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  
**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 -Dspring-boot.build-image.imageName=[IMG\_NAME]**

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**

**(NB: col docker compose non si puo’ usare il BuildPack, poiché il docker compose fa riferimento ai DockerFile dei vari applicativi, separati, mentre il BuildPack non passa per un docker file ma dal pom genera direttamente l’immagine).**

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.  
NB: i services quindi rappresentano il dns lookup statico riferito ai POD che cambiano continuamente IP.

**NB: ci sono vari tipi di services in kubernetes.  
-ClusterIP (quello di default se non si specifica il type quando si dichiara il service) -> solo interno  
-LoadBalancer -> accessibile dall’esterno (attenzione il nome fa confusione perché in realtà tutti i service kubernetes fanno load balancing) e dall’interno (ha anche lui un ip interno). Ha una nodePort associata, che è dove risponde all’esterno (nel range tra 30000 e 32000).  
-NodePort (attenzione non confondere con la nodePort) -> anch’esso accessibile all’esterno, ha anche essa una nodePort associata nel range 30000-32000.  
-Ingresses  
-…**  
  
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 ciascuno delle **variabili di ambiente definite nel file di deployment, che vengono create all’interno di quel container,** e queste variabili di ambiente sono riempite usando delle entriescentralizzate nelle **ConfigMaps** come fossero variabili di sistema  
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}  
SERVIZIO\_DB è una variabile di ambiente (env variable a tutti gli effetti, quindi riferibile anche in maven) del container.  
Questa variabile di ambiente è definita nel deployment.yaml del deployment del pod, e quando la definiamo indichiamo anche il valore che deve avere (non è una buona pratica dare un valore definito) oppure da dove deve andarsela a prendere: se da un config map (indicando il nome e la chiave) o da un secret (i secret sono come le config map ma tipicamente si salvano i valori in base64 di username e pw e altri dati sensibili)   
  
  
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  
In alternativa si usano i VOLUMES.**

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 (detto Kubernetes Control Plane) 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.

Kubernetes permette , tramite l’uso dei **namespaces** di avere piu’ cluster logici su di un unico cluster fisico.  
I **volumes** servono per permettere l’accesso dei container allo spazio di memorizzazione fisico (visto che non lo hanno realmente, es filesystem del nodo nel cluster)-  
I **DemonSet** servono per prevedere che ogni nodo del cluster abbia almeno un’istanza che gira di un pod specifico (poiché con i deployment non è garantito come vengano distributi i pods sui nodes del cluster).  
  
  
  
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 su di un **driver** (cioè ho una macchina virtuale **VirtualBox**, o su di un docker engine).   
Quindi per installare Minikube bisogna prima installare VirtualBOx.

**MACCHINA HOST (es Windows), su questa gira:  
 🡪VM (es VirtualBox), su questa gira:  
 🡪 MINIKUBE , su questo gira:  
 🡪 GIRA IL POD**

ATTENZIONE: installare docker desktop e minikube che usa come driver virtual box porta ad un problema per cui docker parte ma minikube no.  
Questo perché docker desktop per funzionare richiede **Hyper-V**, mentre **VirtualBox** , la macchina virtuale su cui far girare minikube, vuole Hyper-V disabilitato.  
Una prima soluzione è usare come virtualizzazione per il nodo minikube il docker precedentemente installato (e non installare VirtualBox)  
Un altro è usare sì VirtualBox come VM per minikube, ma usare il supporto del docker desktop per il kubernetes embedded (nelle impostazioni) e startare il minikube invece che con minikube start con

**minikube start --driver=virtualbox --no-vtx-check**

(se si blocca, fare **minikube delete**, aprire VirtualBox e cancellare le macchine virtuali, e rilanciare il comando di sopra)

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) NB: NON VA FATTO SE STIAMO USANDO IL KUBERNETES EMBEDDED DI DOCKER DESKTOP, PERCHE’ QUESTO STARTA DA APPLICATIVO

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

**minikube image ls -> per vedefre immagini registro docker del minikube**

**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.

NB: i deployment e i pod possono essere divisi in namespaces,

**kubectl create namespace casdemo**

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)

*NB: in realtà in un unico .yml si possono avere piu’ definizioni di cose differenti (es un deployment e un service, entrambi relativi alla stessa applicazione etc)*

Abbiamo visto quindi come buildare un’immagine (partendo da un Dockerfile, o bypassandolo usando il BuildPack).  
In questo modo produciamo delle immagini che lasciamo (a meno che non pushiamo esplicitamente) nel nostro docker registry locale delle immagini (visualizzabili con docker images o via app).  
Abbiamo anche visto come creare deployment in minikube, indicando le immagini.  
Un problema è che il nostro minikube gira nella VM, e quindi il suo FS non ha accesso al docker registry che è locale al nostro fs nativo (es windows, su cui è installato docker desktop) su cui ci sono le immagini degli applicativi che abbiamo buildato.  
Ci sono varie alternative per risolvere questo:  
1)usare invece di docker build, **minikube build**, e questo fa sì che l’immagine sia salvata nel registro docker del fs della vm su cui sta girando minikube (con i suoi runtime docker separati).  
In questo caso non possiamo usare il BuildPack.  
2)~~buildare il docker file nell’immagine (o usare il BuildPack di mvn per aggirare la cosa) e salvare l’immagine nel docker registry locale, a quel punto usare il comando~~

**~~minikube image load {IMG\_NAME}~~**

~~e questo farò l’upload dell’immagine sul minikube.~~  
3)Oppure si puo’ lanciare   
**minikube docker-env  
e** lanciare l’ultimo comando   
(in windows **CMD**)  
**@FOR /f "tokens=\*" %i IN ('minikube -p minikube docker-env --shell cmd') DO @%i**

(in windows POWERSHELL)  
**& minikube -p minikube docker-env --shell powershell | Invoke-Expression**  
questo fa si che NELLA SOLA CONSOLE in cui è stato lanciato, i comandi docker (es build) vengano interpretati dal docker del minikube e non da quello locale sulla macchina host.

Attenzione pero’\_:

Tip 1: Remember to turn off the imagePullPolicy:Always (use imagePullPolicy:IfNotPresent or imagePullPolicy:Never) in your yaml file. Otherwise Kubernetes won’t use your locally build image and it will pull from the network.  
  
  
per vedere le immagini docker visibili al minikube nella VM si fa:   
**minikube image ls**

**Attenzione kubernetes permette di usare i namespace.**Se non si definisce nulla il namespace è default.  
Tutti i comandi vengono lanciati nel namespace default.  
Ciascun namespace ha le sue risorse che vi appartengono.  
Solo i servizi sono visibili tra namespace differenti.  
Le config map e i secrets no.

**Kubectl get namespace ->** ritorna i ns  
**kubectl create namespace [NOME] ->** crea un namespace (o in alternativa si puo’ usare l’apply di uno .yaml che definisce un namespace)  
**kubectl get [RISORSA] -n=[NOME\_NAMESPACE]**  si fa la query per quello specifico namespace  
es:  
 **kubectl get deployment -n=mionamespace** (ritorna deployments del solo namespace mionamespace)  
**kubectl apply -f deployment.yml –namespace=mionamespace ->** per creare una risorsa nello specifico namespace (in alternativa si puo’ fare il semplice apply del deployment, e il namespace di appartenenza, che deve già esistere, è definito direttamente nel .yml)  
  
**Quando in un deployment.yml si vuole fare riferimento ad un componente in un dato namespace, bisogna mettere .nome\_del\_ns dopo il nome del componente.**  
  
In alternativa si puo’ installare **kubectx** per evitare di dover ogni volta specificare il namespace e usare il comando **kubens** per switchare direttamente tra i ns.

Per specificare un ns quando si crea una risorsa (es con l’apply) si fa –n  
  
Se si vuole evitare di dover specificare un namespace per ogni comando, si puo’ installare kubectx e lanciare il comando   
  
**NB: esiste Helm package Manager che evita di dover creare uno a uno gli yml per ciascun deployment /immagine, ma fornisce delle collezioni di yaml a seconda degli stack che si vuole usare (tali collezioni sono dette Helm Charts) e permette anche di scrivere script per l’automatizzazione di tali dipendenze.**

Esempio di flow spring boot.:

**Prerequisiti:**Installa VirtualBox.  
Installa Minikube (aggiungi path).  
Installa Docker desktop (aggiungi path).  
Abilita supporto per kubernetes embedded del docker desktop (per evitare problema conflitto hyper-V on off).

Vogliamo creare un applicazione spring boot in minikube che gira su un pod.  
Questa deve essere accessibile dall’esterno .  
L’applicazione spring boot nel suo pod, deve dialogare con un db.  
Il db sta su di un altro pod.  
Il db non deve essere accessibile all’esterno (TRANNE CHE PER MOTIVI DI DEBUG, QUERY) quindi avrà un service interno semplice associato. Per fare query di test e prova, accederemo al db nel pod nel cluster usando CompassMongoDB installato sulla macchina host , tramite port forwarding verso il pod.  
Il pod dell’applicativo sa quali sono gli ip del service del db grazie ad una configmap.  
Username e pw del db sono storati in un secret.

**STEP0: STARTARE DOCKER DESKTOP, E STARTARE MINIKUBE   
minikube start –driver=virtualbox --no-vtx-check**

Possiamo controllare l’ip a cui risponde il cluster facendo **kubectl cluster-info** e ottenere la lista di tutti gli elementi dell’unico nodo disponibile facendo **kubectl get all -o wide**

**STEP1:** **INIZIAMO DAL DB (MONGODB)**creare un deployment yml che rappresenta il file di configurazione per il deployment dei pod/del pod del database. L’immagine usata è una presente online sul DockerHub online. Text

Description automatically generated

Come si legge da questo che rappresenta il deployment del pod su cui girera’ mongodb, stiamo definendo un deployment per immagine 4.4.6 di mongodb, e il container deve aprire la porta 27017 (che è la porta su cui di default risponde mongodb).  
Inoltre ciascun tipo di immagine, quando viene configurato il container che fa partire l’applicazione, necessita di una serie di variabili di ambiente che devono essere configurate.  
*(Per vedere come funziona e che porte espone etc si va su docker hub.com e si cerca il nome dell’immagine)*Queste vengono definite nello stessa specifica del container (nel file di deployment) e il nome che la variabile avrà nell’ambiente del container è data dal campo name es MONGO\_INITDB\_ROOT\_USERNAME.  
Potremmo dare un valore direttamente, ma è tipicamente meglio centralizzarle e mettere questi valori nei config map e nei secrets.  
Sostanzialmente in questo esempio l’immagine di mongodb ha bisogno che il container abbia settate 2 variabili d’ambiente (chiamate rispettivamente MONGO\_INITDB\_ROOT\_USERNAME e \_PASSWORD) e questi valori vengono referenziati precendoli dal secret.  
  
  
Prima di proseguire si noti che in questo stesso file di deployment è dichiarato anche il service (senza specificare il tipo, quindi di default sarà un ClusterIP, quindi non accessibile all’esterno, che risponde alla porta 27017 e rigira sulla 27017).  
Tipicamente si dichiara in un file di deployment anche il service associato (le dichiarazioni vanno separate con --- che in yaml è considerata separazione di documenti, altrimenti non funziona).  
  
Quindi innanzitutto prima di poter fare l’apply di questo deployment, dobbiamo fare l’apply del file del secret, in quanto è referenziato dal deployment del database.  
Il secret avrà la forma seguente **(SI NOTI CHE I VALORI DEVONO ESSERE BASE64 ENCODED)**

Text

Description automatically generated

Quindi definiamo il secret facendo  
**kubectl apply -f mongo-secret.yaml**Questo farà si che venga dichiarato il secret.  
Per ora non abbiamo bisogno di dichiarare anche un config map perché nessuno lo referenzia ancora.  
Procediamo quindi facendo l’apply del deployment per il pod di mongodb e del suo service associato.

Con username e password in **BASE64**

Ovviamente va creato il secret prima del deployment, altrimenti da errore.  
Quindi per prima cosa chiamiamo l’apply sul mongo-secret

**kubectl apply -f mongo-secret.yml**

A questo punto possiamo fare l’apply del deployment di mongo db, e questo lo farà anche startare

**kubectl apply -f mongo.yml**Con I soliti **kubectl get all**¸ **kubectl get pod** ¸**kubectl describe pod ID\_POD**  e **kubectl logs ID\_POD**  
possiamo vedere i vari stati.  
Una cosa interessante è **kubectl describe service ID\_SERVICE** per vedere se effettivamente il service è agganciato al pod

Per vedere lo stato dello startup facciamo   
**kubectl get pod**, prendiamo il nome del pod e facciamo  
**kubectl describe pod ID\_POD -> per vedere i vari stati del pod  
kubectl logs ID\_POD -> per vedere se ci sono stati errori**

**Kubectl get service**prendo id pod e faccio  
**kubectl describe service [IDSERVICE] ->** per vedere effettivamente che il service è attached al pod.

A black background with white text

Description automatically generated with medium confidence

**Endpoints è il pod di destinazione**  
Se voglio controllare che è quello giusto faccio  
**kubectl get pod -o wide** -> mi restituisce anche l’ip (RICORDA CHE PUO’ VARIARE MA IL SERVICE AUTOMATICAMENTE LO RIAGGANCIA) dei pod.

Se ci volessimo collegare via ssh al terminale del pod di mongodb, per interagire col db usando la sheel, potremmo fare   
**kubectl exec [POD\_ID] -it bash**e entrati li potremmo lanciare mongodb e usare l’interprete cli di mongodb.  
Ma visto che è piu’ comodo usare la gui , a questo punto vogliamo configurare CompassMongoDB che è un client installato sul localhost, per fare le query al db che sta nel pod che a sua volta gira nel cluster (che a sua volta sta nella VM VirtualBox).  
Visto che il service del db comunque non è esposto all’esterno (*avremmo potuto comunque renderlo accessibile all’esterno dichiarandolo di tipo NodePort, o Ingress o LoadBalancer, fornendogli il valore del campo nodePort, che è la porta aperta all’esterno*) allora per accedere **usiamo il port-forwarding di minikube.**

**kubectl port-forward [ID\_POD] [PORTA\_ORIGINELOCALHOST]:[PORTA DEL POD, PAR PAFACOLTATIVO]**

**NB: e l’ip a cui collegarci alla porta usata per il port forward sarà sempre IN LOCALE (localhost/127.0.0.1)  
es   
kubectl port-forward pod/mongodb-deployment-7db954584-pvv6r 28015:**E da Compass Mongo DB ci si collega usando la stringa **mongodb://username:password@localhost:28015/**

**(dove username e passoword sono i valori decodificati dal base64 che abbiamo salvati nel secret usato per inizializzare il db)**  
A questo punto ci colleghiamo al db con CompassMongoDB e dobbiamo creare un db chiamato **test  
  
STEP2:**A questo punto dobbiamo creare il nostro applicativo spring boot, configurarne i file di properties in maniera tale che puntino a variabili d’ambiente, buildare il jar e fare l’immagine docker (partendo dal docker file o facendo direttamente senza docker file l’immagine con il BuilPack) e portarla sul dockerhub locale al minikube. A quel punto potremmo usare apply con un deployment.yml che punta all’immagine del nostro applicativo (che dovrà essere visibile quindi al kubernetes) e che avrà dei reference a dei valori di variabili di ambiente (per i properties dell’applicativo) i cui valori verranno presi da config map e secret.  
Quindi per prima cosa creiamo il nostro applicativo spring boot, con le dipendenze maven per mongoDB.  
  
Text

Description automatically generated

E usiamo questo file application.properties  
  
Text

Description automatically generated

Le cose interessanti qui sono il fatto che l’applicativo risponde sulla 8081 (contesto /testkub) e che  
la stringa url di connessione al db, che per mongodb deve avere forma  
**mongodb://user:pw@host:porta/dbname**in questo caso è tutta costruita con valori ${} che provengono dall’environment (in quanto non presenti essi stessi nel properties) con i valori dopo i : nelle {} per quando non li trova (poiché se lo startiamo in locale non riuscirà a trovarli, mancando quelle variabili nell’env visto che non ci troviamo ancora nel container docker=  
Facciamo implementazione (che risponde a /teskhub/helloworld ) e proviamo a buildare con   
**mvn clean install.**Una volta buildato proviamo a runnare con  
**mvn spring-boot:run**Ovviamente salirà ma non riuscirà a collegarsi al db perché i valori di fallback che abbiamo dato non funzionano.  
  
Quindi se tutto funziona per prima cosa dobbiamo fare la build dell’immagine Docker, e portarla sul repository visibile a minikube (che non ha accesso a quello della macchina locale di docker desktop).  
Per fare questo lanciamo  
  
**minikube docker-env**Copiamo l’ultima riga (escluso il rem) e la lanciamo.  
Di solito  
**@FOR /f "tokens=\*" %i IN ('minikube -p minikube docker-env --shell cmd') DO @%i**  
 Questa riga fa si che tutti i successivi comandi lanciati in quella bash, andranno verso il docker del minikube e non quello locale.  
  
A questo punto o usiamo un Dockerfile per il progetto, e facciamo docker build del docker file (linkando al jar prodotto nel target/ dell’applicazione spring boot) oppure andiamo nella folder dove è presente il .pom del progetto e lanciamo  
**mvn spring-boot:build-image -Dspring-boot.build-image.imageName=testkub**  
Questo fa si che il BuildPack automaticamente faccia la build e la porti nel repository sul kubernetes.  
Per essere certi che sia presente sul minikube lanciamo  
**minikube image ls**

A questo punto creiamo il deployment.yaml dell’applicativo spring boot, che fa riferimento al nome dell’immagine buildata (e portata nel minikube) , e alle variabili che vanno create nell’environment del container.  
Nello stesso file è anche definito il service agganciato al pod dell’applicativo Text

Description automatically generated

Si noti che :  
la porta aperta del container è la 8081 perché è la porta che abbiamo dichiarato nell’application.properties che deve essere quella applicativa.  
Si noti inoltre che le variabili che creiamo dell’env hanno lo stesso NAME per come erano usate nell’application.properties, e i valori vengono presi o dal secret (già creato) o dal config map (che dobbiamo creare).  
Si noti inoltre che il servizio associato a questo pod, è di tipo NodePort, quindi è aperto all’esterno, e la porta esposta all’esterno è la 30081 (la sua porta interna è la 8080, ma nessuno la userà) mentre la porta del pod a cui gira è la 8081 (perché è quella aperta sul pod).  
Se quindi inizializziamo prima il config map facendo  
**kubectl apply -f mongo-configmap.yaml**e poi lanciamo l’apply del deployment dell’applicativo spring boot e del suo service raggiungibile da fuori con  
**kubectl apply -f springbootapp.yaml**possiamo controllare con **kubectl get all** che è tutto funzionante, e preso l’id del pod fare  **kubectl logs POD\_ID** per vedere le loggate dell’applicativo spring boot che sale.

Ovviamente possiamo accedere all’applicativo collegandoci all’ip del nodo del cluster (ottenibile facendo **kubectl cluster-info**) sulla porta esposta cioè la 30081, oppure lanciando il comando  
**kubectl service [ID\_SERVIZIO\_DELL’APPLICATIVO]**   
nel caso di esempio  
<http://192.168.59.101:30081/testkub/hello>  
  
Graphical user interface, text, application

Description automatically generated

**INGRESS**Abbiamo visto che esistono vari tipi di servizi in k8.  
Il default è ClusterIP, che non è raggiungibile dall’esterno.  
Poi abbiamo NodePort e LoadBalancer, che sono raggiungibili dall’esterno su di una data porta esposta sul nodo.  
In realtà mentre NodePort e LoadBalancer sono servizi associati ad un pod, gli **Ingress** invece sono servizi che fanno multiplexing verso servizi .  
Un ingress viene mappato su uno o piu’ hostname, e gestisce tutte le richieste che hanno nell’url quell’hostname, con i vari sottopath. Ciascuna viene girata verso un servizio e una porta.  
Questo permette di tenere i servizi multiplexati solamente interni.  
NB: gli ingress permettono di fare multiplexing usando 2 parti dell’url : l’host e il path.  
Tuttavia l’host (es [www.myapp.com](http://www.myapp.com)) deve essere registrato nel dns (o nel file localhost) sull’ip esterno che kubernetes associa all’ingress.  
Quindi un ingress viene creato come sempre usando un deployment.yml.  
Pero’ un ingress da solo non basta, l’ingress rappresenta solo la mappa per il routing.  
Quello che effettivamente fa il routing è detto **INGRESS CONTROLLER.**Ci sono varie implementazioni di INGRESS CONTROLLER, il default è **K8s Nginx Ingress Controller**

Tornando all’esempio dell’applicazione spring boot, noi l’avevamo esposta creando un service di tipo NodePort che esponeva sulla porta 30001, e il servizio era associato all’applicativo spring boot.  
Se invece volessimo usare l’ingress, basterebbe fare l’apply del file.  
  
Prima di fare questo pero’ dobbiammo creare l’ingress controller facendo  
**minikube adadons enable ingress ->** abilita l’ingress controller (nginx ingress controller)  
  
Text, timeline

Description automatically generated

Questo vuol dire che quando si va all’url [www.demo.com](http://www.demo.com) (che dobbiamo registrare ancora nell’host) senza nessun path aggiuntivo, si viene dirottati verso il servizio agganciato al pod dell’applicazione spring boot. La porta destinazione è la 8080 perché il servizio associato al pod spring boot (che avevamo dichiarato nell’esempio precedente) ha 8080 come porta interna (la 30001 era l’esterna) **ma l’ingress aggancia SULLA PORTA INTERNA DEL SERVIZIO multiplexato.**Quindi faremo   
**kubectl apply -f demo-ingress.yaml**E questo crea l’ingress .  
A questo punto se facciamo  
**kubectl get ingress**  e vediamo che ip è stato assegnato all’ingress, andiamo sul file di **host**  e registriamo tale ip all’host name indicato nell’ingress (demo.com).

**(FACENDO kubectl get all non vengono mostrati gli ingress!)**Quindi per debuggare gli ingress:  
**kubect get ingress   
e  
kubectl describe ingress INGRESS\_NAME -> per vedere se ci sono errori nell’associazione**  
Quindi se andremo su [www.demo.com/testkub/hello](http://www.demo.com/testkub/hello) staremo andando verso l’applicazione nel pod di spring boot usando l’ingress invece che la node port.  
  
Quindi l’ingress ci permette di avere tanti servizi interni e fare multiplexing su ciascuno di essi a seconda del path e dell’hostname, invece di dover esporre ciascuno dei servizi per ogni applicativo all’esterno ognuno con una sua porta, poiché la porta sull’ingress sarà unica, ma verrà usato path e hostname per scegliere a quale servizio agganciarsi.  
  
**(NB con gli ingress è anche possibile definire un servizio di fallback  
NB con gli ingress si possono anche usare i certificati TSL per avere HTTPS )**

**Tipicamente davanti agli ingress nei provider cloud si mette un LOAD BALANCER di terze parti**

**NB: per cancellare un pod bisogna cancellare il deployment rispettivo.**

Discorso dei volumes pv,pvc etc.

Tipicamente questi

Come abbiamo visto negli esempi precedenti, i database che hanno bisogno di salvare i propri dati usano un persistent volume (pv,che è la dichiarazione di uno spazio di persistenza “fisico” sulla VM) e di un persistent volume claim (pvc, che è l’effettivo mount sul pod dello spazio), e poi il container del db lo referenzia nel suo deployment spec.

Questo non vale solo per i database ma anche per le applicazioni di immagini proprie.  
Quando si lavora con il deployment di un applicativo che ha bisogno di accedere al fs, tipicamente (come fatto per mysql) viene creato prima un PV (persistent volume) poi si crea un PV claim (cioè l’uso di un persistent volume). Il primo monta sul FS di k8s la folder, e il secondo viene usato nel deployment di un applicazione per fare il CLAIM di quel PV, in maniera tale che tramite il claim k8 lo mette nel pod dell’applicazione.  
Tipicamente questi PV in realtà andrebbero sostituiti con delle risorse ESTERNE al k8s, cioè dei NFS esterni.  
Inoltre minikube permette un ulteriore livello di semplificazione, in quanto invece di dichiarare il PV, dichiarare il PV CLAIM, e poi usarlo nel deployment del pod dell’applicazione che lo usa, si puo’ dichiarare direttamente SOLTANTO nel deployment dell’applicazione il full path di una cartella presente sulla VM in cui sta girando il minikube. Quindi automaticamente il POD in minikube potrà accedere direttamente a questa folder sulla VM (il pod sta in minikube e il minikube sta nella vm).  
Per fare questo bisogna dichiarare 2 cose nel deployment dell’applicazione.  
Come da immagine  
Text

Description automatically generated  
Cioè sia il volumes.hostpath.path sia il volumeMounts.mountPath.  
(**non mi è chiara la differenza, ma sembrerebbe che il secondo deve avere effettivamente il percorso ASSOLUTO della reale cartella presente sulla VM)**  
Ovviamente questa folder deve esistere sulla VM, e deve essere il percorso assoluto partendo dalla root. In questo caso è stata configurata sulla VirtualBox (VM) una folder chiamata minikube\_sharedfolder che sta proprio nella root, ed è condivisa rispetto alla macchina di host (windows). A sua volta verrà quindi condivisa dal pod.  
Quindi la struttura sarebbe

**MACCHINA HOST (es Windows), su questa gira: (folder A1 condivisa con VM)  
 🡪VM (es VirtualBox), su questa gira: (folder A2 gemella di A1)  
 🡪 MINIKUBE , su questo gira:   
 🡪 GIRA IL POD (accede direttamente ad A2)**

Come debuggare da eclipse in remoto verso un ‘applicazione che gira su un pod k8/minikube.

Innanzitutto bisogna settare nel deployment dell’applicazione gli args che vanno passati allo startup dell’applicazione per abilitare il debug su porta (es 5005)

Text

Description automatically generated

A questo punto bisogna aprire la porta scelta (5005 in questo caso) all’esterno del cluster k8s.  
Per fare questo è inutile creare un servizio ad hoc, quindi basta fare port forwarding.  
  
**kubectl port-forward POD\_ID 5005:5005**Quindi ora si puo’ creare una config in eclipse di debug remoto che punta a localhost verso la porta 5005.