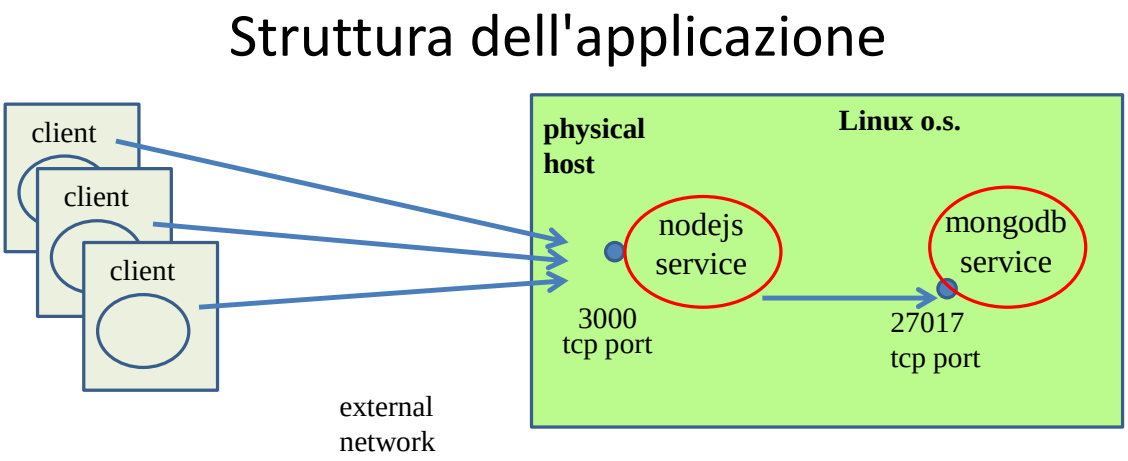
Un esempio di uso di docker.

Prof. Vittorio Ghini - Sistemi Operativi - 20/12/2019

L'esempio di applicazione da realizzare.

Vediamo passo per passo come costruire una applicazione web, implementata mediante **nodejs**, che sfrutta le librerie express per interfacciarsi un database gestito dal dbms **mongodb**.

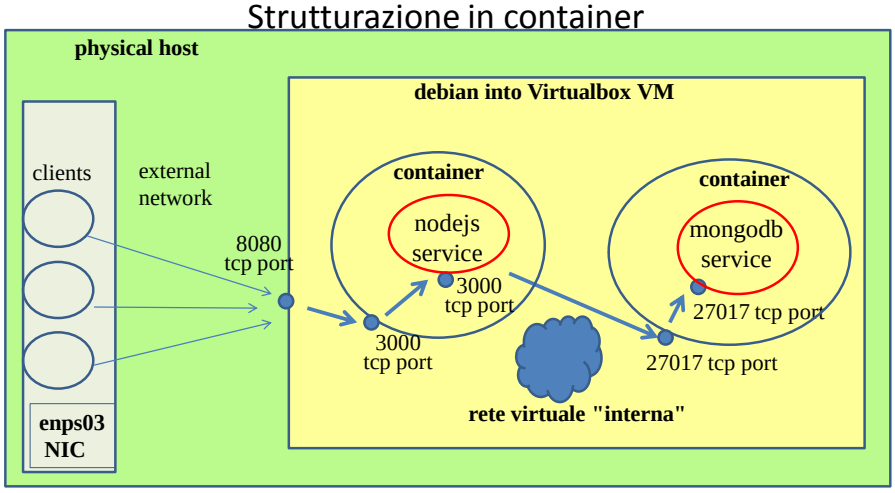


Cosa fa l'applicazione:
Il server web nodejs accetta richieste HTTP di tipo POST contenenti due parametri seq1 e seq2. Il server controlla se nel database documentale esiste già una quadrupla (seq1 seq2 as1 as2) in cui i primi due campi sono proprio quelli ricevuti dal server web. In tal caso il server web restituisce al client la quadrupla trovata. Se invece la quadrupla non esiste già, allora il server web calcola due nuove stringhe as1 e as2, funzione dei due parametri seq1 e seq2, poi salva nel database la quadrupla formata da seq1 seq2 as1 as2, infine restituisce al client la quadrupla salvata.

Struttura in container dell'applicazione:

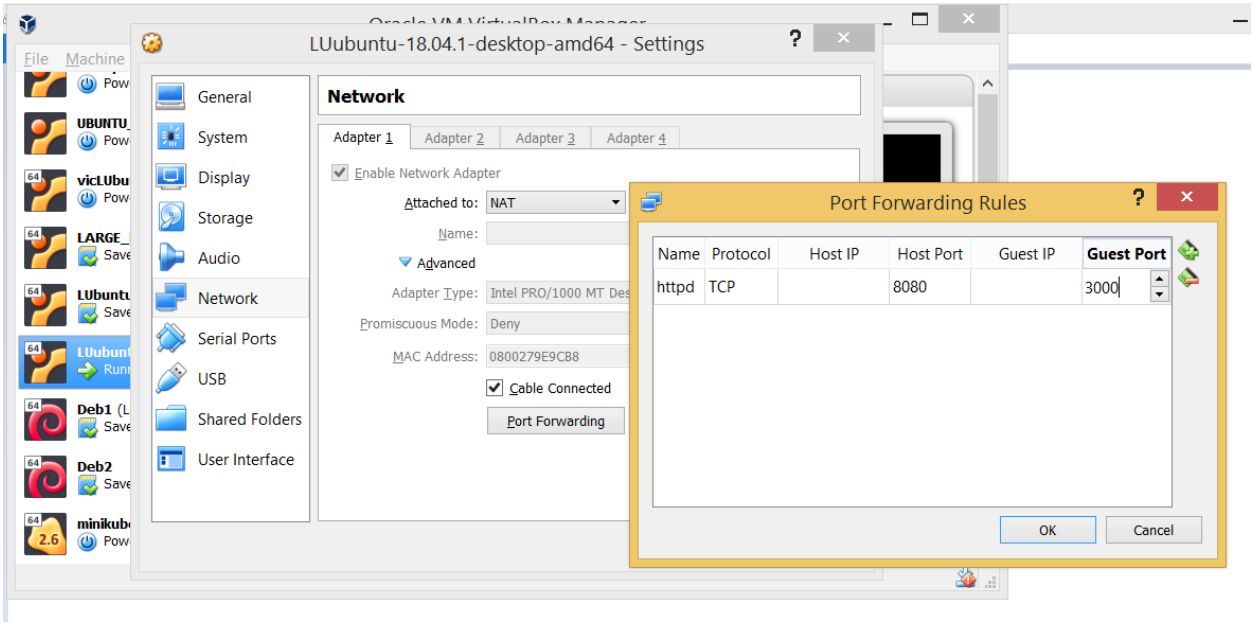
Ciascuno dei due componenti, uno nodejs con express e mongoose e l'altro mongodb, viene realizzato e pacchettizzato in un proprio **container docker**.
L'applicazione (formata dai due container) viene costruita ed eseguita in una macchina Linux, sfruttando il pacchetto docker (ed eventualmente il pacchetto docker-compose, se si volesse automatizzare all'estremo).

Strutturazione in container



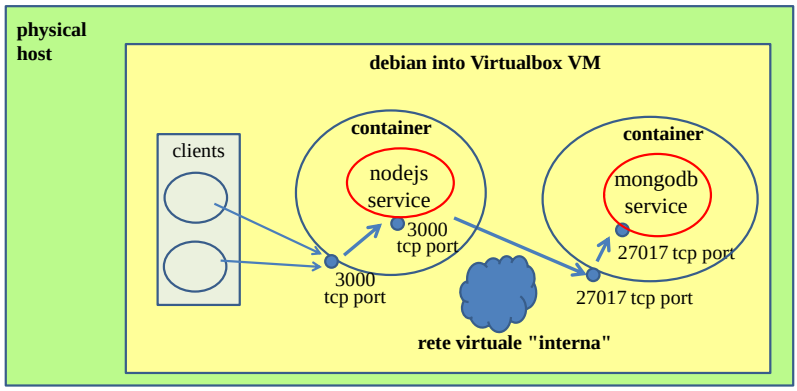
La struttura dell'applicazione, e la sua organizzazione in container è rappresentata nello schema qui sopra, in cui è indicato il caso in cui l'applicazione è eseguita su un o.s. Linux ospitato all'interno di una macchina virtuale (gialla) gestita da virtualbox; nello schema compaiono anche i client web (browser), che non consideriamo parte dell'applicazione. Ovviamente, nulla vieta che l'applicazione venga eseguita

Qualora l'applicazione venga eseguita all'interno di una macchina virtuale in Virtualbox, occorrerà configurare virtualbox affinchè renda visibile all'esterno della macchina virtuale la porta su cui si vuole ricevere le connessioni provenienti dai client. Si veda a tale proposito la figura che descrive come configurare il port forwarding in virtual box, esponendo la porta 3000 sulla porta esterna 8080.



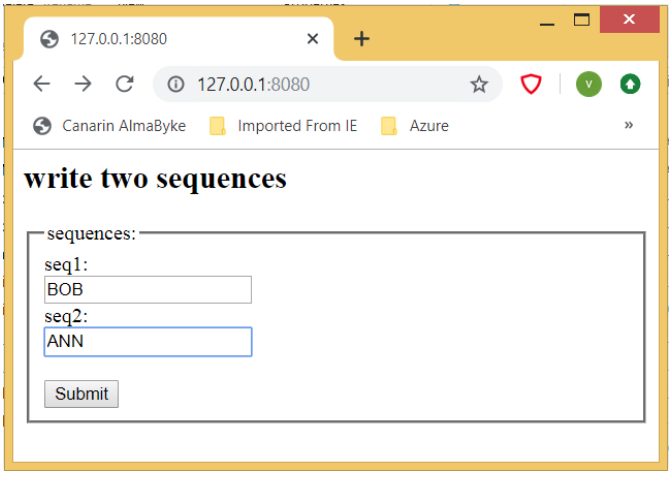
Nulla vieta che, per semplicità di utilizzo, i test vengano eseguiti collocando i client all'interno della macchina virtuale stessa, come da schema nella seguente figura.

Strutturazione in container
test con client collocati nella macchina virtuale



Solo in tal caso, non occorrerà più configurare Virtualbox per esporre la porta del servizio fuori dalla macchina virtuale stessa; i client si collegheranno alla porta esposta esternamente al container di nodejs.

Per consentire un facile utilizzo mediante un browser fuori della macchina virtuale, è stata aggiunta una pagina html, generata dinamicamente, che viene ottenuta mediante una richiesta GET / e che visualizza una pagina web con un FORM che richiede le due sequenze.
In tal modo, dall'esterno della macchina virtuale, è possibile utilizzare un browser e accedere alla pagina web con indirizzo http://127.0.0.1:8080/ che espone il FORM e poi effettua la richiesta POST con le due sequenze digitate nella form.



Il codice sorgente dell'applicazione e tutti i file che servono a creare i container dell'applicazione e a metterli in esecuzione (in pratica tutti i file citati in questo documento) sono contenuti nell'archivio gzippato SISTOP_Docker.tgz.

Organizzazione del documento

Questo documento è organizzato in 5 parti:

1) **installazione dei pacchetti docker e docker-compose** in una macchina Linux.

2) Costruzione della rete virtuale "interna" che connette tra loro i due container del web server nodejs e di mongodb.

3) **Costruzione ed esecuzione del container con dentro l'applicazione web basata su nodejs** ed express, con particolare accento sul modo di garantire che il servizio nodejs riesca collegarsi al servizio del container mongodb. Ciò sarà garantito collocando entrambi i container in una rete virtuale, precedentemente create, che chiameremo "interna" che automaticamente realizza un DNS implicito. In tal modo, il nome del container di mongodb potrà essere usato come nome dell'host virtuale in cui mongodb è in esecuzione e potrà quindi essere risolto in un indirizzo IP mediante il quale nodejs si collega a mongodb.

4) **Perché occorre modificare l'immagine di default dell'applicativo mongodb.**

5) **Costruzione dell'immagine del container con mongodb** ed il database.
Vedremo come realizzare il container con il dbms mongodb che mantiene e gestisce il database dbsa con una collection 'alignments'. Definiremo un file Dockerfile che consente di creare (build) l'immagine del container con mongodb ed il nostro database.

6) **Esecuzione del container con mongodb ed il database dbsa.**
Indicheremo poi come effettuare manualmente il dispiegamento e l'esecuzione del container.

7) **Esecuzione del container con dentro l'applicazione web** basata su nodejs, express e mongoose.

8) **Automatizzazione di costruzione, dispiegamento ed esecuzione di tutti i container.**
Armonizzazione ed Automatizzazione di costruzione, dispiegamento ed esecuzione di tutti i container necessari all'applicazione e dell'ambiente di esecuzione dei container, ed infine terminazione dell'applicazione, mediante un Makefile per semplificare le operazioni.

1) Installazione dei pacchetti docker e docker-compose in una macchina Linux.

1.1) Installazione di docker in una macchina Lubuntu.

Prerequisites: Ubuntu 18.04 LTS, an account to Docker Hub
Installing Docker Community Edition (CE) from Docker repository (not from Linux distribution).

- update your packages list
  sudo apt-get update
- install some packages to allow apt use packages over https
  sudo apt install apt-transport-https ca-certificates curl software-properties-common
- add the GPG key for the official Docker repository to your system
  curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

- Add the Docker repository to APT sources
  sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu bionic stable"
- update the package database
  sudo apt update
- Verify the repository (docker repo, not the default ubuntu repository)
  apt-cache policy docker-ce
- Install Docker
  sudo apt install docker-ce docker-ce-cli containerd.io
- Check that docker daemon is running
  sudo systemctl status docker

Installing Docker now gives you not just the Docker service (daemon) but also the docker command line utility, or the Docker client.

1.2) Come eseguire il comando docker senza dover usare sudo.

- Inserisci l'utente nel gruppo docker
  sudo usermod -aG docker ${USER}
  ad esempio: sudo usermod -aG docker vic
- Verifica che l'utente ora appartiene al gruppo docker
  id -nG
- Ricarica l'utente
  su - ${USER}
  ad esempio: su - vic

1.3) Installazione di docker-compose in una macchina Lubuntu in cui è già stato installato docker (Docker compose is a tool that helps in deployment of applications).

Queste istruzioni installano la versione 1.24.0 di docker-compose dal repository di docker, non da quello di Ubuntu. Per verificare se la 1.24.0 è ancora le versione più recente, consultare su github la pagina delle release di Docker. In caso esistesse una versione più recente, sostituire nei seguenti comandi la stringa 1.24.0 con la stringa che indica la versione che si vuole installare.

- sudo apt update
- sudo apt install py-pip python-dev libffi-dev openssl-dev gcc libc-dev make
- sudo apt install py-pip python-dev libffi-dev openssl-dev gcc libc-dev make
- sudo curl -L "https://github.com/docker/compose/releases/download/1.24.0/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
- sudo chmod +x /usr/local/bin/docker-compose
- docker-compose --version

1.4) Cos'é il registry di docker

Il registry locale di docker è un servizio installato quando viene installato docker. Mantiene nel filesystem le immagini dei container che sono state utilizzate.

2) Costruzione della rete virtuale "interna" che connette tra loro i due container del web server nodejs e di mongodb.

Creo una rete virtuale che verrà usata per comunicare tra i container e la chiamo "interna". Lo scopo vero è non dover definire degli indirizzi IP per i container e sfruttare invece i **nomi dei container come se fossero dei nomi di host, che verranno risolti in indirizzi IP dal DNS implicito che docker realizza automaticamente per ciascuna rete virtuale che crea**. I container che usano una stessa rete virtuale possono vedere gli altri container di quella rete usando i nomi dei container come se fossero dei nomi di host, ed il DNS implicito li risolve. Creo la rete virtuale usando il comando:

docker network create -d bridge interna

Verificare l'esistenza della rete con il comando

docker network ls

E, per vedere più dettagli sulla rete denominata "interna", eseguire:

docker network inspect interna

3) Costruzione dell'immagine del container con dentro l'applicazione web basata su nodejs, express e mongoose

Andare nella directory ./SISTOP/nodejs in cui sono presenti i file necessari.

./app
./app/package-lock.json
./app/package.json
./app/**index.js**
./app/src
./app/src/controllers
./app/src/controllers/controller.js
./app/src/lib
./app/src/lib/sequences_alignment.js
./app/src/routes
./app/src/routes/routes.js
./app/src/models
./app/src/models/alignmentModels.js
./**Dockerfile**

C'è un **Dockerfile** ed una sottodirectory app che contiene tre files index.js package.json e package-lock.json ed una sottodirectory src a sua volta contenente 4 sottodirectory: controllers,lib,models,routes. Le quattro sottodirectory contengono l'applicazione vera e propria.

Il file package.json contiene le dipendenze dell'applicazione, cioè le informazioni che servono a capire quali pacchetti per nodejs installare per poter eseguire correttamente tutte le funzioni richiamate nel codice javascript dell'applicazione.

Il file index.js è quello che deve essere eseguito per far partire l'applicazione vera e propria, mediante il comando:

nodejs index.js

All'interno del file index.js **c'è la parte che stabilisce la connessione tra il web service ed il dbms mongodb** mediante la libreria express. Il file index.js è così fatto:

```
// file index.js

var express = require('express');
var bodyParser = require('body-parser');
var mongoose = require('mongoose')
var Alignment = require('./src/models/alignmentModels')

//Creo istanza di express (web server)
var app = express();
```

```
//importo parser per leggere i parametri passati in POST
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());

//connessione al db
// il secondo mongodb, quello colorato in rosso,  e' il nome del
container di mongo
mongoose.connect('mongodb://mongodb/dbsa', { useNewUrlParser: true,
useFindAndModify: false });

//mongoose.connect('mongodb://username:password@host:port', {
useNewUrlParser: true, useFindAndModify: false });

var routes = require('./src/routes/routes');
routes(app);

//metto in ascolto il web server
app.listen(3000, function () {
  console.log('Node API server started on port 3000!');
});
```

Notare che l'istruzione:

```
mongoose.connect('mongodb://mongodb/dbsa', { useNewUrlParser: true,
useFindAndModify: false });
```

usa il nome del container (**mongodb, quello colorato in rosso**) come se fosse un nome di host, poiché sappiamo che il container di mongodb lo collocheremo nella stessa rete virtuale chiamata "interna" del container dell'applicazione, e quindi il DNS implicito riuscirà a risolvere il nome del container nel'indirizzo IP del container all'interno della rete virtuale "interna".

Notare anche l'istruzione generica commentata

```
//mongoose.connect('mongodb://username:password@host:port', {
useNewUrlParser: true, useFindAndModify: false });
```

che ricorda come sia possibile inserire anche uno username ed una password per accedere a mongodb. Nel nostro caso non è necessario poiché abbiamo configurato mongodb per non richiedere username e password.

Poi scriviamo (c'é già) il **Dockerfile** nella directory nodejs per costruire l'immagine del container nodejsapp dell'applicazione web.

```
FROM ubuntu:xenial

ENV WORKINGDIR=/root/app
# indica dove verranno eseguiti i comandi indicati da RUN
WORKDIR ${WORKINGDIR}

RUN   mkdir -p ${WORKINGDIR} && chmod 666 ${WORKINGDIR}

# copia tutta la directory app e le sottodirectory nel container
COPY  ./app  ${WORKINGDIR}/
```

```
# installo nel container i pacchetti necessari
# e poi butto gli archivi non più utili
RUN   apt-get -y update                &&    \
      apt-get -y install apt-utils     &&    \
      apt-get -y install nodejs        &&    \
      apt-get -y install npm           &&    \
      apt-get -y clean
# installo nel container le dipendenze indicate in package.json
RUN   npm install

# ricordo che devo rendere accessibile dall'esterno del container
# la porta 3000 su cui nodejs rimane in attesa di connessioni
# da parte dei client
# Occhio, non espongo la porta, questa EXPOSE è solo una nota per
# ricordare che devo esporre la porta quando eseguo il container.
EXPOSE 3000

# dico quale comando eseguire automaticamente quando eseguirò il
# container: serve per far partire il servizio web nel container.
CMD nodejs index.js
```

A questo punto è tutto pronto, posso costruire l'immagine del container con nodejs e la nostra applicazione, andando nella directory nodejs contenente il Dockerfile, ed eseguendo il comando:

docker build -t nodejsapp .

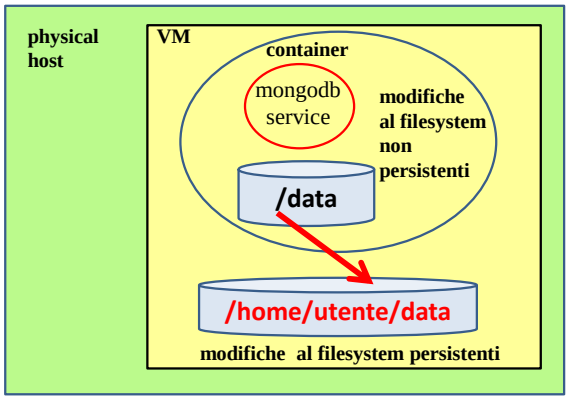
Il comando impiega un poco di tempo e poi finisce scrivendo su disco l'immagine di un container con nome nodejsapp.
Verificare l'esistenza con il comando:

docker images | grep nodejsapp

4) Perché occorre modificare l'immagine di default dell'applicativo mongodb.

Il tipico container di mongodb inserisce la propria configurazione ed i file in cui memorizza i dati dei suoi database, in una sua directory /data
Però, poiché il database potrebbe essere terminato volutamente o per crash, con la terminazione si perderebbero le modifiche apportate nel frattempo alla base di dati, perché non si salverebbero le modifiche.
Per tale motivo, **il container mongodb è fatto per mappare la propria directory interna /data in una directory esterna al container, cioè una directory della macchina fisica per assicurare la persistenza.**
In questo modo, se anche il container termina, il database rimane nel disco fisico.

Rendere persistenti le modifiche al DB
salvando in volume esterno la
configurazione ed i dati



Più precisamente, il **container mongodb è fatto per mappare separatamente due proprie directory /data/db e /data/configdb in due directory esterna al container, cioè due directory della macchina virtuale.**

5) Costruzione dell'immagine del container con mongodb ed il database.

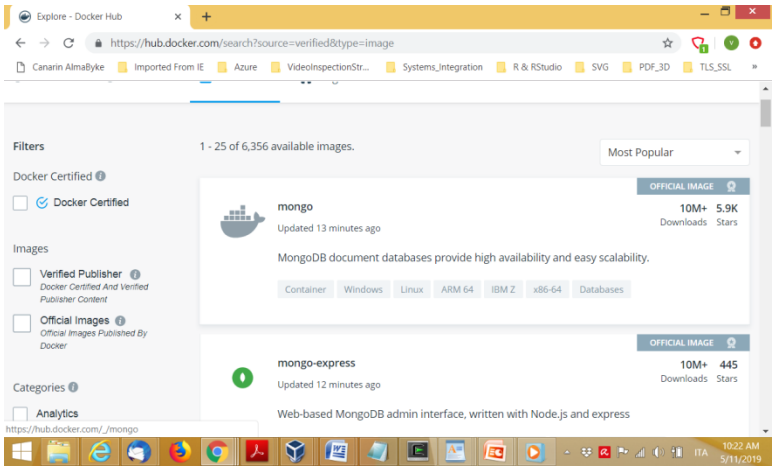
Creare una directory SISTOP e due sottodirectory nodejs e mongodb in cui metteremo i file per costruire il container rispettivamente per l'applicazione web e per il database documentale.

Andare nella directory ./SISTOP/mongodb

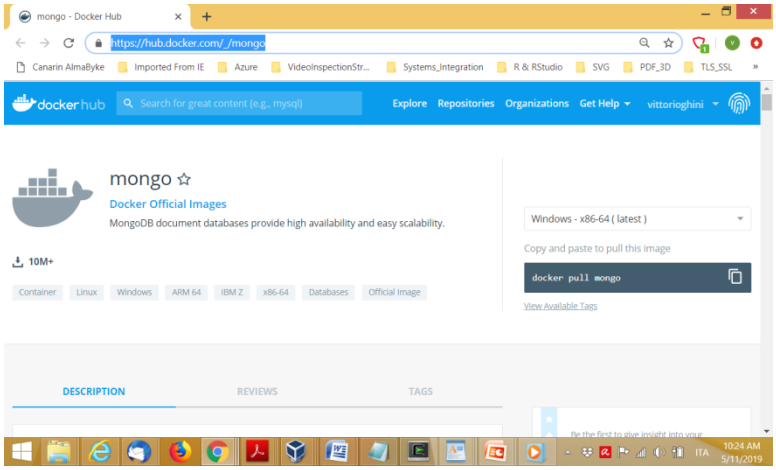
5.1) Recuperare i file necessari al build dell'immagine per mongodb.

Per prima cosa devo cercare i file necessari al build dell'immagine del container di mongodb, in particolare il Dockerfile ed un file docker-entrypoint.sh, che da quel Dockerfile viene usato e copiato all'interno del costruendo container.

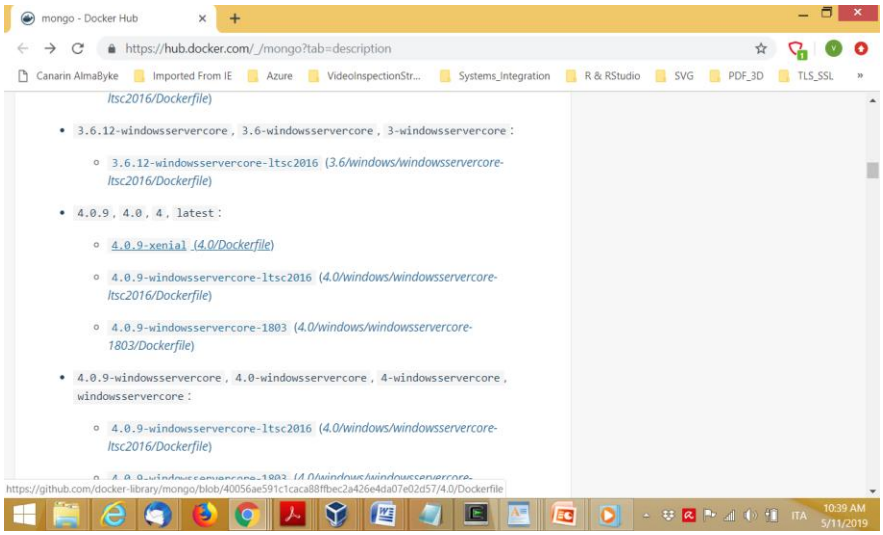
Cercare l'immagine ufficiale di mongodb sul repository di dockerhub:
https://hub.docker.com/search?source=verified&type=image



Per informazioni, aprire la pagina nel link "Container" di mongo:
https://hub.docker.com/_/mongo



Selezionare la sezione "Description" e scorrere la pagina e trovare nella sezione "Shared Tags" la parte riferita alla versione "latest" per Linux: è la 4.0.



Aprire la pagina del Dockerfile al link individuato da "4.0.9-xenial(*4.0/Dockerfile*)" che punta a: https://github.com/docker-library/mongo/blob/40056ae591c1caca88ffbec2a426e4da07e02d57/4.0/Dockerfile

Nella pagina web del Dockerfile appena aperta, copiare il link del Dockerfile RAW contenuto nel riquadro "RAW":
https://github.com/docker-library/mongo/raw/40056ae591c1caca88ffbec2a426e4da07e02d57/4.0/Dockerfile

Salvare localmente il Dockerfile con il seguente comando:

```
wget https://github.com/docker-library/mongo/raw/40056ae591c1caca88ffbec2a426e4da07e02d57/4.0/Dockerfile
```

Nella pagina web del Dockerfile, risalire di un livello cliccando sul link individuato dalla stringa "mongo / 4.0 ":

```
https://github.com/docker-library/mongo/tree/40056ae591c1caca88ffbec2a426e4da07e02d57/4.0
```

Appare l'elenco dei file da usare per il build dell'immagine. Cliccare sul link per il file docker-entrypoint:

https://github.com/docker-library/mongo/blob/40056ae591c1caca88ffbec2a426e4da07e02d57/4.0/docker-entrypoint.sh

Nella pagina che si apre copiare l'indirizzo del file docker-entrypoint.sh RAW contenuto nel riquadro "RAW":

```
https://github.com/docker-library/mongo/raw/40056ae591c1caca88ffbec2a426e4da07e02d57/4.0/docker-entrypoint.sh
```

Salvare localmente il file docker-entrypoint.sh con il seguente comando:

```
wget https://github.com/docker-library/mongo/raw/40056ae591c1caca88ffbec2a426e4da07e02d57/4.0/docker-entrypoint.sh
```

5.2) Scrivere i file necessari ad inizializzare il db mongodb.

Predisponiamo una prima modifica, per inizializzare nel nostro database mongo, un database che chiamo "dbsa" ed una collection che chiamo "alignments" e per inserire almeno un documento nella collection.

Creo e scrivo un file in linguaggio javascript che farò eseguire dal mio mongodb solo la prima volta che lo metterò in esecuzione. Infatti, tutti i file di scripting .js o .sh che, nel container, si trovano nella directory /docker-entrypoint-initdb.d/ verranno eseguiti la prima volta che il container viene messo in esecuzione, o meglio, la volta che il container viene messo in esecuzione e nel db non ci sono database.
Chiamo il file **"mydbinit.js"** e questo è il contenuto:

```
var conn = new Mongo();
var db = conn.getDB('dbsa');

// crea collection 'alignments' e la lascia se già esiste
db.createCollection('alignments', function(err, collection) {});

// elimina gli eventuali documenti della collection 'alignments'
// nel caso esistesse già
try { db.alignments.deleteMany( { } ); } catch (e) { print (e); }

db.alignments.insert({\"s1\": \"GCATGCU\", \"s2\": \"GATTACA\",
\"as1\": \"GCATGC-U\", \"as2\": \"G-ATTACA\"})
```

Poi assegno i permessi di esecuzione a tutti per gli script:

```
chmod 777 mydbinit.js docker-entrypoint.sh
```

5.3) Modificare una prima volta il Dockerfile per far eseguire l'inizializzazione del DB alla prima esecuzione del container.

Aggiungere, subito dopo la riga commentata, una riga in cui si **ordina di copiare il file mydbinit.js** nella directory /docker-entrypoint-initdb.d/ del container in cui mongodb cercherà gli script da eseguire dopo che il container è stato messo in esecuzione.

```
COPY ./mydbinit.js /docker-entrypoint-initdb.d/
```

5.4) Effettuare il build dell'immagine del container.

Abbiamo i file che ci servono, ora li modifichiamo e facciamo il build della nostra immagine.

Creare l'immagine nuova di mongodb, che chiamo "mymongo" lanciando il build che usa il nuovo Dockerfile:

docker build -t "mymongo" . **NB:C'è un punto alla fine**

Se tutto va a buon fine, vedrò tra le immagini del registry locale anche la mia immagine mymongo, lanciando il comando:

docker images | grep mymongo

Quella è l'immagine del container creato e salvato in locale.

6) Esecuzione del container con mongodb ed il database dbsa

Verifico che la rete virtuale che ho chiamato "interna" e che servirà a connettere i due container sia ancora in esecuzione, eseguendo il comando:

docker network ls -f name=interna

mi aspetto un output di questo tipo

NETWORK ID	NAME	DRIVER	SCOPE
7e9f77130cd4	**interna**	bridge	local

Se per caso la rete non esistesse dovrei crearla, come indicato al precedente punto 3, eseguendo il comando:

docker network create -d bridge interna

Creata la rete virtuale, **metto in esecuzione in background** il container contenente mongodd, partendo dall'immagine che ho creato, eseguendo:

docker run -itd --network interna -p 27017-27019:27017-27019 --name mongodb mymongo

Notare che ho assegnato come nome "mongodb" al container che metto in esecuzione.
Verifico che il container battezzato mongodb sia in esecuzione, invocando il comando:

docker ps -a

Mi aspetto un output come segue:

```
CONTAINER ID   IMAGE    COMMAND                  CREATED         STATUS       PORTS
NAMES
0f70e9a6f83e   mymongo  "docker-entrypoint.s…"  7 minutes ago  Up 7 minutes
0.0.0.0:27017-27019->27017-27019/tcp   mongodb
```

In caso di errori potreste vedere che lo status del container non è Up bensì è **Exited**. In tal caso, lanciate il comando seguente per visualizzare il contenuto del log del container chiamato mongodb e cercare di capire cosa è successo. Potete usarlo in generale per debugging anche mentre il container è ancora in esecuzione.

docker logs mongodb

Se invece il container è in esecuzione normalmente (status Up), è bene verificare che tutto sia funzionante; perciò **faccio eseguire, nel container di mongodb in esecuzione, un comando che usa la shell del client mongo per eseguire uno script .js passato a riga di comando**. Questo script interroga mongodb chiedendo di visualizzare il contenuto della collection

docker exec -it mongodb /usr/bin/mongo --eval "var conn = new Mongo(); var db = conn.getDB('dbsa'); var cursor = db.alignments.find(); while (cursor.hasNext()) { printjson(cursor.next()); }"

Ci aspettiamo un output come quello che segue, che indica che nel nostro db dbsa c'è un solo documento:

```
MongoDB shell version v4.0.9
connecting to:
mongodb://127.0.0.1:27017/?gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("3df697d6-f2b8-46c5-a69d-
43ea51d04b3d") }
MongoDB server version: 4.0.9
{
      "_id" : ObjectId("5cd6a937518b4b8160497e25"),
      "s1" : "GCATGCU",
      "s2" : "GATTACA",
      "as1" : "GCATGC-U",
      "as2" : "G-ATTACA"
}
```

6.1) Accedere ai servizi del container mongodb via client mongodb (non ancora via web).

A questo punto possiamo utilizzare il container mongodb mediante applicativi che si connettono alla porta esposta dal container.

6.1.1) Query scritta tramite console interattiva del client mongodb.

Ad esempio, se nell'host locale abbiamo installato il client per connetterci a mongodb (sudo apt install mongodb-clients), allora possiamo connetterci al server mongodb, e in particolare al nostro database dbsa, aprendo come interfaccia la shell di comandi del client mongodd, così:

```
mongo 127.0.0.1:27017/dbsa
```

Dall'interno di questa shell interattiva potremo lanciare comandi, ad esempio:

```
use dbsa
db.createCollection("alignments")
db.alignments.find()
db.alignments.insert({"s1": "GCATGCU", "s2": "GATTACA", "as1":
"GCATGC-U", "as2": "G-ATTACA"})
db.alignments.find()
exit
```

6.1.2) query passata a riga di comando al client mongodb.

Oppure, sempre usando la shell di comandi del client mongo, potremmo direttamente eseguire delle query a riga di comando, passando delle query realizzate con degli script js, ad esempio inserendo dei nuovi documenti, così:

```
mongo 127.0.0.1:27017/dbsa --eval "var conn = new Mongo(); var db
= conn.getDB('dbsa'); db.alignments.insert({\"s1\": \"CAZCAZ\",
```

```
\"s2\": \"ZACZAC\", \"as1\": \"CAZCAZ-U\", \"as2\": \"G-
ZACZACVAF\"});"
```

oppure visualizzando quelli esistenti, così:

```
mongo 127.0.0.1:27017/dbsa --eval "var conn = new Mongo(); var db
= conn.getDB('dbsa'); var cursor = db.alignments.find(); while (
cursor.hasNext() ) { printjson( cursor.next() ); }"
```

6.2) Terminazione del container mongodb.

Quando non vi serve più il container mongodb potete terminarlo. **Il momento in cui decidete di terminare l'esecuzione del vostro container non è critico**, poiché le modifiche effettuate dal container nel database sono state via via salvate nella directory ESTERNA al database. Non perdete le modifiche che avete introdotto nel database. Possiamo perciò terminare il container ed eliminare il container stesso:

```
docker stop mongodb
docker rm mongodb
```

7) Esecuzione del container con dentro l'applicazione web basata su nodejs, express e mongoose

Ora possiamo fare partire il container con l'applicazione web, usando il comando:

docker run -itd --rm --network interna --name nodejsapp -p 3000:3000 nodejsapp

Notare che attacco il container alla stessa rete virtuale **"interna"** del container di mongodb. Inoltre assegno nome "nodejsapp" al container che metto in esecuzione. Con il parametro -p 3000:3000 dico che voglio che la porta 3000 del container sia accedibile anche da fuori il container, in modo che i client esterni possano accedere al serve nodejs che lavora ed attende sulla porta 3000. Il parametro --rm significa che quando il container terminerà l'esecuzione, cioè verrà stoppato, docker dovrà eliminarlo automaticamente.

Per usufruire dell'applicazione, da dentro l'host fisico in cui eseguo il container, posso lanciare un browser grafico o testuale con cui mandare un messaggio HTTP POST con cui passo due parametri seq1 e seq2 e provoco così la generazione di 2 stringhe, as1 ed as2, ed il successivo salvataggio in mongodb della quadrupla in formato json {s1,s2,as1,as2} dove s1=seq1 ed s2=seq2.

Ad esempio, lanciando il comando:

curl --header "Content-Type: application/x-www-form-urlencoded" --request POST --data 'seq1=ALFABETAGAMMA&seq2=VAFFA' 0.0.0.0:3000

faccio generare due stringhe as1 ed as2 e viene restituito un documento json che è quello che è stato salvato nel database mongo.

{"s1":"ALFABETAGAMMA","s2":"VAFFAERIVAFFA","as1":"A-LF-ABETAGAMMA","as2":"VA-FFA-ERIVAFFA"}

Per stoppare il container userò il comando:

docker stop <nome container>

cioè, nel caso specifico:

docker stop nodejsapp

Infine, per eliminare il container, dopo averlo stoppato, userò il comando:

docker rm <nome container>

cioè, nel caso specifico:

docker rm nodejsapp

Poiché questo container non salva nessun dato, non ha senso fare un commit dopo averlo utilizzato.

8) Automatizzazione di costruzione, dispiegamento ed esecuzione di tutti i container, mediante Makefile.

Per automatizzare tutto quanto, è necessaria una modifica del codice dell'applicazione nodejsapp perché per riuscire a connettersi a mongodb occorre aspettare che mongodb sia stato inizializzato.

Inserisco perciò una orrenda pausa di 10 secondi prima di tentare la connessione da nodejsapp a mongodb. Aggiungo inoltre un po' di codice per visualizzare il tipo di errore. Agisco perciò sul file index.js

Tratto dal nuovo file **index.js**

```
// aspetto 10 sec che il container di mongo sia su
function pausecomp(millis)
{    var date = new Date();    var curDate = null;
    do { curDate = new Date(); } while(curDate-date < millis);
}
pausecomp(10000);

//connessione al db^M
mongoose.set('useFindAndModify', false);
mongoose.set('connectTimeoutMS', 30);
mongoose
.connect(
        'mongodb:// mongodb:27017/dbsa',
        { useNewUrlParser: true })
.then(() => console.log('MongoDB Connected'))
.catch((err) => console.log(err));
```

Ora possiamo procedere ad automatizzare creazione ed esecuzione dei container.

Il modo più meccanico per farlo consiste nello scrivere un Makefile che esegue tutti i comandi che abbiamo visto finora.
Nella directory SISTOP c'è questo Makefile.

```
SHELL=/bin/bash
MONGO_SERVICE_NAME=mongodb
NODEJSAPP_SERVICE_NAME=nodejsapp
MONGO_IMAGE=mymongo
NODEJSAPP_IMAGE=nodejsapp
NETWORK=interna
MONGO_PORTS_EXPOSED="27017-27019:27017-27019"
NODEJSAPP_PORTS_EXPOSED="3000:3000"
MONGODOCKERFILEDIR=./mongodb
NODEJSAPPDOCKERFILEDIR=./nodejs

all:	build up

.PHONY: build network up rmnetwork clean cleanall down downrmi stop rmi rm

build:
	ps ; cd ${MONGODOCKERFILEDIR} && docker build -t ${MONGO_IMAGE}  .
	cd ${NODEJSAPPDOCKERFILEDIR} && docker build -t ${NODEJSAPP_IMAGE} .

network:
	- docker network create -d bridge interna

up:	network
	docker run  -itd  --network interna  -p 27017-27019:27017-27019  --name mongodb  mymongo
	docker run -itd --rm --network interna --name nodejsapp -p 3000:3000 nodejsapp

cleanall: downrmi

clean:	down

downrmi: down  rmi

down:	stop rm rmnetwork

stop:
	- docker stop ${MONGO_SERVICE_NAME}  ${NODEJSAPP_SERVICE_NAME}

rm:
	- docker rm ${MONGO_SERVICE_NAME}  ${NODEJSAPP_SERVICE_NAME}

rmi:
	- docker rmi ${NODEJSAPPIMAGE} ${MONGOIMAGE}

rmnetwork:
	- docker network rm ${NETWORK}
```