Sistemi Cloud

Relazione del progetto:

TheSmilingChat

Docenti: prof. Pappalardo, prof. Fornaia

Studente: Luca Pappagallo

Indice generale

Introduzione

Questo progetto, denominato **TheSmilingChat**, è stato realizzato per il corso di Sistemi Cloud tenuto dal prof. Pappalardo e dal prof. Fornaia.
Ciò che si intende dimostrare è la padronanza di tecnologie quali Docker, del tool Docker Compose e dell'orchestratore integrato Swarm, oltre che di Kubernetes e della sua realizzazione all'interno del celebre servizio di cloud computing AWS.

Il progetto consiste in una Web Application che simula utenti che chattano scambiandosi smiles (faccine). Sono presenti alcuni pulsanti che permettono di aumentare e diminuire il numero di chat per emulare situazioni di sovraccarico del server, oltre alla possibilità di generare dei crash, opportunamente gestiti dai servizi di orchestrazione.

All'interno della Web Application si è scelto di separare gli aspetti di:

- **Frontend**: realizzato con VueJS ed il server di sviluppo Vite, che abilita funzioni quali l'hot reloading delle pagine quando è presente una modifica al codice sorgente ed altre di auto-minificazione durante il building del progetto in produzione per rendere la navigazione sul sito più rapida.

- **Backend**: realizzato con Flask, un microframework Python che permette di creare delle API o servire pagine web.

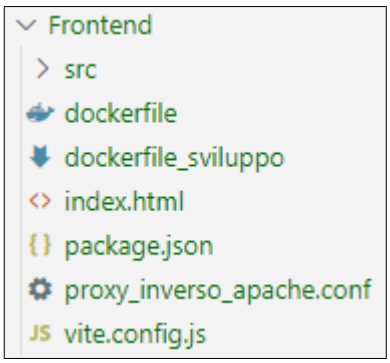
- **Database**: realizzato in MySQL.

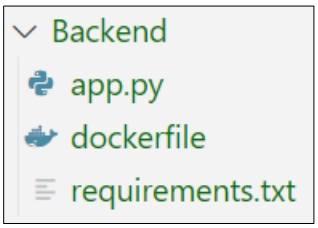
Si è quindi proceduto allo sviluppo, rendendo funzionante la Web Application seguendo una procedura che possiamo definire 'classica', ovvero eseguendo i vari servizi singolarmente su un PC e configurandoli manualmente.
Avendo preso nota dei vari passaggi necessari all'installazione e alla configurazione di tali strumenti, è stata eseguito il passaggio a Docker.

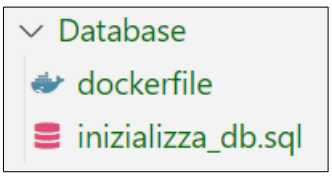
Docker

Docker permette di realizzare applicazioni containerizzate, che rendono più semplice per lo sviluppatore creare nuove applicazioni partendo da 'immagini' già pronte e semplificano allo strenuo la replicazione in ambienti differenti.
L'ambiente utilizzato per creare il progetto è Windows 11, per cui è stato scaricata l'adeguata versione di Docker Desktop dal sito ufficiale ed è stata avviata.

I tre componenti, Frontend, Backend e Database, sono stati inseriti all'interno di tre differenti container, facendo uso di tre dockerfile, contenuti nelle cartelle principali del progetto:







Di seguito vengono mostrati i tre dockerfile:

- Il dockerfile del **Frontend** utilizza la tecnica del *Multi-Stage Build*, una tecnica avanzata di Docker che consente di combinare più immagini, prendendo i dati necessari alla combinazione da un "*build-stage*" ed inserendoli nel "*final-stage*" (o "*production-stage*"). In questo particolare caso, è stata utilizzata l'immagine di nodejs per permettere la compilazione del progetto, il cui risultato fornisce una cartella /dist che verrà poi copiata all'interno della cartella di apache contenuta nell'immagine httpd.

  Si noti come durante il "*build-stage*" viene prima copiato il file package.json all'interno della directory di lavoro (/app) e poi l'intero contenuto della cartella locale (/Frontend). Questo può sembrare ridondante ma permette di sfruttare i meccanismi di caching dei layers di Docker.

  Nel "*production-stage*" viene inoltre modificata la porta 80 in 808, vengono abilitati i moduli che consentono il Reverse Proxy e viene trasferito un file locale con le regole da inserire nel httpd.conf di Apache (che vedremo nella parte relativa a Docker Swarm).

  Successivamente viene esposta all'esterno del container la porta 808 e viene avviato Apache in foreground (altrimenti l'esecuzione del container terminerebbe).

```
FROM node:14 as build-stage

WORKDIR /app

COPY package*.json ./

COPY . .

RUN npm install
RUN npm run build

FROM httpd:2 as production-stage

COPY --from=build-stage /app/dist /usr/local/apache2/htdocs/

RUN sed -i 's/Listen 80/Listen 808/' /usr/local/apache2/conf/httpd.conf

COPY ./proxy_inverso_apache.conf /usr/local/apache2/conf/extra/proxy_inverso_apache.conf

RUN sed -i 's/#LoadModule proxy_module/LoadModule proxy_module/' /usr/local/apache2/conf/httpd.conf && \
    sed -i 's/#LoadModule proxy_http_module/LoadModule proxy_http_module/' /usr/local/apache2/conf/httpd.conf

RUN echo "Include /usr/local/apache2/conf/extra/proxy_inverso_apache.conf" >> /usr/local/apache2/conf/httpd.conf

EXPOSE 808

CMD ["httpd", "-D", "FOREGROUND"]
```

  Vi è anche un "*dockerfile_sviluppo*" che lancia invece il server di ViteJS, utile per l'hot reloading in fase di sviluppo.

- Il Dockerfile del **Backend** utilizza un'immagine presente nel Docker Hub che possiede Python in versione 3.8.

```
FROM python:3.8

WORKDIR /app

COPY app.py .

COPY requirements.txt .

RUN pip install -r requirements.txt

EXPOSE 5000

CMD ["python", "app.py"]
```

  Viene impostata "app" come cartella di lavoro all'interno del container e qui vengono copiati i file locali "app.py" (che possiede il codice Flask) e "requirements.txt" (che possiede invece le dipendenze Python dell'environment).
  Le dipendenze vengono installate con il gestore dei pacchetti di Python (pip).

  Viene esposta la porta 5000 ed avviato il Server Web Flask.

- Il Dockerfile del **Database** utilizza un'immagine contenente l'ultima versione di MySQL.

```
FROM mysql:latest

RUN chown -R mysql:root /var/lib/mysql/

ENV MYSQL_DATABASE="progetto_sistemi_cloud"
ENV MYSQL_UTENTE="root"
ENV MYSQL_ROOT_PASSWORD="password"

COPY ./inizializza_db.sql /docker-entrypoint-initdb.d

EXPOSE 3306

CMD ["mysqld"]
```

La cartella mysql viene assegnata all'utente mysql:root. Vengono definite alcune variabili d'ambiente quali il nome del database, utente e password.
Si inserisce il file *inizializza_db.sql* nella cartella *docker-entrypoint-initdb.d*, il cui contenuto è automaticamente eseguito da Docker all'avvio del container, si espone la porta 3306 e si avvia il daemon di MySQL.

Docker Compose

Dopo aver definito i tre Dockerfile, si è scelto di utilizzare Docker Compose, tool integrato in Docker che permette la gestione di applicazioni multi-container. La sua configurazione è delegata ad un file YAML, solitamente denominato docker-compose.yml, che contiene le informazioni dei vari container da avviare.

Docker Compose permette anche di utilizzare reti specifiche per il dialogo tra i vari Container. Si è scelto quindi di creare una rete locale di tipo bridge chiamata *sistemi-cloud-net* con il comando:

```
docker network create -d bridge sistemi-cloud-net
```

L'elenco delle reti presenti, inclusa quella appena creata, è visibile con:

```
docker network ls
```

Questo è il contenuto del file **docker-compose.yml**:

```
services:
    miodb:
        container_name: miodb_c
        build:
            context: ./Database
        environment:
            MYSQL_ROOT_PASSWORD: password
        ports:
            - 3306:3306
        networks:
            - sistemi-cloud-net
    miofe:
        container_name: miofe_c
        build:
            context: ./Frontend
        ports:
            - 808:808
        networks:
            - sistemi-cloud-net
    miobe:
        container_name: miobe_c
        build:
            context: ./Backend
        ports:
            - 5000:5000
        networks:
            - sistemi-cloud-net

networks:
    sistemi-cloud-net:
        driver: bridge
```

In questo contesto si parla di 'services', e tra questi vi è quello relativo al container Docker del Database, quello relativo al Frontend e quello relativo al Backend.

La proprietà context, interna a build, specifica il path relativo dentro cui è presente il dockerfile.

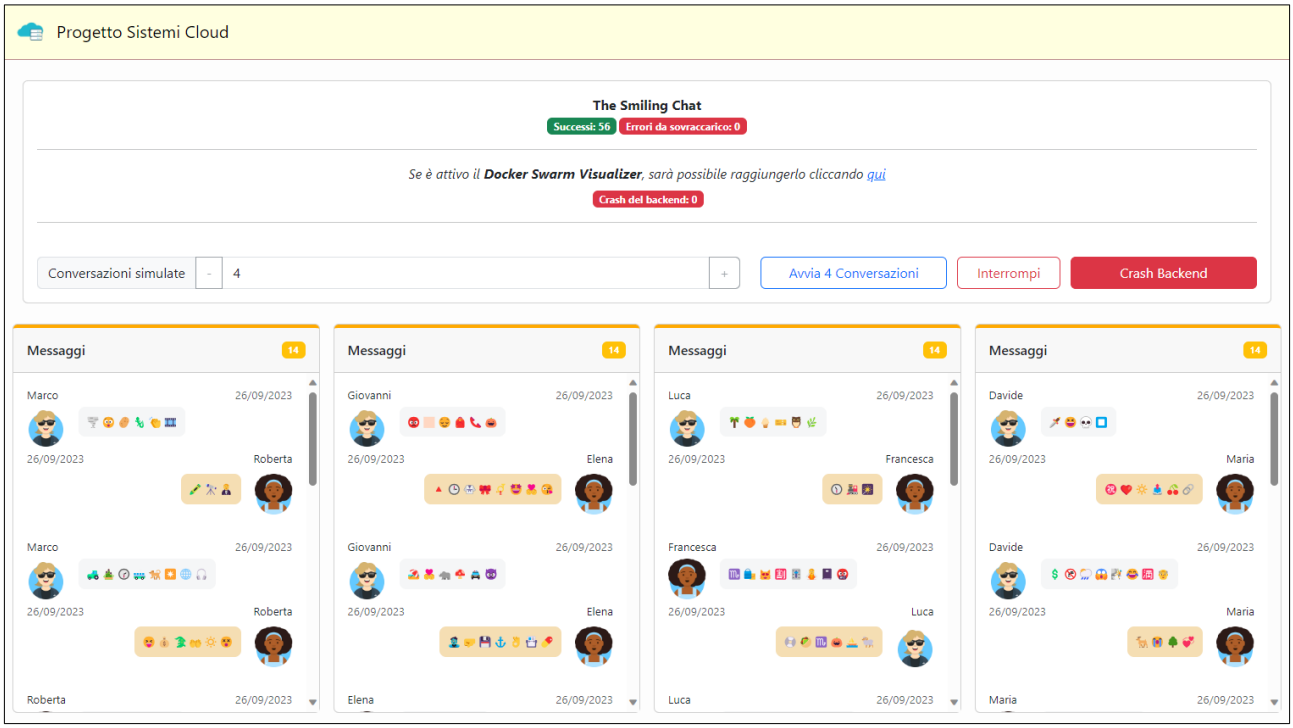
Ogni servizio specifica il nome del container, se vi sono variabili d'ambiente da definire o da sovrascrivere, la porta da aprire sull'host locale (quello su cui è presente l'engine Docker) che corrisponderà ad una porta aperta internamente al container, e la rete di appartenenza, dove si è scelto di utilizzare quella creata poc'anzi (*sistemi-cloud-net*).

Ogni servizio ha un nome, ed al suo avvio viene assegnato un indirizzo IP dinamico.
Docker ha al suo interno un servizio DNS che associa il nome del servizio all'indirizzo IP corrente, semplificando la comunicazione tra i vari servizi.
Grazie a questo meccanismo, il Backend può comunicare con il Database specificando come hostname MySQL direttamente il nome del servizio che contiene il Database (*miodb*) e non il suo IP, che cambierà ad ogni esecuzione.

A questo punto, posizionandosi all'interno della cartella root, in cui è presente il file YAML, è possibile avviare tutti i container con:

`docker compose up --build`
A questo punto sarà possibile visionare la Web Application da un browser su http://localhost:808

L'applicazione si mostra come di seguito:



Il selettore permette di scegliere da 0 a 30 conversazioni da avviare, dopo aver premuto il tasto "A*vvia X Conversazioni*" gli utenti, i cui dati sono memorizzati sul Database, inizieranno a scambiarsi messaggi contenenti smiles, ognuno dei quali effettuerà una richiesta al Backend, ovvero al server Flask. Va da sé, che un numero maggiore di conversazioni aumenta il carico e le risorse impiegate sul Server.
Il badge in alto in verde (*Successi: Y*) mostra il numero di richieste al server pervenute con successo, quello in rosso (*Errori da sovraccarico: Z*) mostra eventuali richieste non andate a buon fine.
Con il pulsante "Interrompi" è possibile fermare lo scambio di messaggi tra gli utenti, mentre con "Crash Backend" viene inviato un messaggio di kill ad uno dei processi che esegue il Backend. In Docker Compose non è presente alcun sistema di recovery e vi è una sola replica del Backend, quindi cliccando questo pulsante, il sistema smette di funzionare del tutto.

Aprendo un altro terminale, è possibile visionare i container attualmente in esecuzione e le relative porte disponibili attraverso il comando:

`docker ps`

```
CONTAINER ID   IMAGE                          COMMAND                  CREATED          STATUS          PORTS                                       NAMES
05a3911d0d9d   progetto_sistemi_cloud-miodb   "docker-entrypoint.s…"   50 seconds ago   Up 48 seconds   0.0.0.0:3306->3306/tcp, 33060/tcp   miodb_c
fbc04e5ac0f1   progetto_sistemi_cloud-miofe   "docker-entrypoint.s…"   50 seconds ago   Up 48 seconds   0.0.0.0:808->808/tcp                miofe_c
ec5f62114ef0   progetto_sistemi_cloud-miobe   "python app.py"          50 seconds ago   Up 48 seconds   0.0.0.0:5000->5000/tcp              miobe_c
```

Per interrompere l'esecuzione, premere in contemporanea i tasti CTRL e C sulla console in cui è stato eseguito l'up. Se si desidera anche eliminare i container creati si può utilizzare:

`docker compose down`

Docker Swarm

Docker Swarm è l'orchestratore interno a Docker. Si è scelto di utilizzarlo per testarne le capacità. A differenza di Docker Compose, Swarm è adatto ad ambienti di produzione, in cui è richiesta una distribuzione altamente scalabile e multi-host dell'applicativo, meccanismi di auto-balance e recovery da eventuali errori.

L'ambiente multi-host è stato simulato in locale utilizzando l'applicativo Multipass, che permette di creare e gestire con facilità macchine virtuali (VM) che dispongono di Ubuntu 22.04.

Multipass è stato scaricato per Windows dal sito ufficiale: https://multipass.run/install
Il sito di Multipass consiglia agli utenti Windows di rendere la propria connessione 'privata' dalle impostazioni di rete. Portata a termine l'installazione, è stata prima abilitata la possibilità di montare cartelle locali con il comando: `multipass set local.privileged-mounts=Yes`, poi si è scelto di utilizzare come hypervisor *virtualbox* (già presente sul proprio sistema) al posto di *hyper-v* con: `multipass set local.driver=virtualbox`.

In seguito sono state create due istanze di VM denominate 'leader' e 'worker1' con i seguenti comandi:

```
multipass launch -n leader -d 25G -m 1024M --network Wi-Fi
```

```
multipass launch -n worker1 -d 25G -m 1024M --network Wi-Fi
```

che crea delle VM con 25 GB di memoria su disco e 1 GB di Ram e associa ad essa l'interfaccia di rete che permetterà di comunicare tra loro. E' possibile vedere le interfacce di rete disponibili con: `multipass networks`.

A questo punto, montiamo la cartella locale contenente il progetto all'interno di una cartella interna alla VM 'leader':

```
multipass mount C:\progetto_sistemi_cloud leader:/home/ubuntu/sistemi_cloud
```

```
Name:           leader
State:          Running
IPv4:           172.22.115.179
                172.18.0.1
                172.17.0.1
Release:        Ubuntu 22.04.3 LTS
Image hash:     b3c31422c3b9 (Ubuntu 22.04 LTS)
CPU(s):         1
Load:           0.07 0.05 0.02
Disk usage:     7.1GiB out of 19.3GiB
Memory usage:   689.7MiB out of 892.2MiB
Mounts:         C:/progetto_sistemi_cloud => /home/ubuntu/sistemi_cloud
                    UID map: -2:default
                    GID map: -2:default

Name:           worker1
State:          Running
IPv4:           172.22.118.49
                172.18.0.1
                172.17.0.1
Release:        Ubuntu 22.04.3 LTS
Image hash:     b3c31422c3b9 (Ubuntu 22.04 LTS)
CPU(s):         1
Load:           0.07 0.02 0.00
Disk usage:     5.3GiB out of 19.3GiB
Memory usage:   280.5MiB out of 892.2MiB
Mounts:         --
```

Possiamo quindi vedere l'elenco di VM create ed altre informazioni utili con:

```
multipass info --all
```

Come mostrato nell'immagine a sinistra, è possibile vedere informazioni quali il quantitativo di risorse a disposizione della VM, l'indirizzo IP, eventuali cartelle montate e lo stato delle macchine (running).

Qualora si decidesse di interrompere l'esecuzione di una VM, è possibile farlo con:

```
multipass stop NOME_VM
```

e si potrà procedere alla sua eliminazione definitiva con `multipass delete NOME_VM` e `multipass purge`.

Installiamo quindi Docker sulle due VM:

```
multipass exec leader -- curl -fsSL https://get.docker.com -o get-docker.sh
multipass exec leader -- sh get-docker.sh
```

```
multipass exec worker1 -- curl -fsSL https://get.docker.com -o get-docker.sh
multipass exec worker1 -- sh get-docker.sh
```

A questo punto l'ambiente multi-host simulato è pronto a divenire un cluster Docker Swarm.

Il file *docker-compose.yml* creato in precedenza però non è adatto all'utilizzo con Swarm, dunque è stato creato un nuovo file denominato *docker-compose-production.yml* che necessita di alcune modifiche rispetto al precedente.

Sarà infatti necessaria una nuova rete di tipo '*overlay*' che definiremo a breve e, inoltre, Swarm non si aspetta più i dockerfile ma le immagini risultanti, esse sono quindi state pacchettizzate e caricate su Docker Hub come spiegato di seguito:

Per caricare le immagini su Docker Hub

Creare un account su https://hub.docker.com, e creare un repository per ogni container, assegnandogli un nome ed una descrizione e successivamente:

Effettuare il login a Docker Hub da terminale:

`docker login`

dopo di ché, entrati nella cartella del Backend, è stata creata l'immagine con il comando:

`docker build -t luca4k4/progetto_sistemi_cloud_backend:v1 .`

a cui è possibile assegnare un Tag, ad esempio v1-release, con:

`docker tag luca4k4/progetto_sistemi_cloud_backend:v1 luca4k4/progetto_sistemi_cloud_backend:v1-release`

A questo punto si può pubblicare l'immagine su docker hub:

`docker push luca4k4/progetto_sistemi_cloud_backend:v1-release`

La stessa operazione è fatta per il Frontend e per il Database.

Il file *docker-compose-production.yml* si presenterà così:

```yaml
services:
    miodb:
        image: luca4k4/progetto_sistemi_cloud_db:v1-release
        environment:
            MYSQL_ROOT_PASSWORD: password
        ports:
            - 3306:3306
        networks:
            - sistemicloud-net
    miofe:
        image: luca4k4/progetto_sistemi_cloud_frontend:v1-release
        ports:
            - 808:808
        networks:
            - sistemicloud-net
        depends_on:
            - miobe
    miobe:
        image: luca4k4/progetto_sistemi_cloud_backend:v1-release
        ports:
            - 5000:5000
        networks:
            - sistemicloud-net
        depends_on:
            - miodb
        deploy:
            replicas: 4

networks:
    sistemicloud-net:
        external: true
```

I tre servizi scaricano l'immagine presente su Docker Hub, e partecipano alla rete "sistemicloud-net".

Il servizio miobe specifica che desidera avere 4 repliche sempre attive, quindi qualora una di queste smettesse di funzionare, Swarm dovrà occuparsi di rigenerarla.

Il resto è molto simile al file docker-compose.yml visto in precedenza.

A questo punto abbiamo tutto per tirare su il cluster Swarm.

Inizializziamo la VM "leader" come Manager (o Leader) Swarm:

`multipass exec leader -- sudo docker swarm init`

Qualora si manifesti quest'errore:

```
C:\progetto_sistemi_cloud>multipass exec leader -- sudo docker swarm init
Error response from daemon: could not choose an IP address to advertise since this system has
multiple addresses on different interfaces (10.0.2.15 on enp0s3 and 192.168.51.109 on enp0s8)
- specify one with --advertise-addr
```

è sufficiente specificare una delle due interfacce aggiungendo un parametro:
`multipass exec leader -- sudo docker swarm init --advertise-addr IP_INTERFACCIA_SCELTA`

Questo genererà un comando docker da eseguire invece sul worker:

```
C:\progetto_sistemi_cloud>multipass exec leader -- sudo docker swarm init
Swarm initialized: current node (k3ulr573k3ssghu88eip12bzt) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-5wvcidus1m17o127a3emwjimwtblg5iokb05s8n20c05sgygz5-b39y2wfvp7f90wpynabvjjyej 192.168.145.246:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

Prima però, come accennato è necessario creare una rete che permetta di comunicare anche attraverso Docker Engine multipli, questa tipologia di rete è denominata '*overlay*', e differisce da quella 'bridge' che è invece isolata ad un unico Docker Engine.
La rete *overlay*, abilitata quindi a funzionare su Swarm, è stata denominata '*sistemicloud-net*' ed è creata sulla VM leader con il comando:

multipass exec leader -- sudo docker network create -d overlay sistemicloud-net

Quindi eseguiamo:

multipass exec worker1 -- sudo COMANDO_OTTENUTO_PRECEDENTEMENTE

Se tutto va a buon fine otterremo:

```
C:\progetto_sistemi_cloud>multipass exec worker1 -- sudo docker swarm join --token SWMTKN-1-2e5d1rv0wqpwwq2f5j3ulah9rde0qrvi7sjqlotgm51udsyioc-50o3jof6uc52649sc2cgqpq88 192.168.51.145:2377
This node joined a swarm as a worker.
```

A questo punto il nostro cluster Swarm è attivo ed in funzione, e possiamo vederne lo stato con:

multipass exec leader -- sudo docker node list

```
C:\progetto_sistemi_cloud>multipass exec leader -- sudo docker node list
ID                            HOSTNAME   STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
nq5vgtw1b59g19l5dow38d4mk *   leader     Ready     Active         Leader           24.0.6
lfehlf1qsuyrm7uc2oflrx04f     worker1    Ready     Active                          24.0.6
```

Ricordiamo che sul leader è stata montata la cartella contenente il progetto (*/home/ubuntu/sistemi_cloud*), e che ha nella root anche il file *docker-compose-production.yml*, quindi possiamo eseguire il deploy su tutti i nodi del cluster con:

multipass exec leader -- sudo docker stack deploy -c /home/ubuntu/sistemi_cloud/docker-compose-production.yml progetto_sistemi_cloud

Se si desidera, si può vedere lo stato dei servizi avviati sulle due VM con:

multipass exec leader -- sudo docker node ps

```
ID            NAME                          IMAGE                                                NODE     DESIRED STATE   CURRENT STATE
tynnhv12q7ib  progetto_sistemi_cloud_miobe.3  luca4k4/progetto_sistemi_cloud_backend:v1-release    leader   Running         Running about an hour ago
oswhp7jxw6ug  progetto_sistemi_cloud_miobe.4  luca4k4/progetto_sistemi_cloud_backend:v1-release    leader   Running         Running 27 minutes ago
v9416htfvv87  progetto_sistemi_cloud_miodb.1  luca4k4/progetto_sistemi_cloud_db:v1-release         leader   Running         Running about an hour ago
```

multipass exec leader -- sudo docker node ps worker1

```
ID            NAME                          IMAGE                                                NODE     DESIRED STATE   CURRENT STATE
x21uhek19uys  progetto_sistemi_cloud_miobe.1  luca4k4/progetto_sistemi_cloud_backend:v1-release    worker1  Running         Running about an hour ago
pidda0tyq7yf  progetto_sistemi_cloud_miobe.2  luca4k4/progetto_sistemi_cloud_backend:v1-release    worker1  Running         Running about an hour ago
w67i4fqpxvjt  progetto_sistemi_cloud_miofe.1  luca4k4/progetto_sistemi_cloud_frontend:v1-release   worker1  Running         Running 2 hours ago
```

Noteremo Frontend e Database lanciati un'unica volta e il Backend avviato invece ben 4 volte, come specificato nel file di configurazione.
E' interessante vedere come sia Swarm stesso a decidere se piazzare i servizi su un nodo piuttosto che su un altro.

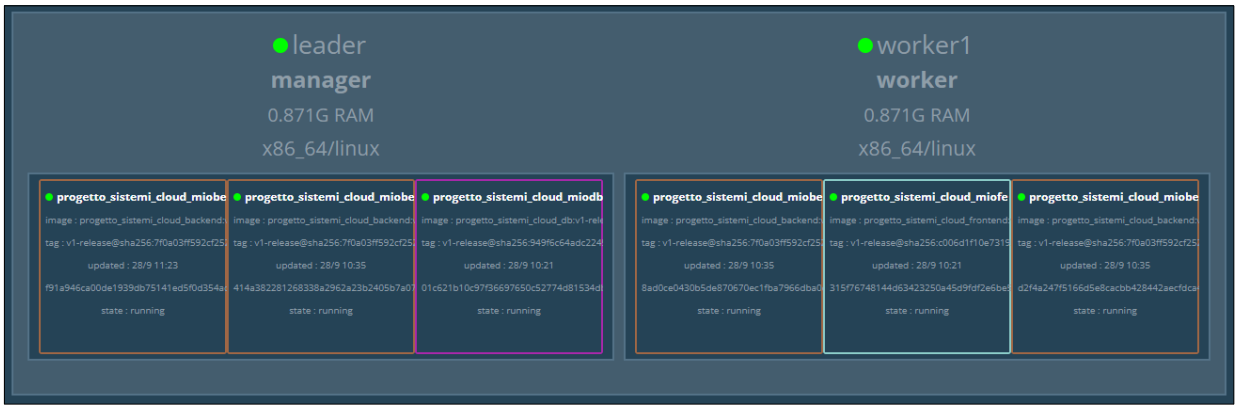
Dopo che il deploy è completo e lo stato dei container passerà a Running (il ché potrebbe richiedere anche 10-15 minuti), il progetto sarà visionabile all'url: http://IP_VM_LEADER:808/
Si ricordi che l'IP delle VM si può ottenere da terminale con: multipass info –all o multipass list.

E' disponibile anche un tool chiamato **Docker Swarm Visualizer**, che permette di avere una GUI per visualizzare le VM ed i relativi servizi avviati.

Per avviarlo, si esegue il comando:

multipass exec leader -- sudo docker run -it -d -p 8080:8080 -v /var/run/docker.sock:/var/run/docker.sock dockersamples/visualizer

Successivamente si potrà utilizzare l'apposito link contenuto nella pagina web del progetto:



Qualora si desiderasse di eliminare il progetto presente nel cluster, si può utilizzare il comando:

multipass exec leader -- sudo docker stack rm progetto_sistemi_cloud

Se invece le VM vengono interrotte e poi riavviate, l'indirizzo IP di queste cambierà e dunque la rete Swarm non funzionerà più. Sarà in tal caso necessario effettuare il leaving dalla rete swarm in entrambe le VM con:

multipass exec leader -- sudo docker swarm leave --force

multipass exec worker1 -- sudo docker swarm leave

e reinizializzarla come visto in precedente.

Come accennato nella parte relativa ai vari Dockerfile, nel Frontend è stato inserito un Reverse Proxy. Infatti in Swarm e Kubernetes, dal momento che non si eseguono più i vari servizi in localhost, per contattare il Backend è necessario che il client conosca il suo indirizzo. Tuttavia il client (ovvero il browser dell'utente che visita il sito), è esterno alla rete Swarm e non ha accesso al DNS interno a Docker che permette di reindirizzare correttamente la richiesta al Backend.

Quindi, attraverso il dockerfile, sono state aggiunge le seguenti regole proxy ad Apache in esecuzione sul Frontend:

```
<Location "/backend/ping">
    ProxyPass "http://miobe:5000/ping"
</Location>

<Location "/backend/get_utenti">
    ProxyPass "http://miobe:5000/get_utenti"
</Location>

<Location "/backend/esegui_operazione">
    ProxyPass "http://miobe:5000/esegui_operazione"
</Location>

<Location "/backend/crash">
    ProxyPass "http://miobe:5000/crash"
</Location>
```

Qualora si utilizzi invece il *dockerfile_sviluppo*, verrà avviato il server Vite, utile in modalità sviluppo per la funzione di hot reloading, ma si dovrà decommentare la configurazione del proxy contenuta nel file *vite.config.js*.

Infine, si accenna al fatto che, Docker Swarm non ha funzioni di auto scaling integrate al momento, ma sono disponibili altre soluzioni come *docker-machine* da allegare come ulteriore worker e si occuperà di inserire o rimuovere nodi in base a dei trigger (ad esempio ad un utilizzo esiguo o intenso di CPU e RAM).

Possiamo procedere a interrompere le VM create con:

multipass stop leader
multipass stop worker1

Kubernetes (Microk8s)

Dopo aver testato le funzionalità e aver capito il funzionamento di Docker Swarm, si è passati ad una delle varie realizzazioni di Kubernetes (k8s), e si è scelta quella creata da Canonical (la stessa azienda che ha realizzato la distro linux Ubuntu) denominata Microk8s, che si occupa di eseguire un cluster k8s a singolo nodo.
E' possibile installarlo in due modi: tramite l'installer per Windows scaricabile dal sito ufficiale (https://microk8s.io/#install-microk8s), oppure, dal momento che abbiamo precedentemente installato Multipass, possiamo creare una macchina virtuale Ubuntu e installarlo tramite gestore dei pacchetti. Si è scelto di seguire quest'ultima strada. Quindi è stata creata una macchina virtuale con 4 GB Ram e 25 GB di disco fisso denominata k8s:

```
multipass launch -n k8s -d 25G -m 4096M --network Wi-Fi
```

su cui è stata montata la cartella contenente il progetto, in maniera simile a quanto visto per Docker Swarm:

```
multipass mount C:\progetto_sistemi_cloud k8s:/home/ubuntu/sistemi_cloud
```

Accediamo alla nuova macchina virtuale:

```
multipass shell k8s
```

e installiamo microk8s:

```
sudo snap install microk8s --classic
```

altrimenti avviamolo manualmente con:

```
sudo microk8s start
```

assicuriamoci che microk8s stia stato avviato con:

```
sudo microk8s status
```

A questo punto abilitiamo alcuni moduli:

```
sudo microk8s enable dashboard dns registry istio
```

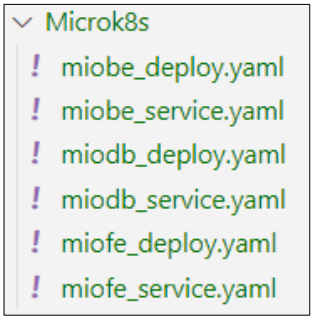
Per vedere i vedere i *deploy*, i *service*, ecc. in funzione, oppure semplicemente per controllare che il client *kubectl*, utilizzato per inviare comandi al nostro cluster, risponda correttamente, possiamo utilizzare:

```
sudo microk8s kubectl get all --all-namespaces
```

Applichiamo le configurazioni contenute nell'apposita cartella /Microk8s:

```
sudo microk8s kubectl apply -f /home/ubuntu/sistemi_cloud/Microk8s/
```

I file contenuti nella cartella "Microk8s" sono i seguenti:



Questa conterrà sia i *deploy*, che utilizzano le immagini Docker caricate in precedenza su Docker Hub e che impiegano etichette per identificare i Pod, sia i *service* che permettono agli stessi di essere fruibili esternamente, e che vengono associate ai relativi deploy proprio tramite le etichette.

A questo punto è necessario attendere che i vari Pods passino in stato *Ready/Running*. Possiamo controllare lo stato con: `sudo microk8s kubectl get pods`
Una volta che essi saranno attivi, possiamo esternalizzare il Frontend affinché sia visibile dal nostro browser:

```
sudo microk8s kubectl port-forward -n default service/miofe 808:808 --address='0.0.0.0'
```

Da notare che questo comando è bloccante per il terminale che lo esegue.

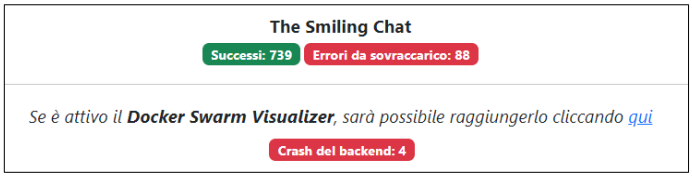
Dunque possiamo aprire il browser su http://IP_MACCHINA_VIRTUALE:808/ (l'IP è ottenibile eseguendo in un altro terminale: `multipass list`) e vedremo l'applicazione in esecuzione.

Utilizzando lo stesso comando di prima (`sudo microk8s kubectl get pods`) è possibile notare cosa accade ai Pods quando viene meno uno dei Backend:

```
ubuntu@k8s:~$ sudo microk8s kubectl get pods
NAME                    READY   STATUS      RESTARTS       AGE
miodb-6cdd968d6-6qss2   1/1     Running     1 (129m ago)   139m
miofe-5fc96b5c7b-r6w9l  1/1     Running     1 (129m ago)   139m
miobe-784c7c44f5-55fph  1/1     Running     2 (35m ago)    139m
miobe-784c7c44f5-tprzp  1/1     Running     2 (34m ago)    139m
miobe-784c7c44f5-cpw62  0/1     Completed   2 (34m ago)    139m
miobe-784c7c44f5-8j6zx  0/1     Completed   2 (34m ago)    139m
```

Gli ultimi due Pods, corrispondenti a due istanze del Backend, sono terminati a causa del click sul pulsante "Crash Backend" presente nell'applicativo. Questi due vengono, in pochissimo tempo, rigenerati, mantenendo un downtime minimo.

Cliccare più volte sul pulsante appena citato mentre gli utenti simulati dall'applicativo si scambiano messaggi, genererà per alcuni momenti degli errori alle richieste eseguite al Backend, fin tanto che k8s non rigenererà i Pods, successivamente le richieste verranno di nuovo gestite correttamente.

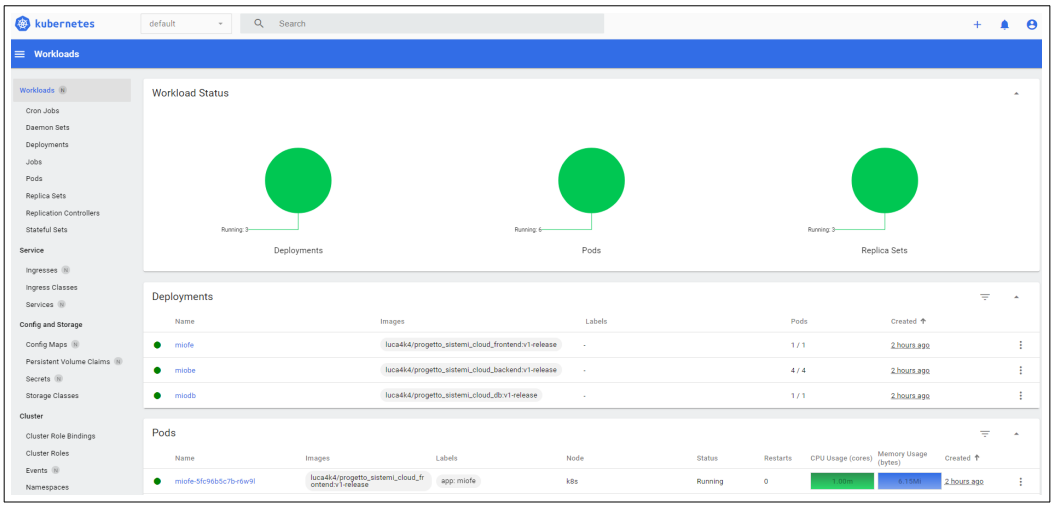


Tra i moduli di Microk8s abilitati in precedenza, è presente la Dashboard, che mette a disposizione una GUI che mostra tutto ciò che riguarda il cluster.

Possiamo eseguire il comando bloccante:

```
sudo microk8s dashboard-proxy
```

questo farà il forward della porta locale 443 sulla 10443 e genererà un token da inserire all'indirizzo http://IP_MACCHINA_VIRTUALE:10443/, che apparirà così:



Anche qui sarà possibile notare quando uno dei Pod del nodo verrà meno ed il numero di repliche create. Inoltre sono disponibili anche informazioni quali le risorse utilizzate da ogni Pod.
A tal proposito ricordiamo che è presente un Load Balancer interno a k8s che distribuisce le richieste ricevute a più Pod, in modo da distribuire meglio l'utilizzo delle risorse.

Quando la nostra applicazione non è più utile, possiamo rimuoverla con:

```
sudo microk8s kubectl delete -f /home/ubuntu/sistemi_cloud/Microk8s/
```

e ricordiamo di interrompere l'esecuzione di microk8s (`sudo microk8s stop`), uscire dalla shell della VM (`exit`) e fermare la macchina virtuale (`multipass stop k8s`)

Amazon AWS (Elastic Kubernetes Service)

Dopo aver testato il cluster a singolo nodo in locale con Microk8s, si è scelto di provare il servizio di k8s gestito offerto da Amazon AWS, denominato Elastic Kubernetes Service (EKS).
Si è scelto di operare all'interno della regione geografica "*eu-west-3*", localizzata a Parigi. Inoltre, come suggerito da AWS, è stato creato un utente IAM (diverso dall'utente root) con cui operare. Questo è stato aggiunto ad un gruppo denominato "*sistemi_cloud*" a cui sono state assegnate le Policy di autorizzazione necessarie ad EKS:

AWS Service	Access Level
CloudFormation	Full Access
EC2	**Full:** Tagging **Limited:** List, Read, Write
EC2 Auto Scaling	**Limited:** List, Write
EKS	Full Access
IAM	**Limited:** List, Read, Write, Permissions Management
Systems Manager	**Limited:** List, Read

Infatti, si è scelto di operare con la CLI "eksctl" al posto di utilizzare la GUI interna ad AWS, creando quindi il cluster direttamente da riga di comando e non dall'interno dell'interfaccia grafica di AWS, in quest'ultimo caso sarebbe stata sufficiente la policy "*AmazonEKSClusterPolicy*".

Come suggerito nella documentazione ufficiale, sono stati scaricate e installate le 3 CLI: **aws**, **eksctl** e **kubectl**.

La prima è stata utilizzata per configurare l'utente IAM creato, utilizzando il comando: `aws configure` e inserendo le informazioni della chiave di accesso creata dalla GUI di AWS. Questa inserirà le credenziali di accesso in un file localizzato in : *C:\Users\[NOME_UTENTE]\.aws\config.*

Sarà quindi possibile testare l'accesso se l'accesso è eseguito correttamente con:

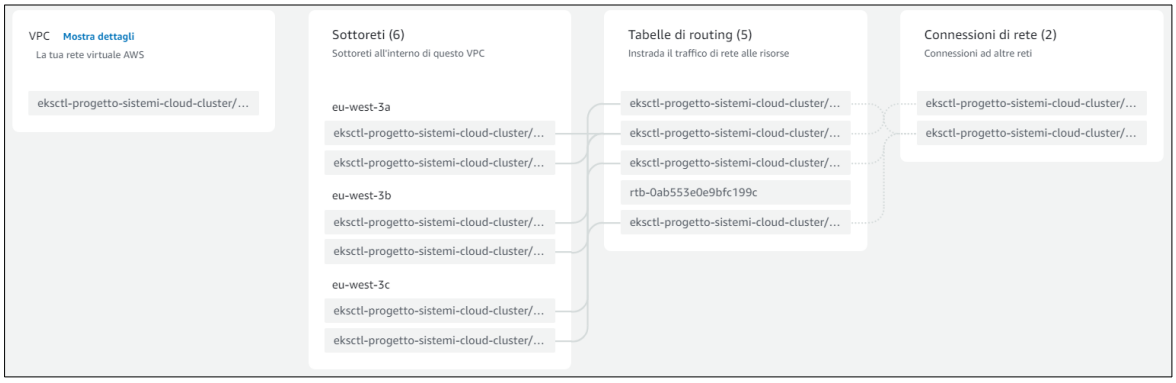
```
aws sts get-caller-identity
```

In seguito è stato utilizzata la CLI di EKS (eksctl) per creare il cluster, utilizzando eksctl.exe (scaricabile da https://eksctl.io/installation/#direct-download-latest-release-amd64x86_64-armv6-armv7-arm64) dentro la root del progetto e lanciando il seguente comando (che impiega circa 15-20 minuti):

```
.\eksctl create cluster --version=1.28 --name progetto-sistemi-cloud --region eu-west-3
```

Questo creerà un cluster Kubernetes versione 1.28, denominato *progetto-sistemi-cloud*, localizzato in *eu-west-3*, e di default verranno associati ad esso due istanze EC2 di tipo *m5.large*, con 2 cpu virtuali e 8 GB Ram. Queste ultime sono fuori dal tier di prova di AWS ed hanno attualmente un costo di circa 10 centesimi/ora nella regione geografica selezionata.

Viene inoltre creata una VPC e delle sottoreti che hanno questa struttura:



La VPC avrà di default 3 Availability Zone (AZ), che avranno a loro volta due sotto-reti ciascuna. Queste saranno inserite in tabelle di routing, che renderanno possibile accedere al cluster dall'esterno attraverso un Internet Gateway. L'accesso dall'esterno è gestito da un Load Balancer legato al Pod contenente il Frontend, che attualmente, essendo in singola replica, punterà sempre alla stessa macchina.

Da notare come, queste configurazioni avviate da eksctl, vengano eseguite in uno stack da un altro servizio di AWS chiamato CloudFormation (ecco perché l'utente IAM creato dovrà avere *Full Access* a questo servizio).

Come detto, di default verranno create due istanze *m5.large*, in un primo momento si è fatto un tentativo impostando delle istanze presenti nel *free tier* di AWS (*t3.micro*), tuttavia 4 dei 6 Pods non andavano mai oltre lo stato di Pending, come se le risorse di queste istanze non fossero sufficienti ad avviarli tutti.

Avviato il cluster, possiamo utilizzare kubectl per distribuire l'applicazione. Dalla root del progetto diamo dunque:

```
kubectl apply -f .\Microk8s\
```

Possiamo infatti utilizzare le stesse configurazioni Kubernetes già utilizzate per Microk8s. Controlliamo lo stato di avvio dei Pods con:

```
kubectl get all -o wide
```

Qui noteremo che il tipo del servizio del Frontend (che ricordiamo essere denominato "*miofe*") è LoadBalancer ed ha di conseguenza un IP esterno che possiamo utilizzare, sulla porta 808, per raggiungere l'applicazione.

```
C:\Users\luca4\Desktop\provapp>kubectl get all -o wide
NAME                          READY   STATUS    RESTARTS       AGE     IP               NODE                                              NOMINATED NODE   READINESS GATES
pod/miobe-6c8f64579c-5v7c8    1/1     Running   4 (4m9s ago)   8m56s   192.168.52.175   ip-192-168-35-187.eu-west-3.compute.internal   <none>           <none>
pod/miobe-6c8f64579c-8gcsf    1/1     Running   3 (4m8s ago)   8m56s   192.168.55.90    ip-192-168-35-187.eu-west-3.compute.internal   <none>           <none>
pod/miobe-6c8f64579c-8vrwt    1/1     Running   3 (3m48s ago)  8m55s   192.168.28.81    ip-192-168-17-109.eu-west-3.compute.internal   <none>           <none>
pod/miobe-6c8f64579c-vpp7g    1/1     Running   3 (3m49s ago)  8m56s   192.168.21.208   ip-192-168-17-109.eu-west-3.compute.internal   <none>           <none>
pod/miodb-7b7c458856-h9bwj    1/1     Running   0              8m55s   192.168.60.12    ip-192-168-35-187.eu-west-3.compute.internal   <none>           <none>
pod/miofe-6d6b5d467d-vbv7w    1/1     Running   0              8m55s   192.168.53.250   ip-192-168-35-187.eu-west-3.compute.internal   <none>           <none>

NAME                 TYPE           CLUSTER-IP       EXTERNAL-IP                                                                PORT(S)          AGE     SELECTOR
service/kubernetes   ClusterIP      10.100.0.1       <none>                                                                     443/TCP          22m     <none>
service/miobe        ClusterIP      10.100.58.166    <none>                                                                     5000/TCP         8m56s   app=miobe
service/miodb        NodePort       10.100.185.206   <none>                                                                     3306:30301/TCP   8m55s   app=miodb
service/miofe        LoadBalancer   10.100.173.162   a9bbb48a7665247da9bdabe218ff0223-1968172930.eu-west-3.elb.amazonaws.com   808:30280/TCP    8m55s   app=miofe

NAME                     READY   UP-TO-DATE   AVAILABLE   AGE     CONTAINERS   IMAGES                                                  SELECTOR
deployment.apps/miobe    4/4     4            4           8m56s   miobe        luca4k4/progetto_sistemi_cloud_backend:v1-release       app=miobe
deployment.apps/miodb    1/1     1            1           8m55s   miodb        luca4k4/progetto_sistemi_cloud_db:v1-release            app=miodb
deployment.apps/miofe    1/1     1            1           8m55s   miofe        luca4k4/progetto_sistemi_cloud_frontend:v1-release      app=miofe

NAME                                DESIRED   CURRENT   READY   AGE     CONTAINERS   IMAGES                                                  SELECTOR
replicaset.apps/miobe-6c8f64579c    4         4         4       8m57s   miobe        luca4k4/progetto_sistemi_cloud_backend:v1-release       app=miobe,pod-template-hash=6c8f64579c
replicaset.apps/miodb-7b7c458856    1         1         1       8m56s   miodb        luca4k4/progetto_sistemi_cloud_db:v1-release            app=miodb,pod-template-hash=7b7c458856
replicaset.apps/miofe-6d6b5d467d    1         1         1       8m56s   miofe        luca4k4/progetto_sistemi_cloud_frontend:v1-release      app=miofe,pod-template-hash=6d6b5d467d
```

Dunque quando i Pod saranno in stato di Ready, possiamo raggiungere l'applicativo che funzionerà come già visto nella sezione relativa a Microk8s.

Possiamo dunque rimuovere l'applicazione con `kubectl delete -f .\Microk8s\` e distruggere il cluster con:

```
.\eksctl delete cluster progetto-sistemi-cloud
```

Alternativa a creare le istanze di EC2 è un nuovo servizio AWS denominato Fargate, che permette di eseguire container serverless con risorse che vengono allocate dinamicamente quando necessario. Su Fargate però non si può utilizzare il classico LoadBalancer degli EC2 bensì è necessario configurare un ALB Ingress. Fargate ha un costo di 0.05$/ora per ogni vCPU e di 0.005$/ora per GB di RAM.