TP: Kubernetes, mise en œuvre

Objectifs :

1. Création de cluster
2. Administration de cluster
3. Déployer des charges de travail sur des clusters : Deployments, Services, Statefulsets
4. Créer des Secrets, ConfigMaps, des Volumes Persistants (PV) et Volumes Persistants Claim (PVC), des StorageClass …
5. Configurer des restrictions d'utilisation de ressources, sur les Nodes, Pods, Namespaces

1. Les solutions d'installation

Installation de Microk8s

Prérequis : Snap

Installer Snap sur Linux. (pour les pc CentOs passez directement à l'étape "tester snap")

Pour les versions d'Ubuntu entre 14.04 LTS (Trusty Tahr) et 15.10 (Wily Werewolf), ainsi que pour les versions Ubuntu qui n'incluent pas snap par défaut, snap peut être installé à partir du Centre logiciel Ubuntu en recherchant snapd.

```
$ sudo apt update
$ sudo apt install snapd
```

Déconnectez-vous et reconnectez-vous, ou redémarrez votre système pour vous assurer que les chemins de snap sont correctement mis à jour.

Tester Snap

Pour tester votre système, installez le composant logiciel hello-world et assurez- vous qu'il fonctionne correctement :

```
$ sudo snap install hello-world
hello-world 6.4 from Canonical✓ installed
# si erreur faire : systemctl restart snapd
$ hello-world Hello
World!
```

Snap est maintenant installé et prêt à fonctionner !

Installer Microk8s

```
$ sudo snap install microk8s --classic

# Pour connaitre le status de votre microk8s
$ sudo microk8s status
# Ou avec l'option --wait-ready
$ sudo microk8s status --wait-ready

# Pour ne plus avoir à saisir chaque fois sudo, faire :
$ sudo usermod -a -G microk8s $USER
$ sudo chown -f -R $USER ~/.kube
$ newgrp microk8s
# si changement ne s'opère pas, ouvrir un autre shell

# Pour créer un raccourci, de la commande "microk8s kubectl" ou de "
microk8s.kubectl", faire :
$ sudo snap alias microk8s.kubectl kubectl

# tester avec la commande suivante
$ kubectl version
```

Les plugins disponibles de Microk8s

Avec la commande « `microk8s status` », on peut voir le statut de Microk8s : savoir s'il est en *running* ou non.
Mais aussi, on obtient comme résultat la liste de tous les plugins (« addons ») disponibles sur Microk8s, avec ceux qui sont déjà activés (« enabled ») de ceux qui ne le sont pas (« disabled »).

```
$ microk8s status --wait-ready
    microk8s is running
    high-availability: no
      datastore master nodes: 127.0.0.1:19001
      datastore standby nodes: none
    addons:
    enabled:
       ha-cluster          # (core) Configure high availability on the current node
       helm                # (core) Helm - the package manager for Kubernetes
       helm3               # (core) Helm 3 - the package manager for Kubernetes
    disabled:
       cert-manager        # (core) Cloud native certificate management
       community           # (core) The community addons repository
       dashboard           # (core) The Kubernetes dashboard
       dns                 # (core) CoreDNS
       gpu                 # (core) Automatic enablement of Nvidia CUDA
       host-access         # (core) Allow Pods connecting to Host services smoothly
       hostpath-storage    # (core) Storage class; allocates storage from host directory
       ingress             # (core) Ingress controller for external access
       kube-ovn            # (core) An advanced network fabric for Kubernetes
       mayastor            # (core) OpenEBS MayaStor
       metallb             # (core) Loadbalancer for your Kubernetes cluster
       metrics-server      # (core) K8s Metrics Server for API access to service metrics
       observability       # (core) A lightweight observability stack for logs, traces
    and metrics
       prometheus          # (core) Prometheus operator for monitoring and logging
       rbac                # (core) Role-Based Access Control for authorisation
       registry            # (core) Private image registry exposed on localhost:32000
       storage             # (core) Alias to hostpath-storage add-on, deprecated
```

Comme vous pouvez le constater Microk8s est bien installé et démarré en mode single-node.

Pour activer les add-ons nécessaires il suffit simplement de lancer la ligne de commande suivante:

```
$  microk8s enable storage dns ingress dashboard
```

Pour désactiver les add-ons :

```
$  microk8s uninstall add-on1 add-on2
```

Pour désinstaller tout microk8s proprement:

```
$  snap remove microk8s
microk8s supprimé
```

2. Utilisation de Kubernetes / Microk8s

Premières commandes CLI

```
# création du déploiement et du pod nginx
$ kubectl create deployment nginx --image=docker.io/library/nginx:latest
$ kubectl get deployments
$ kubectl describe deployment nginx
$ kubectl get pods
$ kubectl describe pod nginx-6864b94f57-5j5md
# afficher les log du pod nginx
$ kubectl logs nginx-6864b94f57-5j5md
# se connecter au pod nginx
$ kubectl exec -it nginx-6864b94f57-5j5md –bash

# création d'un namespaces
$ kubectl create namespace development

# création du déploiement dans le namespace development
$ kubectl create deployment nginx --image=docker.io/library/nginx:latest -n=development

# création du service NodePort
$ kubectl expose deployment/nginx --type="NodePort" --port 80
$ kubectl get svc
$ kubectl describe service nginx

# modifier l'image du pod à chaud
$ kubectl set image deployments/nginx nginx=docker.io/library/nginx:nginx-v3
$ kubectl get po
```

Scale up :

Augmenter le nombre de répliques du déploiement par :

```
$ kubectl scale deployments/nginx --replicas=4
$ kubectl get deployments/nginx
```

Nettoyage :

Si une application n'est plus nécessaire, il suffit de la supprimer.

On peut toutefois ramener le replicas de l'application à 0. Ce qui permet de rendre disponible cette application très rapidement. Le nettoyage est enfin fait avec :

```
$ kubectl delete deployments/nginx
$ kubectl delete service nginx
```

Microk8s Dashboard:

```
$ microk8s enable dashboard
# pour lancer le dashboard:
$ microk8s dashboard-proxy
# aller sur le lien https://127.0.0.1:10443 et coller le token affiché sur le terminale
```

3. Objets de base Kubernetes

PhpMyAdmin est une interface d'administration mysql et **Mariadb** basée sur le web libre et écrite en php.

Voici un diagramme que nous aimerions implémenter avec les objets Kubernetes:

En utilisant les fichiers Yaml, créer les objets nécessaires pour déployer l'application phpMyAdmin et sa connexion avec la base de données mariadb;

Indices:

- Utiliser les images docker suivantes : **mariadb:10.4** et **phpmyadmin/phpmyadmin:latest**
- Les variables d'environnement d'un déploiement **phpmyadmin** sont définies comme suit:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
```

```
spec:
  template:
    spec:
      containers:
      - name: test
        env:
          - name: PMA_HOST
            value: <host>
          - name: PMA_PORT
            value: <port>
          - name: MYSQL_ROOT_PASSWORD
            value: <mdp_root>
```

Remarques :

Notez ici que le **database_url (PMA_HOST)**est le même que le nom du service mariadb, `mariadb-service`

Le **containerPort** est :

- le port ouvert sur le serveur mariadb pour phpMyAdmin: 3306
- le port ouvert sur phpMyAdmin: 80

Le **LoadBalancer** attribue à un service une adresse IP externe afin qu'il reçoive des requêtes externes. Contrairement au service interne qui a un Cluster-IP (interne) par défaut, le type de service LoadBalancer a une IP supplémentaire (externe).

Comme le montre la sortie de la commande get, en utilisant un Loadbalancer, il reste en état <pending>, ce qui signifie qu'il n'est pas encore affecté. Ceci est spécifique à Microk8s, sinon, il devrait en avoir un. Par exemple, dans des environnements tels que AWS ou GCP, etc.

Habituellement, le type de service **LoadBalancer** expose le service en externe à l'aide de l'équilibreur de charge d'un fournisseur de cloud. Pour Microk8s, avec le plugin **Ingress** activé, on peut accéder au service à l'externe sur le port spécifique de l'objet **Ingress** créé. *Ce point sera abordé plus en profondeur dans la suite du cours.*

Observations :

```
$ kubectl get svc | grep mariadb-service
$ kubectl describe service mariadb-service
$ kubectl describe service phpmyadmin-service
$ kubectl get pod -o wide
$ kubectl get all | grep php
$ kubectl logs phpmyadmin-deployment-78fcf796b8-f6mwd
$ kubectl exec -it maraidb-deployment-78fcf796b8-f6mwd -- bash
$ kubectl get svc
```

4. Exploiter les objets Kubernetes

Reprendre les objets précédemment créer en utilisant ConfigMap, Secrets et les Volumes

Volume MariaDB

Ajouter un volume pour monter le dossier **/var/lib/mysql** du conteneur mariadb

Secret MariaDB

La **secret** (mot de passe root pour mariaDB) doit être mis en place avant tout déploiement car les pods doivent accéder aux secrets. Alors, créons la secret avec un encodage en base64, voici un exemple:

```
$ echo -n 'username' | base64
dXNlcm5hbWU=
$ echo -n 'password' | base64
cGFzc3dvcmQ=
```

PhpMyAdmin ConfigMap

De même avec les secrets, une **configMap**, définissant les accès à MariaDB, doit déjà être dans notre cluster pour que notre phpMyAdmin puisse l'utiliser comme référence.

5. Création de conteneur personnalisé

Conteneuriser un service python :

Ci-dessous la structure de notre programme :

```
app
├── requirements.txt
└── src
    └── server.py
```

server.py est un simple service Flask.

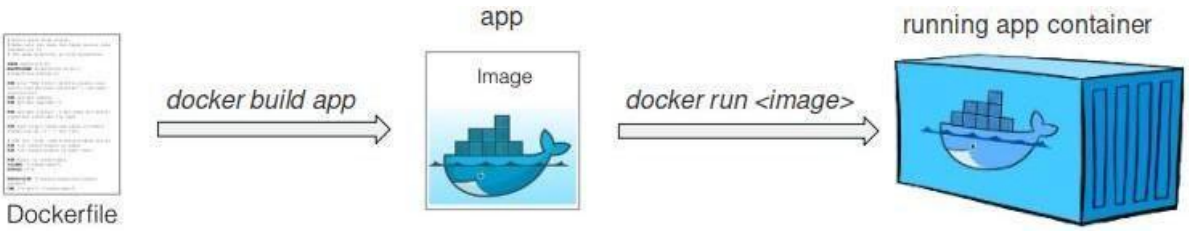
```
from flask import Flask
server = Flask(__name__)
@server.route("/")
def hello():
    return ‘<h1> Hello from Flask ! </h1>’
if __name__== "__main__":
   server.run(host='0.0.0.0')
```

Quant au fichier requirements.txt, il contient les dépendances à installer avant l'exécution du programme.

requirements.txt :

```
Flask==2.1.0
```

DockerFile, en secours pour la conteneurisation :



Dockerfile

```
# set base image (host OS)
FROM python:3.8

# set the working directory in the container
WORKDIR /code

# copy the dependencies file to the working directory
COPY requirements.txt .

# install dependencies
RUN pip install -r requirements.txt

# copy the content of the local src directory to the working directory
COPY src/ .

# command to run on container start
CMD [ "python", "./server.py" ]
```

Build et exécution de l'image

```
$ docker build -t myimage .
$ docker images
$ docker run -d --name container_name -p 5000:5000 myimage
$ docker ps
$ docker ps -a
$ docker logs -f container_name
$ curl http://localhost:5000
```

6. Exploiter les objets Kubernetes

Le **kubelet** utilise des « liveness probes » (sondes de vivacité) pour savoir quand redémarrer un conteneur.

Par exemple, les liveness probes peuvent intercepter un blocage, où une application est en cours d'exécution, mais incapable de progresser. Le redémarrage d'un conteneur dans un tel état peut aider à rendre l'application plus disponible malgré les bogues.

Le kubelet utilise des « readiness probes » (sondes de disponibilité) pour savoir quand un conteneur est prêt à commencer à accepter du trafic.

Un pod est considéré comme prêt lorsque tous ses conteneurs sont prêts. Une utilisation de ce signal est de contrôler quels pods sont utilisés comme backends pour les services.
Lorsqu'un pod n'est pas prêt, il est supprimé des équilibreurs de charge de service.

Le kubelet utilise des « startup probes » (sondes de démarrage) pour savoir quand une application conteneur a démarré. Si une telle probe est configurée, elle désactive les vérifications de liveness et de readiness jusqu'à ce qu'elle réussisse, en s'assurant que ces probes n'interfèrent pas avec le démarrage de l'application.

Cela peut être utilisé pour adopter des contrôles de liveness sur les conteneurs à démarrage lent, en évitant qu'ils ne soient tués par le kubelet avant qu'ils ne soient opérationnels.

7. Kubernetes en production

Ingress / Ingress contrôler :

Créer un frontal ingress qui affiche l'application phpMyAdmin sur le domaine "formation.test.com"

Si Ingress ne fonctionne pas sur **CentOs**, lancer:

```
sudo systemctl stop firewalld
sudo iptables -P FORWARD ACCEPT
```

Gestion des accès :

Récupérer ca.crt et ca.key ici : **/var/snap/microk8s/current/certs**

Générer la clé de Marc:

```
$ openssl genrsa -out marc.key 2048

$ openssl req -new -key marc.key -out marc.csr -subj "/CN=marc/O=datascience"

$ openssl x509 -req -in marc.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out marc.crt
-days 365
```

Créer le context de Marc:

```
$ kubectl config set-credentials marc --client-certificate=marc.crt
--client-key=marc.key

$ kubectl config set-context marc-context --cluster=microk8s-cluster
--namespace=datascience --user=marc
```

Utiliser le context en tant que Marc:

```
$ kubectl --context=marc-context get pods
```

Créer les ressources **Role** et **RoleBinding** pour donner des accès à Marc et re tester.

7. Déploiement d'un cluster Kubernetes (Microk8s)

Nous allons créer des machines virtuelles qui vont constituer les nœuds de notre cluster, avec Multipass.

Multipass : c'est « VM Ubuntu à la demande pour n'importe quel poste de travail ».
Il permet d'obtenir une machine virtuelle Ubuntu instantanée avec une seule commande. Multipass peut lancer et exécuter des machines virtuelles et les configurer avec cloud-init comme un cloud public. Vous pouvez donc prototyper gratuitement le lancement de votre cloud localement.

Installer Multipass sur Linux (Ubuntu) :

```
$ sudo snap install Multipass
```

Chaque VM Multipass tourne sur un Ubuntu 18.04 LTS ou Ubuntu 20.04 LTS.

Créer une VM Ubuntu avec Multipass :

```
$ multipass launch --name kube-node --mem 6G --disk 50G –cpus 2
```

Installer Docker:optionnel car Microk8s vient avec Containerd (qui est un conteneur runtime)

Vérifier la visibilité réseau :

```
$ multipass get local.driver

 # doit être lxd, sinon istaller lxd et le définir comme driver
$ multipass set local.driver=lxd
```

Installer Microk8s sur chaque VM créée

(Voir guide d'installation de microk8s)

```
$ multipass exec kube-node -- sudo snap install microk8s –classic

# Ouvrir le shell de votre VM :
$ multipass shell kube-node
```

Joindre les différents nœuds par :

Déploiement : Ajouter un nœud worker au master-node
Lancer « microk8s add-node » sur le master (# Noter bien l'output de cette commande).

```
$ microk8s add-node
```

```
Join node with:
microk8s join ip-172-31-20-243:25000/DDOkUupkmaBezNnMheTBqFYHLWINGDbf

If the node you are adding is not reachable through the default
interface you can use one of the following:

microk8s join 10.1.84.0:25000/DDOkUupkmaBezNnMheTBqFYHLWINGDbf
microk8s join 10.22.254.77:25000/DDOkUupkmaBezNnMheTBqFYHLWINGDbf
```

Lancer microk8s join ip … sur le worker-node :

```
microk8s join ip-172-31-20-243:25000/DDOkUupkmaBezNnMheTBqFYHLWINGDbf
```

```
# faire sur chaque nœud :
$ kubectl get nodes
```

Ajouter un label à un noeud :

```
$ kubectl get nodes --show-labels
$ kubectl label nodes <your-node-name> disktype=ssd
```

Ajouter **nodeAffinity** à un pod précédemment créé pour le reprogrammer dans le node ayant le label disktype=ssd

Retrait d'un node du cluster

Le retrait d'un node du cluster se fait via la commande:

```
$ microk8s leave
# puis depuis le noeud master:
$ microk8s remove-node <your-node-name>
```