

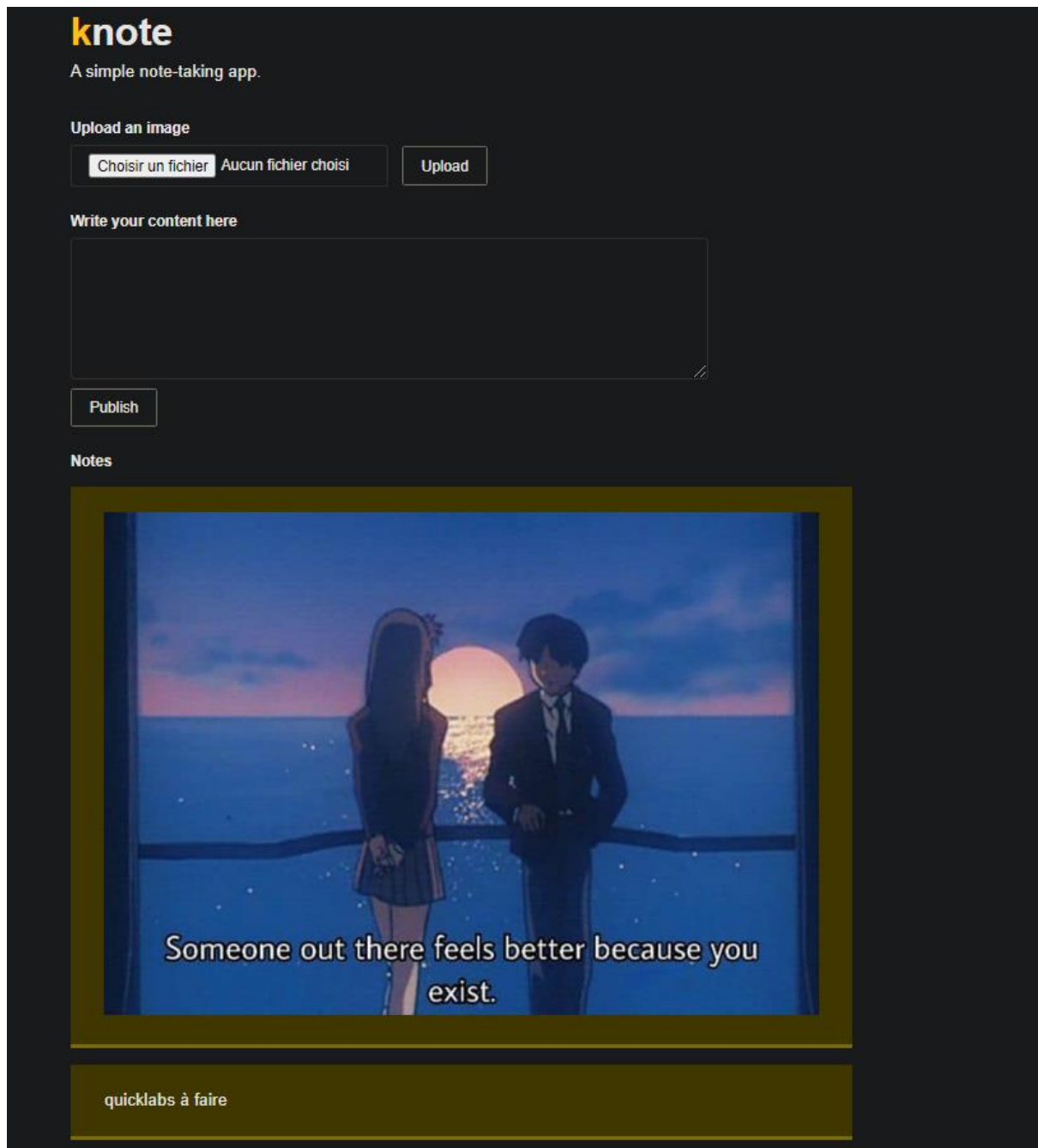
Rapport de projet

PROGRAMMATION WEB
CHETTAB ABDELKADER

1. Introduction et résumé :

Le but de ce projet est de développer une application avec des microservices en utilisant Spring boot et Kubernetes.

L'application permettra de prendre des notes en uploadant aussi des photos pouvant accompagner la prise de notes.



2. Bootstrap de l'application :

Nous allons construire le squelette de l'application en utilisant Spring initialize

<https://start.spring.io/>

Vu que c'est une application de note développée en Java en utilisant Spring, on la nommera Knote-java en choisissant les paramètres suivants :

- Projet: Maven Project
- Langage: Java
- Spring boot: 2.6.3
- Packaging: Jar
- Java: 11 (car c'était la version de mon sdk)

Il faut ensuite ajouter les dépendances suivantes :

- **Spring web** : créer des applications Web, y compris RESTful, à l'aide de Spring MVC. Utilise Apache Tomcat comme conteneur intégré par défaut.
- **Spring boot actuator** : permet de surveiller et de gérer votre application - comme la santé de l'application, les métriques, les sessions, etc.
- **Apache freemarker** : bibliothèque Java pour générer une sortie texte (pages Web HTML, e-mails, fichiers de configuration, code source, etc.). Elle est basée sur des modèles.
- **Spring Data MongoDB** : driver et implémentation pour que les interfaces Spring Data fonctionnent avec MongoDB.
- **Lombok** : bibliothèque d'annotations Java qui aide à réduire le code boiler plat.

Front end :

Nous pouvons finalement générer le projet et obtenir le squelette de notre application.

Nous devons ensuite coder la partie front-end en créant 2 fichiers qui doivent être placés dans les répertoire suivant :

src/main/resources/static/ et src/main/resources/templates/

Le coeur de notre application sera construit dans le fichier :

src/main/java/io/learn k8s/knote-java/Knot Java Application.java

3. Connecter la base de donnée :

Nous utiliserons MongoDB comme base de données car elle est simple à implémenter et à utiliser. Il suffit d'entrer l'url de notre database dans le fichier

src/main/resources/application.properties

L'url étant :

spring.data.mongodb.uri=mongodb://localhost:27017/dev

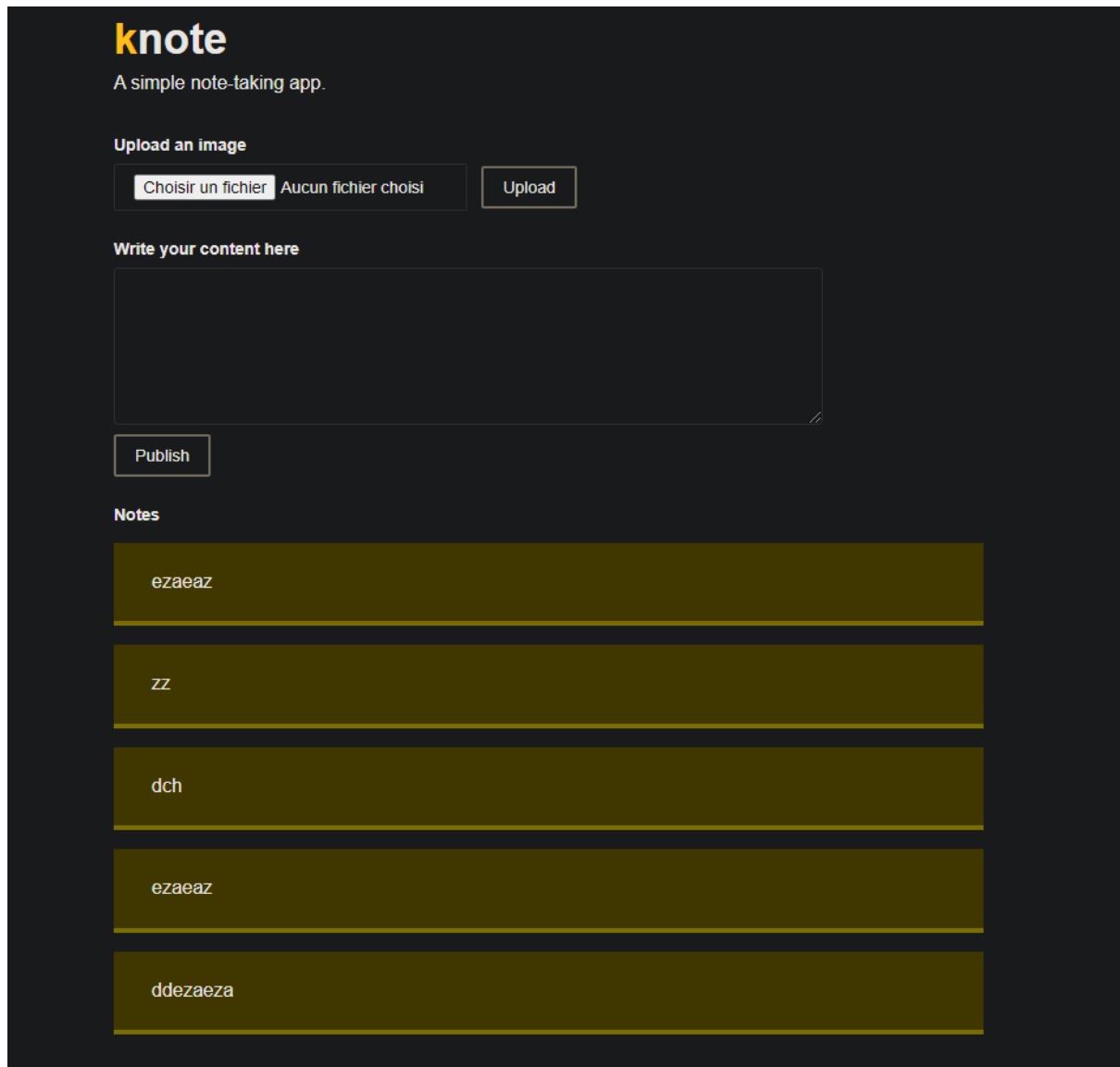
Une fois notre application de base terminée, on peut l'exécuter via la commande :

mvn clean install spring-boot:run

qui se connectera à la base de données MongoDB qu'on a démarré précédemment. On devrait pouvoir accéder à l'application via :

<http://localhost:8080/>

Un premier aperçu de l'application :



4. Conteneurisé l'application:

1) Créer l'image :

Nous créons le DockerFile permettant de mettre notre application sous forme d'une image.

```
FROM adoptopenjdk/openjdk11:jdk-11.0.2.9-slim
WORKDIR /opt
ENV PORT 8080
EXPOSE 8080
COPY target/*.jar /opt/app.jar
ENTRYPOINT exec java $JAVA_OPTS -jar app.jar
RUN echo hello
```

Nous allons l'enregistrer en tant que Dockerfile dans le répertoire racine de votre application. Le Dockerfile ci-dessus inclut les commandes suivantes :

- FROM définit la couche de base du conteneur, dans ce cas, une version d'OpenJDK 11
- WORKDIR définit le répertoire de travail sur /opt/. Chaque instruction suivante s'exécute à partir de ce dossier.
- ENV est utilisé pour définir une variable d'environnement.
- COPY copie les fichiers jar de /target/ dans le répertoire /opt/ à l'intérieur du conteneur.
- ENTRYPOINT exécute java \$JAVA_OPTS -jar app.jar à l'intérieur du conteneur.

Le Run echo hello me permet juste de pouvoir déboguer en cas de problème et suivre l'exécution sur console.

Nous pouvons donc maintenant construire notre image avec la commande :

docker build -t knote-java

Nous constatons que notre nouvelle image a été ajoutée :

```
C:\Users\Redak\Desktop\prog avancée\knote-java-01>docker build -t knote-java .
[+] Building 4.4s (10/10) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 220B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/adoptopenjdk/openjdk11:jdk-11.0.2.9-slim
=> [auth] adoptopenjdk/openjdk11:pull token for registry-1.docker.io
=> [1/4] FROM docker.io/adoptopenjdk/openjdk11:jdk-11.0.2.9-slim@sha256:f08341ea88e6dfae4a8ded768941574ead9b6794730f55afb8f54fe4bf7c1366
=> [internal] load build context
=> => transferring context: 27.15MB
=> CACHED [2/4] WORKDIR /opt
=> [3/4] COPY target/*.jar /opt/app.jar
=> [4/4] RUN echo hello
=> exporting to image
=> => exporting layers
=> => writing image sha256:bd5bd49c52a8a6f3829769c4194c1513ed22c3a742dc32acadfaab971c44b1e2
=> => naming to docker.io/library/knote-java

Use 'docker scan' to run Snyk tests against images to find vulnerabilities and learn how to fix them

C:\Users\Redak\Desktop\prog avancée\knote-java-01>docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
knote-java latest bd5bd49c52a8 8 seconds ago 385MB
ubuntu latest ba6accedd29 3 months ago 72.8MB

C:\Users\Redak\Desktop\prog avancée\knote-java-01>
```

2) Un container pour la base de données :

MongoDB est fourni sous la forme d'une image Docker nommée mongo sur Docker Hub. Nous allons d'abord télécharger (pull) l'image de mongodb grâce à la commande :

docker pull mongo

```
C:\Users\Redak\Desktop\prog avancée\knote-java-01>docker pull mongo
Using default tag: latest
latest: Pulling from library/mongo
ea362f368469: Pull complete
ecab26900ceb: Pull complete
1847fcb70562: Pull complete
a7de23811c0d: Pull complete
29dd51833fb9: Pull complete
5eccd2be8afb: Pull complete
cd8a8cd6879f: Pull complete
9a70f2129242: Pull complete
9305088dd964: Pull complete
71e019230114: Pull complete
Digest: sha256:e2c554c7281d665f735a2325fd6f6f4079f305b1f8ccb4458c925636664e3526
Status: Downloaded newer image for mongo:latest
docker.io/library/mongo:latest

C:\Users\Redak\Desktop\prog avancée\knote-java-01>docker images
REPOSITORY      TAG         IMAGE ID      CREATED        SIZE
knote-java      latest     bd5bd49c52a8  10 minutes ago 385MB
mongo           latest     b94db0d43242  7 hours ago   698MB
ubuntu          latest     ba6accedd29   3 months ago  72.8MB

C:\Users\Redak\Desktop\prog avancée\knote-java-01>
```

3) Uploader notre image knote-java :

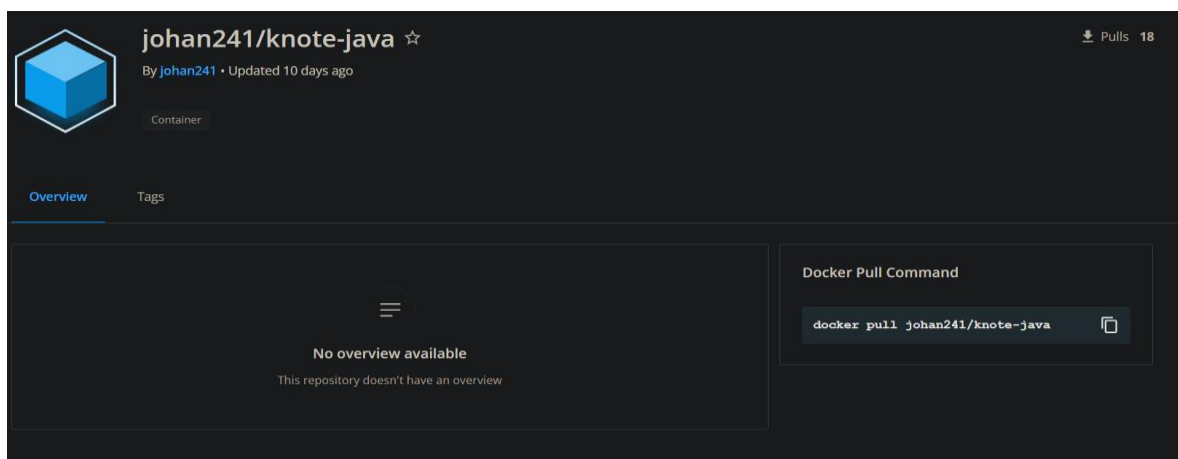
Nous allons uploader l'image nous venons de créer à partir de notre application sur notre docker hub (johan241).

Il faut d'abord renommer notre image selon les critères imposés, le nom passera donc de **knote-java** à **johan241/knote-java**

Pour l'uploader il faut exécuter la commande :

docker push johan241/knote-java:1.0.0

On retrouve notre image dans le lien : <https://hub.docker.com/r/johan241/knote-java>



4) Création d'un network :

On peut maintenant exécuter notre application en ayant un container pour notre partie front end et un container pour notre database.

Il faut tout d'abord créer un docker network et exécuter nos 2 container sur ce network afin qu'ils puissent communiquer entre eux.

```
C:\Users\Redak\Desktop\prog avancée\knote-java-01>docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
f3a407154662	bridge	bridge	local
3da1de180cc0	host	host	local
4ca67e107b24	<u>knote</u>	bridge	local
3a3658a73545	none	null	local

Nous créons ensuite un container sur ce network knote pour la base de donnée en utilisant l'image mongo avec la commande suivante :

```
docker run --name=mongo --rm --network=knote mongo
```

Nous créons aussi un container knote qui va contenir notre application en utilisant l'image qu'on a push sur notre docker hub.

```
C:\Users\Redak>docker run --name=knote-java --rm --network=knote -p 8080:8080 -e MONGO_URL=mongodb://mongo:27017/dev johan241/knote-java:1.0.0
Unable to find image 'johan241/knote-java:1.0.0' locally
1.0.0: Pulling from johan241/knote-java
898c46f3b1a1: Already exists
83366dfa0a50: Already exists
841d4cd74a92: Already exists
8e1bee0f8701: Already exists
8fb802d85d60: Already exists
88a0d02a4ea8: Already exists
ae546bb81870: Already exists
4f4fb708ef54: Already exists
746526a92e4f: Already exists
Digest: sha256:8defaac234eb1cf6907899039073a10d40b9cc8e8efae353aece443f18b0
Status: Downloaded newer image for johan241/knote-java:1.0.0
Picked up JAVA_TOOL_OPTIONS:

Spring
=====
:: Spring Boot :: (v2.1.6.RELEASE)
```

Notre application est en train d'écouter sur le port 8080 et d'émettre sur le port 8080 de notre machine.

Notre application devrait donc être disponible sur l'adresse : **http://localhost:8080/**

Nous pouvons vérifier que nos container sont en exécution.

```
C:\Users\Redak>docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
ee7c64f6ebd0	mongo	"docker-entrypoint.s..."	About a minute ago	Up 59 seconds	27017/tcp	mongo
4089ddc91365	johan241/knote-java:1.0.0	"/bin/sh -c 'exec ja..."	7 minutes ago	Up 7 minutes	0.0.0.0:8080->8080/tcp	knote-java

5. Kubernetes :

1) Création d'un cluster Kubernetes :

Le but étant de créer un cluster Kubernetes local, il existe plusieurs façons d'en créer un.

J'ai opté pour Minikube car je l'ai trouvé plus simple à utiliser malgré qu'il soit en pratique seulement utilisé pour des tests et non pour un déploiement réel.

Minikube crée un cluster Kubernetes à nœud unique s'exécutant dans une machine virtuelle

Il faut tout d'abord installer Minikube et kubectl en utilisant scoop or chocolatey par exemple: **scoop install kubectl & scoop install minikube**. Ensuite exécuter minikube à l'aide de la commande : **start minikube**

Nous pouvons vérifier notre cluster à l'aide de la commande `kubectl cluster-info`

```
C:\Users\Redak>kubectl cluster-info
Kubernetes control plane is running at https://127.0.0.1:54832
CoreDNS is running at https://127.0.0.1:54832/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

“Kubernetes control” étant le cœur de notre cluster, une “interface” qui nous permettra d'interagir avec notre cluster

2) Les fichiers YAML :

La manière dont Kubernetes va devoir déployer, gérer et servir notre application conteneurisée est décrite dans les fichiers yaml présent dans le dossier kube de notre projet :

- **knote.yaml** : pour la partie front end
- **mongo.yaml** : pour la partie base de donnée

On retrouve dans chacun de ces deux fichiers une partie correspondant au **déploiement** et l'autre pour la **mise en service**.

À l'aide des commandes suivantes on crée nos Pods correspondant à Knote and MongoDB containers.

```
C:\Users\Redak\Desktop\prog avancée\knote-java-01>kubectl apply -f kube
deployment.apps/knote unchanged
service/knote unchanged
persistentvolumeclaim/mongo-pvc unchanged
service/mongo unchanged
deployment.apps/mongo unchanged

C:\Users\Redak\Desktop\prog avancée\knote-java-01>kubectl get pods --watch
NAME                                READY   STATUS    RESTARTS   AGE
knote-67cbc5c665-7nfmb             1/1     Running   1           8d
mongo-5cc97976c-5chbd              1/1     Running   2           8d

C:\Users\Redak\Desktop\prog avancée\knote-java-01>minikube service knote --url
* Tunnel de démarrage pour le service knote.
|-----|-----|-----|-----|
| NAMESPACE | NAME   | TARGET PORT | URL                  |
|-----|-----|-----|-----|
| default    | knote  |              | http://127.0.0.1:65465 |
|-----|-----|-----|-----|
http://127.0.0.1:65465
! Comme vous utilisez un pilote Docker sur windows, le terminal doit être ouvert pour l'exécuter.
```

On peut donc accéder à notre application sur : <http://127.0.0.1:65430/>

Avec cette url, on accède au node lead qui lui interagit avec les pods que nous avons créés qui contiennent notre application (front end et backend).

3) Scaling your app :

Nous allons répliquer notre application dans d'autres pods. Je l'ai répliqué 2 fois puis 3 fois pour voir la différence et essayer de capturer le moment de création d'un pod.


```

C:\Users\Redak\Desktop\prog avancée\knote-java-01>kubectl scale --replicas=2 deployment/knote
deployment.apps/knote scaled

C:\Users\Redak\Desktop\prog avancée\knote-java-01>kubectl get pods -l app=knote --watch
NAME                                READY   STATUS    RESTARTS   AGE
knote-67cbc5c665-7nfmb             1/1     Running   1          8d
knote-67cbc5c665-xmdzp             1/1     Running   0          8s

C:\Users\Redak\Desktop\prog avancée\knote-java-01>kubectl scale --replicas=3 deployment/knote
deployment.apps/knote scaled

C:\Users\Redak\Desktop\prog avancée\knote-java-01>kubectl get pods -l app=knote --watch
NAME                                READY   STATUS              RESTARTS   AGE
knote-67cbc5c665-7nfmb             1/1     Running            1          8d
knote-67cbc5c665-bldt5             0/1     ContainerCreating   0          2s
knote-67cbc5c665-xmdzp             1/1     Running            0          28s
knote-67cbc5c665-bldt5             1/1     Running            0          4s

```

Il y a malheureusement un défaut quand je duplique mes containers : il faut parfois rafraichir plusieurs fois la page pour qu'une image se charge. Cela doit être dû au fait qu'on duplique la base de données dans d'autres containers. En dupliquant seulement la partie front end je n'ai pas de problème. Je suppose que c'est à cause de la politique de load balancer qui va utiliser certain pod de base de données au lieu d'autres. J'ai cherché comment faire pour synchroniser les différents pods de mon back end (si un container ayant comme label mongoDB change tous les autres container ayant comme label mongoDB seront mis à jour).

6. QUICK LABS :

J'ai effectué la première quête (y compris le challenge) avec un ami en classe sur son compte, même si nous n'étions pas binôme pour le projet. N'ayant donc pas effectué les labs sur mon compte, je les ai refaits seul en ajoutant 14€ de crédits.

Tout s'est passé facilement vu que je l'avais déjà fait sauf pour le dernier lab où j'ai une erreur et mon lab a été fermé (car je pense que j'ai créé un groupe en trop et le lab n'a pas aimé).

Voilà les étapes que j'ai suivi pour le dernier lab:

1) Première partie : création d'un projet:

- nom de l'instance: nucleus-jumphost911
- machine type f1-micro (série N1)
- image type (Debian Linux)

2) Kubernetes cluster :

1. gcloud config set compute/zone us-east1-b
2. gcloud container clusters create nucleus-webserver1
3. gcloud container clusters get-credentials nucleus-webserver1
4. kubectl create deployment hello-app --image=gcr.io/google-samples/hello-app:2.0
5. kubectl expose deployment hello-app --type=LoadBalancer --port 8083
6. kubectl get service

3) Setup an HTTP load balancer:

a) Enregistrer le fichier startup.sh comme demandé

b) Créez un modèle d'instance :

```
gcloud compute instance-templates create nginx-template --metadata-from-file startup-script=startup.sh
```

c) Créer un pool cible :

```
gcloud compute target-pools create nginx-pool
```

d) Créez un groupe d'instances géré: (ici je me suis trompé, j'ai recréé un groupe sans supprimer le précédent ce qui mis fin au lab)

```
gcloud compute instance-groups managed create nginx-group \  
--base-instance-name nginx \  
--size 2 \  
--template nginx-template \  
--target-pool nginx-pool
```

e) Créer une règle de pare-feu nommée Firewall_rule :

```
gcloud compute firewall-rules create Firewall_rule1 --allow tcp:80
```

```
gcloud compute forwarding-rules create nginx-lb \  
--region us-east1 \  
--ports=80 \  
--target-pool nginx-pool
```

f) Créer une vérification d'état :

```
gcloud compute http-health-checks create http-basic-check
```

```
gcloud compute instance-groups managed \  
set-named-ports nginx-group \  
--named-ports http:80
```

g) Créer un service de backend:

```
gcloud compute backend-services create nginx-backend \  
--protocol HTTP --http-health-checks http-basic-check --  
global
```

```
gcloud compute backend-services add-backend nginx-backend \  
--instance-group nginx-group \  
--instance-group-zone us-east1-b \  
--global
```


h) Créez un mappage d'URL et un proxy HTTP cible :

```
gcloud compute url-maps create web-map \  
--default-service nginx-backend
```

```
gcloud compute target-http-proxies create http-lb-proxy \
```

--url-map web-map

- i) Créer une règle de transfert:
**gcloud compute forwarding-rules create http-content-rule **
**--global **
**--target-http-proxy http-lb-proxy **
--ports 80



Chettab Abdelkader
Date d'abonnement : 2021
Votre profil n'est pas public ni accessible.
[Rendre votre profil public](#)

[Activités](#)[Badges](#)

Activité	Type	Date de début	Date de fin	Score	Réussite
Create and Manage Cloud Resources : atelier challenge	Atelier	Il y a 1 heure	Il y a 24 minutes	40.0/100.0	
Set Up Network and HTTP Load Balancers	Atelier	Il y a 1 heure	Il y a 1 heure	100.0/100.0	✓
Kubernetes Engine: Qwik Start	Atelier	3 janv. 2022	3 janv. 2022	100.0/100.0	✓
Getting Started with Cloud Shell and gcloud	Atelier	3 janv. 2022	3 janv. 2022	100.0/100.0	✓
Getting Started with Cloud Shell and gcloud	Atelier	3 janv. 2022	3 janv. 2022	100.0/100.0	✓
Compute Engine: Qwik Start - Windows	Atelier	2 janv. 2022	2 janv. 2022	100.0/100.0	✓
Creating a Virtual Machine	Atelier	2 janv. 2022	2 janv. 2022	100.0/100.0	✓
A Tour of Google Cloud Hands-on Labs	Atelier	28 déc. 2021	28 déc. 2021	100.0/100.0	✓
Create and Manage Cloud Resource	Quête	27 déc. 2021			✓