo stato dei nodes  
**kubectl get pod** -> da i pods   
**kubectl get services** -> da i services  
**kubectl get deployments** -> ti da i deployments dichiarati  
**kubectl get replicaset ->** da la lista dei replicaset creati

**kubectl create -h ->** :help per il create command  
NB: non esiste il create pod, poiché non si lavora direttamente con i pods, ma con l’astraction layer del **deployment.**Il **deployment** rappresenta la blueprint per una o piu’ istanza di pod. Ad un deployment è associato un **replicaset** , che indica il numero di istanze replicate del pod rappresentato dal deployment.  
**kubectl create deployment NOME\_DEPL –-image=IMAGE\_NAME [options]->** serve per creare da riga di comando un deployment per un set di repliche di pod, direttamente da riga di comando.  
IMAGE\_NAME deve essere una delle immagini pubblicate su docker hub.  
Quando viene creato un deployment ( aggiornato uno esistente ) kubernetes automaticamente lo starta o aggiorna quello startato.  
(NB: in alternativa alla kubectl create deployment si puo’ usare il  
**kubectl run** )  
Usando l’istruzione sopra, automaticamente tutte le altre impostazioni del pod vengono date in default.  
**L’id dei pod generati al momento della creazione del deployment di riferimento sono ottenuti concatenando nome del deployment-id replicaset – id pod**  
Anche col replicaset non si lavora direttamente, ma sempre con il deployment che rappresenta astrazione verso replicaset e istanze dei pod.  
**Deployment -> Replicaset -> POD INSTANCES (distribuite su piu’ nodi)  
  
kubectl edit deployment NOME\_DEPLOYMENT ->**serve per visualizzare il file di configurazione del deployment (che se creato col comando di sopra contiene gran parte di cose autogenerate di default)  
Il file di configurazione di un deployment se editato automaticamente kubernetes stoppa tutti i pod associati, crea un nuovo replicaset con nuove istanze di pod, e rilancia questi ultimi.  
  
**kubectl logs POD\_NAME ->** log sul pod  
**kubectl describe POD\_NAME ->** descrizione di cosa e’ avvenuto nel pod come sequenza di stati  
**kubectl exec -it POD\_NAME – bin/bash -> si collega in ssh all’istanza del pod**

Visto che creare un deployment specificando da riga di comando tutte le varie opzioni è scomodo, è possibile salvare tutto su un file e fare

**kubectl apply -f FILE\_NAME.yaml**  
NB : una volta lanciato l’apply, se viene modificato il file di configurazione, viene fatto update dei pods e dei replicaset , proprio come se avessimo lanciato il kubectl edit.

Ma come sono fatti questi file di configurazione ?  
Hanno 4 sezioni:  
type,metadata, specs e status.  
  
In type è specificato il tipo di file (per cosa, cioè deployment, service etc…), in meta i metadati cioè ad esempio il nome del deployment o del service, e nello spec le specifiche (es nel caso di deployments c’e’ l’immagine docker del container dei pod associati).  
Lo status lo genera kubernetes con l’effettivo runtime status rispetto alle specification (salvato nell’etcd)

….

Tuttavia dato l’elevato grado di complessità della gestione kubernetes, esistono suite per le aziende che forniscono kubernetes as service, es RedHat OpenShift, Portainer etc…