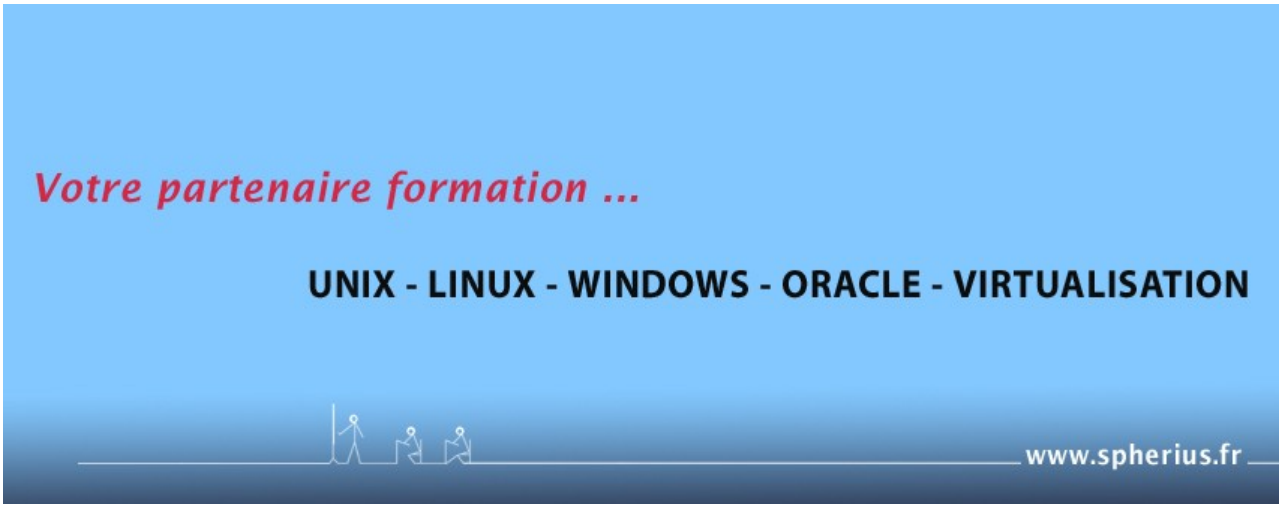
KUBERNETES

CORRECTION

EXERCICES



SOMMAIRE

Les auteurs de ce document sont :

- Steeven Herlant,
- Jean-Marc Baranger,
- Theo Schomaker.

Les versions utilisées pour ce support d'exercices sont :

- Ubuntu: 20.04

Les références sont : les documents disponibles sur le site web de Kubernetes.

L'environnement de formation comprend 4 machines :
- admin : machine d'administration pour nous connecter aux nœuds de notre cluster.
- master : machine qui servira comme machine maitre de notre cluster kubernetes.
- worker1 et worker2 : nœuds supplémentaires de notre cluster kubernetes.

Sur toutes les machines le mot de passe de l'utilisateur user1 est user1, celui de l'administrateur root est root.

Les clefs ssh ont été échangé entre les machines pour éviter de saisir le mot de passe à chaque connexion.

Correction des Exercices

Correction – Partie 1 : Installation et Configuration de Kubernetes

Exercice 1 : L'installation des pré-requis

Installer les paquets nécessaires.

```
# apt install -y apt-transport-https gnupg2 software-properties-common
ca-certificates
```

Configurer le dépôt de Kubernetes.

```
# mkdir -p /etc/apt/keyrings
# curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.28/deb/Release.key \
    | sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
# cat <<EOF > /etc/apt/sources.list.d/kubernetes.list
deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg]
  https://pkgs.k8s.io/core:/stable:/v1.28/deb/ /
EOF
```

Modifier sysctl pour l'utilisation de Kubernetes.

```
# cat <<EOF > /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
EOF
# sysctl --system
```

Activer les modules br_netfilter et overlay.

```
# modprobe br_netfilter
# modprobe overlay
```

Enlever la swap.

```
# swapoff -a
```

Editer la fstab pour supprimer la ligne permettant le montage de la swap.
Configurer le dépôt Docker.

```
# curl -s https://download.docker.com/linux/ubuntu/gpg | apt-key add -
# add-apt-repository "deb [arch=amd64] \
"https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
```

Installer containerd.

```
# apt install -y containerd.io
```

Mettre en place la configuration de containerd.

```
# mkdir -p /etc/containerd
```

```
# containerd config default | tee /etc/containerd/config.toml
```

Modifier la configuration containerd afin d'utiliser systemd-cgroup.

```
# sed -i 's/SystemdCgroup = false/SystemdCgroup = true/' \
/etc/containerd/config.toml
```

Redémarrer et activer le service systemd de containerd.

```
# systemctl restart containerd
# systemctl enable containerd
```

Installer Les paquets nécessaires au fonctionnement du cluster Kubernetes.

```
# apt install -y kubelet kubeadm kubectl
```

Activer Kubelet au démarrage.

```
# systemctl enable --now kubelet
```

Bloquer les versions des paquets.

```
# apt-mark hold kubelet kubeadm kubectl
```

Exercice 2 : L'installation du cluster

Initialiser le master.

```
# kubeadm init --apiserver-advertise-address=192.168.56.31 --node-name \
$HOSTNAME --pod-network-cidr=10.244.0.0/16
```

Noter pour plus tard la commande de join fournie à la fin de l'initialisation.

```
kubeadm join 192.168.56.31:6443 --token 71gtp4.71h19g1junprrgos \
    --discovery-token-ca-cert-hash
sha256:732c62c66df2f181cc8c0814363aaa058b2473c81a0bc8d6f0a922c64f5bcd7c
```

Cette commande est générée à chaque kubeadm init.

Se connecter en tant que user1.

```
# su - user1
```

Copier le fichier de configuration dans le répertoire personnel de l'utilisateur user1.

```
$ mkdir -p $HOME/.kube
$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Vérifier l'état des noeuds.

```
$ kubectl get nodes
```

Vérifier l'état des pods.

```
$ kubectl get pods --all-namespaces
```

Récupérer les fichiers manifests pour Calico.

```
$ wget -O /tmp/tigera-operator.yml \
https://docs.projectcalico.org/manifests/tigera-operator.yaml
$ wget -O /tmp/custom-resources.yml \
https://docs.projectcalico.org/manifests/custom-resources.yaml
```

Modifier le fichier /tmp/custom-resources.yml afin de prendre en compte la bonne interface réseau, ainsi que le bon réseau pour nos pods.

```
$ cat /tmp/custom-resources.yml
...
spec:
  # Configures Calico networking.
  calicoNetwork:
    nodeAddressAutodetectionV4:
      interface: enp0s8
    # Note: The ipPools section cannot be modified post-install.
    ipPools:
    - blockSize: 26
      cidr: 10.244.0.0/16
      encapsulation: VXLANCrossSubnet
      natOutgoing: Enabled
      nodeSelector: all()
...
```

Déployer Calico.

```
$ kubectl create -f /tmp/tigera-operator.yml
$ kubectl create -f /tmp/custom-resources.yml
```

Vérifier l'état des pods. Qu'est-ce qui a changé ?

```
$ kubectl get pods --all-namespaces
```

Nous pouvons constater la création des pods pour Calico ainsi que le changement d'état des pods coredns.

Vérifier l'état des noeuds. Qu'est-ce qui a changé ?

```
$ kubectl get nodes
```

Notre node master est maintenant à l'état Ready.

Afficher les conteneurs en cours d'execution, utiliser la commande pour containerd suivante :

```
$ sudo ctr --namespace k8s.io containers list
CONTAINER                                                           IMAGE
RUNTIME
076b5847efeb335361e009fe1812fbda9fc5f69cdc83c143cfa6973c4723cf98
k8s.gcr.io/etcd:3.5.3-0                          io.containerd.runc.v2
08a6aeb3d4be2dc4e1872c994101a8259310159807beca540f64323d38ea2d99    k8s.gcr.io/pause:...
io.containerd.runc.v2
1569600835625e834bcb5b5fd61d614cb9ab80ce80a0fdcf69937d4523958bc4    k8s.gcr.io/pause:...
io.containerd.runc.v2
23b97fd761cd3c65d671b4b58a880ea90b61d13ff1eeb438ae7d4929d314ae54    k8s.gcr.io/pause:...
io.containerd.runc.v2
3dec7154ebd14ec668b9f94c42b6924ed8ea16d8a6cf63b0e220d2e47c41543f
docker.io/calico/node:v3.23.3                    io.containerd.runc.v2
435e072d8eab662e3e91dcbba6757538e76b0381ba233c626458180e521cb760    k8s.gcr.io/kube-
```

```
controller-manager:v1.24.3     io.containerd.runc.v2
4eaf12e335f0fa62b1b319956d35882407d14b9b91980e864c145abc1bd828d2    k8s.gcr.io/pause:...
io.containerd.runc.v2
4f298cc464ddb7ee61ab3a7dc3dca5ff4399b474c50592100d7f5cf65f3afcc1    k8s.gcr.io/pause:...
io.containerd.runc.v2
513d6ef7eb1a5ceb91b55ed257ee12843eeb9d2ac1cfcfd6452758834d7b2fe4
quay.io/tigera/operator:v1.27.12               io.containerd.runc.v2
5a5d6092d4e77d8e2f63d6be3865c5935e839314aaa73476091adbc06cd1bd72    k8s.gcr.io/pause:...
io.containerd.runc.v2
5a72dcae640421a3d5c9ac202adae18d4317470d1972efae7d4ce8205f6511cb
docker.io/calico/pod2daemon-flexvol:v3.23.3    io.containerd.runc.v2
61037ea210909829eeaf7f2f72651ca34b265e6239f9e8a02724363f34825939
k8s.gcr.io/coredns/coredns:v1.8.6              io.containerd.runc.v2
7df4a726dd601348504477d1c6962fee72e826b7af604befcbbdd5b06c6151e3    k8s.gcr.io/pause:...
io.containerd.runc.v2
8529e9cdf533a0de2ae63760540f016920dcef3713cfccaf0c6bc9eb37241502    k8s.gcr.io/kube-
scheduler:v1.24.3              io.containerd.runc.v2
8b364c703cc4da0fb50ea84d07957daaf64d5106f38b3f57d51caa6df2f82f69    k8s.gcr.io/pause:...
io.containerd.runc.v2
92f7a8ce6a369664d8a0bf64248d2dbfdb68cf7292279d0423420ca03856d059
k8s.gcr.io/coredns/coredns:v1.8.6              io.containerd.runc.v2
b43a9c1cb43eb6a66e01d013f74e1ba4ba434469ca571df710bbf27b6b924792
docker.io/calico/cni:v3.23.3                   io.containerd.runc.v2
ba1b748d51c21dd4389869cf2e8b18e4059b64aedd2518d4b81f7c1b0c06c712    k8s.gcr.io/pause:...
io.containerd.runc.v2
be603e5418204bb7dc735b7590dc406edf60fa3a6afc871250c2d65436721a63    k8s.gcr.io/kube-
apiserver:v1.24.3              io.containerd.runc.v2
c74c1ca1534ab4f4cb9fed8d18af92efd4cdd0227664783bb9fe257d6db320bd    k8s.gcr.io/kube-
proxy:v1.24.3                  io.containerd.runc.v2
e485f569bf74f6a77ff8abe45f75dcb1a379500b639e0d29f47c8e67e910a968
docker.io/calico/typha:v3.23.3                 io.containerd.runc.v2
f5f50b26acb398bef6e066d9673d97d2ade1f1a523349f8ba1202c595d3aafa2    k8s.gcr.io/pause:...
io.containerd.runc.v2
```

Faire joindre les workers dans le cluster (en tant que root).

```
# kubeadm join 192.168.56.31:6443 --token 71gtp4.71h19g1junprrgos \
    --discovery-token-ca-cert-hash
sha256:732c62c66df2f181cc8c0814363aaa058b2473c81a0bc8d6f0a922c64f5bcd7c
```

Contrôler que les workers ont bien rejoint le cluster.

```
# kubectl get nodes
```

Vérifier les pods maintenant présent sur le cluster. Que constatez-vous ?

```
# kubectl get pods --all-namespaces -o wide
```

Il y un nouveau pod calico-node ainsi qu'un pod kube-proxy par nœud. Un pod calico-kube-controller est présent quand à lui sur le master.

Correction – Partie 2 : Administration du cluster en cli

Exercice 1 : Récupérer des informations

Afficher l'état des nœuds du cluster de manière détaillée.

```
$ kubectl get nodes -o wide
```

Afficher l'aide de kubectl.

```
$ kubectl help
```

Afficher l'aide de kubectl get.

```
$ kubectl get --help
```

Récupérer les informations des nœuds en formattant la sortie en json.

```
$ kubectl get nodes -o json
```

Afficher les pods présents sur le cluster dans tous les namespaces, en utilisant l'option courte.

```
$ kubectl get pods -A
```

Exercice 2 : Premiers pods

Lancer un premier pod nommé pod1, à partir de l'image debian, sans autre option.

```
$ kubectl run --image debian pod1
```

Vérifier le status du pod. Pourquoi est-il dans ce status ?

```
$ kubectl get pods
```

Le pod va créer un conteneur basé sur une image debian. Cependant, aucun processus ne toune par défaut sur notre instance. Le conteneur ne peut donc pas rester up.

Supprimer le pod.

```
$ kubectl delete pod pod1
```

Relancer le même pod, en y attachant cette fois-ci un TTY.

```
$ kubectl run --image debian -ti pod1
```

Afficher le nom du pod.

```
# hostname
```

Quitter le TTY à l'aide d'un <CTRL>+d. Quel est l'état du pod ?

```
# <CTRL-D>
```

Le pod est toujours à l'état Running. Cela est dû au fait que nous nous sommes juste détaché du TTY, nous ne l'avons pas arrêté. Celui-ci tourne donc toujours dans notre conteneur, ce qui le maintien up.

Afficher les informations concernant le pods à l'aide de la sous-commande get pour obtenir l'adresse ip qui lui a été affecté.

```
$ kubectl get pods -o wide
```

Afficher la description du pod.

```
$ kubectl describe pod pod1
```

Afficher les informations concernant le pod au format json, puis au format yaml.

```
$ kubectl get pods -o json pod1
$ kubectl get pods -o yaml pod1
```

Utiliser les custom-columns pour afficher, le nom et l'adresse ip du pod, ainsi que le nœud sur lequel il tourne.

```
$ kubectl get pods -o \
custom-columns=nom:.metadata.name,ip:.status.podIP,noeud:.spec.nodeName
```

Utiliser la sous-commande attach pour se reconnecter au pod, puis se deconnecter.

```
$ kubectl attach -ti po pod1
# <CTRL-D>
```

Supprimer le pod.

```
$ kubectl delete pod pod1
```

Exercice 3 : Premiers déploiements

Créer un déploiement nommé mon-serveur-web, à partir d'une image nginx.

```
$ kubectl create deploy --image nginx mon-serveur-web
```

Afficher les informations et descriptions concernant ce déploiement, ainsi que les informations concernant le pod.

```
$ kubectl get deploy
$ kubectl describe deploy mon-serveur-web
$ kubectl get pod
$ kubectl describe pod mon-serveur-web-d4bb64c4c-h26n9
```

Pourquoi le nom du pod n'est pas mon-serveur-web ?
Il s'agit ici d'un déploiement. C'est donc Kubernetes qui va générer un nom pour les pods qui seront créés lors du déploiement.

Supprimer le pod. Que constatez vous ?

```
$ kubectl delete pod mon-serveur-web-d4bb64c4c-h26n9
```

Un nouveau pod est automatiquement créé par Kubernetes.

Editer le déploiement, pour y ajouter un nouveau conteneur, nommé mon-debian, et basé sur une image debian. Ajouter une commande que lancera notre conteneur, un sleep de 1 minute.

```
$ kubectl edit deploy mon-serveur-web
spec:
  containers:
    - image: debian
      name: mon-debian
      command: ["sleep","60"]
```

Afficher à nouveau les informations et la description du pod, que constatez-vous ?

```
$ kubectl get pods
$ kubectl describe pod mon-serveur-web-79d8b875-hsf2n
```

Le pod contient bien deux conteneurs, comme nous pouvons le constater via le nombre 2/2 au get ou encore dans la description du pod.

Analyser la partie Events dans la description du pod. Que constatez-vous ?
Toutes les minutes, le conteneur débian est recréé. Cela est dû au sleep qui se termine. Notre conteneur s'arrête donc, et il est automatiquement recréé par le déploiement.

Supprimer le deploiement.

```
$ kubectl delete deploy mon-serveur-web
```

Exercice 4 : Autocompletion

Vérifier la présence du paquet bash-completion sur le master, l'installer le cas échéant.

```
$ sudo dpkg -l|grep bash-completion
$ sudo apt install -y bash-completion
```

Ajouter à la fin du .bashrc de l'utilisateur user1 la commande permettant d'activer la completion de kubectl

```
$ echo "source <(kubectl completion bash)" >> ~/.bashrc
```

Se delogguer/relogguer pour que la complétion soit prise en compte.

Vérifier le bon fonctionnement en tapant la commande kubectl, suivi de <TAB><TAB>

Correction – Partie 3 : L'utilisation des fichiers yaml

Exercice 1 : Exporter les fichiers

Créer un déploiement nommé mon-debian, à partir d'une image debian en version 11.

```
$ kubectl create deploy --image debian:11 mon-debian
```

Vérifier que la version du conteneur est conforme à celle demandée :

Exporter les informations concernant ce déploiement en yaml. Que constatez vous ?

```
$ kubectl get deploy mon-debian -o yaml
```

Un grand nombre de spécifications ont été générées par défaut, alors que nous n'avons donné que l'image à utiliser en option à la création de notre déploiement.

Dans quel état est le pod ?
Le pod est en CrashLoopBackOff, car il n'a pas de processus actif et s'arrête donc immédiatement à chaque fois que le déploiement le recréé.

Exporter la configuration du déploiement dans un fichier mon-debian.yml.

```
$ kubectl get deploy mon-debian -o yaml > mon-debian.yml
```

Editer le fichier généré :

```
$ vi mon-debian.yml
```

- Changer l'image utilisée pour une image debian 10, et y ajouter un sleep de 60 secondes.

```
- image: debian:10
  command: ["sleep", "60"]
```

Supprimer le deploiement.

```
$ kubectl delete deploy mon-debian
```

Recréer le déploiement et vérifier.

```
$ kubectl apply -f mon-debian.yml
$ kubectl get pods
$ kubectl get pod mon-debian-7c9654c6b5-t78hc -o yaml
```

Exercice 2 : Créer ses propres fichiers

Créer un fichier « mon-premier-pod.yml » pour lancer un pod avec les contraintes suivantes :

- nom du pod : mon-premier-pod
- conteneur :
  - image : ubuntu
  - nom : mon-ubuntu
  - commande : sleep 600

```
---
apiVersion: v1
kind: Pod
metadata:
  name: mon-premier-pod
spec:
  containers:
  - name: mon-ubuntu
    image: ubuntu
    command: ["sleep","600"]
```

Appliquer la configuration :

```
$ kubectl apply -f mon-premier-pod.yml
```

Executer un processus bash en mode interactif et terminal dans le conteneur, puis vérifier la version d'ubuntu utilisée.

```
$ kubectl exec -ti mon-premier-pod -- bash
root@mon-premier-pod # cat /etc/os-release
root@mon-premier-pod # <CTRL-D>
```

Modifier le fichier de configuration pour utiliser une image ubuntu 18.04, vérifier.

```
$ vi mon-premier-pod.yml
---
apiVersion: v1
kind: Pod
metadata:
  name: mon-premier-pod
spec:
  containers:
  - name: mon-ubuntu
    image: ubuntu:18.04
    command: ["sleep","600"]
```

```
$ kubectl apply -f mon-premier-pod.yml
$ kubectl exec -ti mon-premier-pod -- bash
root@mon-premier-pod # cat /etc/os-release
```

Créer un fichier pour lancer un déploiement avec les contraintes suivantes :

- nom du déploiement : mon-premier-deploiement
- selecteur :
  - matchLabels: app => web
- template :
  - label : app => web
  - conteneur :
    - nom : mon-serveur-web
    - image : nginx

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mon-premier-deploiement
spec:
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
      - name: mon-serveur-web
        image: nginx
```

```
$ kubectl apply -f mon-premier-deploiement.yml
```

Récupérer l'ip du pod généré ainsi que le nœud sur lequel il a été déployé, et vérifier à l'aide de la commande curl le bon fonctionnement du serveur web, sur le nœud qui héberge le pod.

```
$ kubectl get pods -o wide
$ curl 10.0.235.136
```

Supprimer le pod mon-premier-pod et le déploiement mon-premier-deploiement.

```
$ kubectl delete pod mon-premier-pod
$ kubectl delete deploy mon-premier-deploiement
```

Correction – Partie 4 : Administration

Exercice 1 : Utilisation des ressources

Créer un deploiement avec les contraintes suivantes :

- nom du déploiement : utilisation-ressources
- selecteur :
  - matchLabels: stress => test
- template :
  - label : stress => test
  - conteneur :
    - nom : ubuntu1
    - image : ubuntu
    - command : sleep 10 minutes

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: utilisation-ressources
spec:
  selector:
    matchLabels:
      stress: test
  template:
    metadata:
      labels:
        stress: test
    spec:
      containers:
      - name: ubuntu1
        image: ubuntu
        command: ["sleep","600"]
```

```
$ kubectl apply -f utilisation-ressources1.yml
```

Identifier le nœud sur lequel tourne le conteneur et lancer dans un nouveau terminal un top. Le laisser tourner et noter le load average du nœud. Vérifier également la consommation RAM en utilisant la command free.

```
$ kubectl get pods -o wide
$ ssh worker1
$ free
$ top
```

Executer un processus bash à l'intérieur du conteneur généré à l'aide d'un autre terminal. Faire un update des paquets et installer le paquet stress

```
$ kubectl exec -ti utilisation-ressources-5d49d97945-pk7t4 -- bash
# apt update && apt install -y stress
```

Lancer la commande stress avec l'option -c 1000 pour demander l'utilisation d'autant de cpu

```
# stress -c 1000
```

Laisser le processus tourner et retourner vérifier l'état de la commande top, notamment le load average. Que constatez-vous ?
Le load average s'emballe et monte en flèche. C'est également le cas de la ram.
Normalement, notre commande stress devrait s'arrêter au bout de quelques secondes, manquant de ressources.

Modifier le fichier yaml de votre déploiement pour respecter les contraintes suivantes :

- nom du déploiement : utilisation-ressources
- selecteur :
  - matchLabels: stress => test
- template :
  - label : stress => test
  - conteneur :
    - nom : ubuntu1
    - image : ubuntu
    - command : sleep 10 minutes
    - ressources :
      - requests : cpu à 100milliCPU et RAM à 100Mo
      - limits : cpu à 500milliCPU et RAM à 300Mo

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: utilisation-ressources
spec:
  selector:
    matchLabels:
      stress: test
  template:
    metadata:
      labels:
        stress: test
    spec:
      containers:
      - name: ubuntu1
        image: ubuntu
        command: ["sleep","600"]
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
          limits:
            cpu: 500m
            memory: 300Mi
```

```
$ kubectl apply -f utilisation-ressources.yml
```

Reverifier à l'aide des commandes top et free l'état du worker sur lequel tourne notre pod. Que constatez-vous maintenant ?
Les consommations au niveau du worker ne s'emballe plus, grâce à la limitation que nous avons appliqué à notre conteneur.

Supprimer le déploiement.

```
$ kubectl delete deploy utilisation-ressources
```

Exercice 2 : Labels, NodeSelector et NodeName

Placer 2 labels sur les workers :

- worker1 : datacenter => Paris
- worker2 : datacenter => Marseille

```
$ kubectl label nodes worker1 datacenter=Paris
$ kubectl label nodes worker2 datacenter=Marseille
```

Vérifier.

```
$ kubectl get nodes -o custom-columns=\
NAME:.metadata.name,LABELS:.metadata.labels.datacenter
```

Adapter les fichiers du répertoires des sourcess pour créer 4 déploiements avec les contraintes suivantes :

- Un template de pod contiendra un conteneur basé sur une image nginx, tournant sur le datacenter de Paris
- Un template de pod contiendra un conteneur basé sur une image nginx, tournant sur le datacenter de Marseille
- Un template de pod contiendra un conteneur basé sur une image debian, tournant sur le nœud worker1, et la commande sleep 600 permettant de maintenir le conteneur up.
- Un template de pod contiendra un conteneur basé sur une image debian, tournant sur le nœud worker2, et la commande sleep 600 permettant de maintenir le conteneur up.

```
$ cat nginx-paris.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-paris
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-paris
  template:
    metadata:
      labels:
        app: nginx-paris
    spec:
      containers:
      - name: nginx
        image: nginx
      nodeSelector:
        datacenter: Paris
$ cat nginx-marseille.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
  name: nginx-marseille
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-marseille
  template:
    metadata:
      labels:
        app: nginx-marseille
    spec:
      containers:
      - name: nginx
        image: nginx
      nodeSelector:
        datacenter: Marseille
```

$ cat debian-worker1.yml

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: debian-worker1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: debian-worker1
  template:
    metadata:
      labels:
        app: debian-worker1
    spec:
      containers:
      - name: debian
        image: debian
        command: ["sleep","600"]
      nodeName: worker1
```

$ cat debian-worker2.yml

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: debian-worker2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: debian-worker2
  template:
    metadata:
      labels:
        app: debian-worker2
    spec:
      containers:
      - name: debian
        image: debian
        command: ["sleep","600"]
      nodeName: worker2
```

```
$ kubectl apply -f nginx-paris.yml -f nginx-marseille.yml \
-f debian-worker1.yml -f debian-worker2.yml
```

Verifier les workers sur lesquels tournent les pods.

```
$ kubectl get pods -o wide
```

Nous avons bien nos pods issus des déploiements nginx-paris et debian-worker1 qui tobuurnent sur worker1, et nginx-marseille et debian-worker2 sur worker2.

Supprimer les 4 déploiements.

```
$ kubectl delete deploy  nginx-paris  nginx-marseille debian-worker1
debian-worker2
```

Exercice 3 : Namespaces, Contextes

Créer le namespace exercice3.

```
$ kubectl create ns exercice3
```

Adapter le fichier nginx-marseille.yml présent dans le répertoire des sources pour qu'il utilise le namespace exercice3.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-marseille
  namespace: exercice3
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-marseille
  template:
    metadata:
      labels:
        app: nginx-marseille
    spec:
      containers:
      - name: nginx
        image: nginx
      nodeSelector:
        datacenter: Marseille
```

Déployer le projet.

```
$ kubectl apply -f nginx-marseille.yml
```

Vérifier que le déploiement tourne bien dans le namespace exercice3.

```
$ kubectl get deploy -n exercice3
$ kubectl get pods -n exercice3
```

Créer un nouveau contexte, qui utilisera le namespace exercice3, et l'utilisateur kubernetes-admin, puis switcher sur ce nouveau contexte.

```
$ kubectl config set-context exercice3 --namespace exercice3 --user \
kubernetes-admin --cluster kubernetes
$ kubectl config use-context exercice3
```

Vérifier que le contexte est bien configuré en affichant les pods du namespace actuel.

```
$ kubectl get pods
```

Supprimer le namespace exercice3. Vérifier ensuite l'état des pods et du contexte exercice3. Que constatez-vous ?

```
$ kubectl delete ns exercice3
$ kubectl get pods
$ kubectl config get-contexts
```

Lors de la suppression du namespace, les pods appartenant à celui-ci a été supprimé. En revanche, le contexte est toujours présent et a toujours le namespace exercice3 dans sa configuration.

Changer de contexte en sélectionnant celui par défaut et supprimer le contexte exercice3.

```
$ kubectl config use-context kubernetes-admin@kubernetes
$ kubectl config delete-context exercice3
```

Exercice 4 : Scaling

Créer un nouveau namespace exercice4.

```
$ kubectl create ns exercice4
```

Créer un déploiement avec les contraintes suivantes :

- nom du déploiement : scaling
- replicas : 2
- namespace : exercice4
- selecteur :
  - matchLabels: app => scale
- template :
  - label : app => scale
  - conteneur :
    - nom : scale-nginx
    - image : nginx
    - ressources :
      - requests : cpu à 100milliCPU et RAM à 100Mo
      - limits : cpu à 500milliCPU et RAM à 300Mo

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: scaling
  namespace: exercice4
spec:
  replicas: 2
  selector:
    matchLabels:
      app: scale
  template:
    metadata:
      labels:
        app: scale
    spec:
      containers:
      - name: scale-nginx
```

```
        image: nginx
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
           limits:
            cpu: 500m
            memory: 300Mi
```

Déployer le projet.

```
$ kubectl apply -f scaling.yml
```

Vérifier que le déploiement respecte les contraintes demandées, notamment le nombre de replicas. Quelle est la répartition sur le cluster ?

```
$ kubectl get pods -o wide -n exercice4
```

Nous avons un pod sur chaque worker.

Modifier en ligne de commande le nombre de replicas pour le passer à 5. Quelle est maintenant la répartition ?

```
$ kubectl scale deploy -n exercice4 scaling --replicas=5
```

Nous avons 2 pods sur chaque worker. Le dernier pod tourne sur un worker de façon arbitraire, Kubernetes essayant de répartir de manière équitable le scale.

Supprimer le namespace exercice4.

```
$ kubectl delete ns exercice4
```

Correction – Partie 5 : Les Services

Exercice 1 : ClusterIp

Créer le namespace services.

```
$ kubectl create ns services
```

Adapter le fichier deploy-cluster-ip.yml présent dans le répertoire des sources pour que cela respecte les contraintes suivantes :

- nom du déploiement : deploy-cluster-ip
- replicas : 2
- namespace : services
- selecteur :
  - matchLabels: app => cluster-ip
- template :
  - label : app => cluster-ip
  - conteneur :
    - nom : cluster-ip-nginx
    - image : nginx

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-cluster-ip
  namespace: services
spec:
  replicas: 2
  selector:
    matchLabels:
      app: cluster-ip
  template:
    metadata:
      labels:
        app: cluster-ip
    spec:
      containers:
      - name: cluster-ip-nginx
        image: nginx
```

Déployer le projet.

```
$ kubectl apply -f deploy-cluster-ip.yml
```

Vérifier que chaque worker fait bien tourner un pod.

```
$ kubectl get pods -o wide -n services
```

Copier le fichier index-worker1.html sur le pod tournant sur worker1, dans /usr/share/nginx/html/index.html. Le fichier se trouve dans le répertoire des sources.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Worker1</title>
  </head>
  <body>
    <h1>Vous etes sur Worker1</h1>
  </body>
</html>
```

```
$ kubectl exec -ti -n services cluster-ip-$podId -- bash
$ apt update && apt install -y vim
$ vim /usr/share/nginx/html/index.html
```

Faire la même chose sur le pod tournant sur worker2, et utiliser le fichier suivant :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Worker2</title>
  </head>
  <body>
    <h1>Vous etes sur Worker2</h1>
  </body>
</html>more
```

```
$ kubectl exec -ti -n services cluster-ip-$podId -- bash
$ apt update && apt install -y vim
$ vim /usr/share/nginx/html/index.html
```

Créer maintenant un service de type ClusterIp, avec les contraintes suivantes :

- nom du service : svc-cluster-ip
- namespace : services
- type : ClusterIp
- port d'écoute : 8000
- port utilisé par le pod : 80
- selecteur : app => cluster-ip

```
---
kind: Service
apiVersion: v1
metadata:
  name: svc-cluster-ip
  namespace: services
spec:
  type: ClusterIP
  ports:
  - port: 8000
    targetPort: 80
  selector:
    app: cluster-ip
```

```
$ kubectl apply -f svc-cluster-ip.yml
```

Récupérer les ips des pods.

```
$ kubectl get pods -n services -o wide
```

Vérifier par un curl le bon fonctionnement du serveur web nginx depuis ces 2 ips. Vérifier à l'aide du texte des pages que nous contactons bien le bon serveur web.

```
$ curl 10.0.189.84
```

Récupérer l'ip affectée à notre service.

```
$ kubectl get svc -n services
```

Lancer un curl sur cette ip, en veuillant bien à utiliser le port définit lors de la création du service.

```
$ curl 10.106.237.129:8000
```

Répéter plusieurs fois la commande. Que constatez-vous ?
Un Load Balancing est automatiquement créé par Kubernetes. De ce fait, et comme nous avons 2 réplicas de notre déploiement, nous affichons tour à tour le serveur web 1 ou 2.

Augmenter le nomber de replicas à 3. Répéter le curl. Que constatez-vous ?

```
# kubectl scale deploy -n services  --replicas=3 deploy-cluster-ip
```

Le troisième pod est bien intégré au service. Nous voyons apparaître la page par défaut du serveur web, ce qui signifie que nous avons bien accès via Load Balancing à notre troisième serveur web, en plus des deux premiers.

Exercice 2 : NodePort

Créer un déploiement de conteneur apache. Ce déploiement devra respecter les contraintes suivantes :

- nom du déploiement : deploy-node-port
- replicas : 2
- namespace : services
- selecteur :
  - matchLabels: app => node-port
- template :
  - label : app => node-port
  - conteneur :
    - nom : node-port-apache
    - image : httpd

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-node-port
  namespace: services
```

```
spec:
  replicas: 2
  selector:
    matchLabels:
      app: node-port
  template:
    metadata:
      labels:
        app: node-port
    spec:
      containers:
      - name: node-port-apache
        image: httpd
```

```
$ kubectl apply -f deploy-node-port.yml
```

Vérifier que les pods fonctionnent normalement et repérer les nœuds sur lesquels ils tournent.

```
$ kubectl get pods -o wide -n services
```

Créer maintenant le service suivant :

- nom du service : svc-node-port
- namespace : services
- type : NodePort
- NodePort : 30001
- port d'écoute : 8001
- port utilisé par le pod : 80
- selecteur : app => node-port

```
---
kind: Service
apiVersion: v1
metadata:
  name: svc-node-port
  namespace: services
spec:
  type: NodePort
  ports:
  - nodePort: 30001
    port: 8001
    targetPort: 80
  selector:
    app: node-port
```

```
$ kubectl apply -f svc-node-port.yml
```

Vérifier l'état du service.

```
$ kubectl get svc -n services
```

Se connecter au pod du worker1, installer vim à l'aide de la commande apt et remplacer le fichier /usr/local/apache2/htdocs/index.html par le suivant :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Worker1</title>
  </head>
  <body>
    <h1>Vous etes sur Worker1</h1>
  </body>
</html>
```

```
$ kubectl exec -ti -n services deploy-node-port-$podId -- bash
$ apt update && apt install -y vim
```

Faire la même chose sur le pod tournant sur worker2, et utiliser le fichier suivant :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Worker2</title>
  </head>
  <body>
    <h1>Vous etes sur Worker2</h1>
  </body>
</html>
```

```
$ kubectl exec -ti -n services deploy-node-port-$podId -- bash
$ apt update && apt install -y vim
```

Récupérer l'ip du master.

```
$ ip -c -br a
```

Comme tout à l'heure, lancer des curls consécutifs sur l'ip du master et le port présent dans la configuration du service. Que constatez-vous ?

```
$ curl 192.168.56.31:30001
```

Nous arrivons bien à accéder à nos pods par le port que nous avons défini. Une nouvelle fois, un load balancing a été créé.

Supprimer le namespace services.

```
$ kubectl delete ns services
```

Correction – Partie 6 : Les Daemonsets

Exercice 1 :

Créer un nouveau namespace daemon-set.

```
$ kubectl create ns daemon-set
```

Adapter le fichier deploy-daemon.yml, à partir du fichier deploy-daemon.yml présent dans le répertoire des sources, et respecter les contraintes suivantes :

- nom du déploiement : deploy-daemon
- namespace : daemon-set
- selecteur :
  - matchLabels: app => my-centos
- template :
  - label : app => my-centos
  - conteneur :
    - nom : centos
    - image : centos
    - commande : sleep de 600 secondes

Ajouter dans la section spec, la section NodeSelector avec comme mot-clef work avec la valeur « true ».

```
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: deploy-daemon
  namespace: daemon-set
spec:
  selector:
    matchLabels:
      app: my-centos
  template:
    metadata:
      labels:
        app: my-centos
    spec:
      nodeSelector:
        work: "true"
      containers:
      - name: centos
        image: centos
        command: ["sleep","600"]
```

Attention aux guillemets autour de true ! Les fichiers sont en yaml, et true fait partie des valeurs interprétées (booléen). Il faut donc le mettre entre guillemets pour forcer la chaîne de caractère.

```
$ kubectl apply -f deploy-daemon.yml
```

Afficher les informations du daemonset. Que constatez-vous ?

```
$ kubectl get daemonset -n daemon-set
NAME     DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
daemon   0         0         0       0            0           work=true       2m58s
```

On constate qu'aucun pod n'a été créé. Ce qui est normal, puisque nous n'avons pas donné le label nécessaire sur nos workers.

Mettre en place le label work=true sur le worker2, puis vérifier l'état du daemonset, ainsi que celui des éventuels pods créés.

```
$ kubectl label nodes worker2 work=true
$ kubectl get daemonset -n daemon-set
NAME     DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
daemon   1         1         1       1            1           work=true       5m45s
$ kubectl get pods -o wide -n daemon-set
NAME           READY   STATUS    RESTARTS   AGE   IP             NODE           NOMINATED
NODE   READINESS GATES
daemon-8dmkl   1/1     Running   0          78s   10.244.2.22   worker2.form   <none>
<none>
```

Un pod a été créé automatiquement sur worker2, grâce au daemonset.

Supprimer maintenant la section « NodeSelector »du fichier YAML, puis appliquez la configuration. Que constatez-vous ?

```
$ kubectl apply -f deploy-daemon.yml
$ kubectl get daemonsets.apps -n daemon-set
NAME     DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
daemon   2         2         2       1            2           <none>          10m
$ kubectl get pods -o wide -n daemon-set
NAME           READY   STATUS    RESTARTS   AGE   IP             NODE           NOMINATED
NODE   READINESS GATES
daemon-mb5b5   1/1     Running   0          6s    10.244.2.23   worker2.form   <none>
<none>
daemon-rmfng   1/1     Running   0          46s   10.244.1.19   worker1.form   <none>
<none>
```

Le daemonset a automatiquement déployé un pod sur chaque nœud. La contrainte que nous avions placé ayant été retiré, il s'agit du comportement attendu.

Supprimer le namespace daemon-set.

```
$ kubectl delete ns daemon-set
```

Correction : Partie 7 : Les Volumes

Exercice 1 : HostPath

Créer le namespace volumes.

```
$ kubectl create ns volumes
```

Créer un répertoire /srv/hostpath sur chaque worker.

```
$ ssh worker1 sudo mkdir /srv/hostpath
$ ssh worker2 sudo mkdir /srv/hostpath
```

Changer le propriétaire du répertoire par user1.

```
$ ssh worker1 sudo chown user1: /srv/hostpath
$ ssh worker2 sudo chown user1: /srv/hostpath
```

Créer les fichiers index.html avec le contenu suivant, respectivement sur worker1 et worker2 :

```
<h1>Je suis le HostPath de Worker1</h1>
```

```
<h1>Je suis le HostPath de Worker2</h1>
```

Créer un nouveau déploiement, en adaptant le fichier deploy-host-path.yml, présent dans le répertoire source, et en y implément un volume de type hostPath, à l'aide du répertoire /srv/hostpath précédemment créé sur worker1 et worker2 :

- nom du déploiement : deploy-host-path
- namespace : volumes
- selecteur :
  - matchLabels: app => host-path
- template :
  - label : app => host-path
  - conteneur :
    - nom : host-path-nginx
    - image : nginx
  - NodeName : worker1
  - volume :
    - nom : mon-host-path
    - point de montage : /usr/share/nginx/html

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-host-path
  namespace: volumes
spec:
  selector:
    matchLabels:
      app: host-path
  template:
    metadata:
      labels:
        app: host-path
    spec:
      containers:
      - name: host-path-nginx
        image: nginx
        volumeMounts:
        - mountPath: /usr/share/nginx/html
          name: mon-host-path
      nodeName: worker1
      volumes:
      - name: mon-host-path
        hostPath:
          path: /srv/hostpath
          type: Directory
```

```
$ kubectl apply -f deploy-host-path.yml
```

Vérifier que le pod créé fonctionne correctement, et noter son ip ainsi que le nœud sur lequel il tourne.

```
$ kubectl get pods -o wide -n volumes
```

Lancer un curl sur le serveur web. Que constatez-vous ?

```
# curl 10.0.235.147
```

C'est bien le répertoire correspondant au nœud sur lequel tourne notre pod qui est monté.

Modifier le fichier de configuration pour que le pod tourne sur l'autre nœud (à l'aide de la directive « nodeName » vue précédemment). Récupérer l'ip du nouveau pod et lancer de nouveau un curl. Que constatez-vous ?

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-host-path
  namespace: volumes
spec:
  selector:
    matchLabels:
      app: host-path
  template:
    metadata:
      labels:
        app: host-path
    spec:
      containers:
```

```
    - name: host-path-nginx
      image: nginx
      volumeMounts:
      - mountPath: /usr/share/nginx/html
        name: mon-host-path
    nodeName: worker2
    volumes:
    - name: mon-host-path
      hostPath:
        path: /srv/hostpath
        type: Directory
```

```
$ kubectl apply -f deploy-host-path.yml
```

Cette fois, nous constatons qu'il s'agit de l'autre répertoire qui est monté, ce qui est bien le comportement attendu.

Supprimer le déploiement.

```
$ kubectl delete deploy -n volumes deploy-host-path
```

Exercice 2 : Persistent Volume Claim

Pour cet exercice, nous allons installer un serveur nfs sur le master.
Installer sur le master le paquet nfs-kernel-server.

```
$ sudo apt install -y nfs-kernel-server
```

Installer sur les workers le paquet nfs-common (il devrait déjà être présent).

```
$ ssh worker1 sudo apt install -y nfs-common
$ ssh worker2 sudo apt install -y nfs-common
```

Créer le répertoire /srv/exports sur le master et changer le propriétaire par user1.

```
$ sudo mkdir /srv/exports
$ sudo chown user1: /srv/exports
```

Éditer le fichier de configuration du serveur nfs /etc/exports et insérer la ligne suivante (sur le master):

```
/srv/exports 192.168.56.0/24(rw,sync,no_root_squash)
```

Activer et redémarrer le serveur nfs.

```
$ sudo systemctl enable nfs-kernel-server
$ sudo systemctl restart nfs-kernel-server
```

Vérifier que le serveur NFS est opérationnel :

```
$ showmount -e
Export list for master :
/srv/exports        192.168.56.0/24
```

Nous avons maintenant un serveur NFS opérationnel sur le master.

Créer maintenant un PersistentVolume en adaptant le fichier pv-nfs.yml, présent dans le répertoire des sources, avec les contraintes suivantes :

- nom : pv-nfs
- namespace : volumes
- storageClass : manual
- capacity : 1G
- accessModes : ReadWriteMany
- nfs :
  - server : adresse de votre serveur nfs
  - path : chemin du partage

```
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-nfs
  namespace: volumes
spec:
  storageClassName: manual
  capacity:
    storage: 1G
  accessModes:
    - ReadWriteMany
  nfs:
    server: 192.168.56.31
    path: "/srv/exports"
```

Appliquer la configuration.

```
$ kubectl apply -f pv-nfs.yml
```

Vérifier l'état du PersistentVolume.

```
$ kubectl get pv -n volumes
```

Créer maintenant le PersistentVolumeClaim associé,-en adaptant le fichier pvc-nfs.yml, présent dans le répertoire des sources, pour lui affecter 500 Mo :

```
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-nfs
  namespace: volumes
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 500M
```

Appliquer la configuratoin.

```
$ kubectl apply -f pvc-nfs.yml
```

Vérifier que le PersistentVolumeClaim a été correctement créé.

```
$ kubectl get pvc -n volumes
```

Sur le master, créer le fichier /srv/exports/index.html, avec le contenu suivant :

```
<h1>Bienvenue sur mon site web.</h1>
```

Créer maintenant 2 déploiements :

- Le premier sur base d'image nginx, avec le volume monté dans /usr/share/nginx/html, en le faisant tourner sur worker1.
- Le deuxième sur base d'image httpd, avec le volume monté dans /usr/local/apache2/htdocs, en le faisant tourner sur worker2.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-nfs-nginx
  namespace: volumes
spec:
  selector:
    matchLabels:
      app: nfs-nginx
  template:
    metadata:
      labels:
        app: nfs-nginx
    spec:
      containers:
      - name: nfs-nginx
        image: nginx
        volumeMounts:
        - mountPath: /usr/share/nginx/html
          name: www
      nodeName: worker1
      volumes:
      - name: www
        persistentVolumeClaim:
          claimName: pvc-nfs
```

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-nfs-httpd
  namespace: volumes
spec:
  selector:
    matchLabels:
      app: nfs-httpd
  template:
    metadata:
      labels:
        app: nfs-httpd
    spec:
      containers:
      - name: nfs-httpd
        image: httpd
        volumeMounts:
        - mountPath: /usr/local/apache2/htdocs
          name: www
      nodeName: worker2
      volumes:
      - name: www
        persistentVolumeClaim:
          claimName: pvc-nfs
```

```
$ kubectl apply -f deploy-nfs-nginx.yml
$ kubectl apply -f deploy-nfs-httpd.yml
```

Vérifier que les pods fonctionnent correctement et respectent bien les contraintes.
Noter leurs adresses ip respectives.

```
$ kubectl get pods -o wide -n volumes
```

Lancer un curl sur les 2 serveurs web. Que constatez-vous ?

```
# curl 10.0.189.92
# curl 10.0.235.150
```

Les deux serveurs web utilisent bien le même fichier html.

Modifier le fichier html pour qu'il affiche « Bonjour et Bienvenue » sur le master. Relancer les curls.
Que constatez-vous ?
Les modifications sont bien prises en compte.

Exercice 3 : ConfigMaps et Secrets

Adapter le fichier de configuration conf-mariadb.yml, présent dans le répertoire des sources, pour créer le configMap suivant :
nom : conf-mariadb
namespace : volumes
data :
MYSQL_DATABASE : dbweb
MYSQL_USER : web

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: conf-mariadb
  namespace: volumes
data:
  MYSQL_DATABASE: dbweb
  MYSQL_USER: web
```

```
$ kubectl apply -f conf-mariadb.yml
```

Vérifier que le configMap a été correctement créé, et qu'il contient les bonnes variables.

```
$ kubectl get configmaps -n volumes
$ kubectl describe configmaps -n volumes conf-mariadb
```

Adapter le fichier de configuration secret-mariadb.yml, présent dans le répertoire des sources, pour créer le Secret suivant :
nom : secret-mariadb
namespace : volumes
data :
MYSQL_ROOT_PASSWORD: MonSuperMdpRoot
MYSQL_PASSWORD : MonSuperMdp

```
$ echo -n "MonSuperMdpRoot"|base64
TW9uU3VwZXJNZHBSb290
$ echo -n "MonSuperMdp"|base64
TW9uU3VwZXJNZHA=
```

```
---
apiVersion: v1
kind: Secret
metadata:
  name: secret-mariadb
  namespace: volumes
data:
  MYSQL_ROOT_PASSWORD: TW9uU3VwZXJNZHBSb290
  MYSQL_PASSWORD: TW9uU3VwZXJNZHA=
```

```
$ kubectl apply -f secret-mariadb.yml
```

Vérifier.

```
$ kubectl get secrets -n volumes
$ kubectl describe secrets -n volumes secret-mariadb
```

Créer maintenant un déploiement, en adaptant le fichier deploy-ma-db.yml, présent dans le répertoire des sources, et basé sur une image mariadb. Passer dans l'environnement du conteneur le configmap et le secret précédemment créés.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-ma-db
  namespace: volumes
spec:
  selector:
    matchLabels:
      app: ma-db
  template:
    metadata:
      labels:
        app: ma-db
    spec:
      containers:
      - name: ma-db
        image: mariadb
        env:
        - name: MYSQL_DATABASE
          valueFrom:
            configMapKeyRef:
              name: conf-mariadb
              key: MYSQL_DATABASE
        - name: MYSQL_USER
          valueFrom:
            configMapKeyRef:
              name: conf-mariadb
              key: MYSQL_USER
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: secret-mariadb
              key: MYSQL_ROOT_PASSWORD
        - name: MYSQL_PASSWORD
          valueFrom:
            secretKeyRef:
              name: secret-mariadb
              key: MYSQL_PASSWORD
```

Appliquer la configuration.

```
$ kubectl apply -f deploy-ma-db.yml
```

Vérifier que le déploiement s'est correctement déroulé.

```
$ kubectl get deployments.apps -n volumes
```

Se connecter au conteneur.

```
$ kubectl exec -ti -n volumes deploy-ma-db-6879997db6-l9sw5 -- bash
```

Depuis le conteneur, se connecter à l'instance mariadb en utilisant la commande du même nom, en root et en user web, et en saisissant les mots de passes créés plus tôt.

```
# mariadb -u root -p
# mariadb -u web -p
```

Au prompt de mariadb, vérifier que la database dbweb a bien été créée à l'aide de la commande « show databases ; », puis quitter à l'aide de la commande « quit ; ».

```
MariaDB [(none)]> show databases;
+--------------------+
| Database           |
+--------------------+
| dbweb              |
| information_schema |
| mysql              |
| performance_schema |
+--------------------+
4 rows in set (0.000 sec)
MariaDB [(none)]> quit
```

Se déconnecter du pod

```
# <CTRL-D>
```

Supprimer le namespace volumes.

```
$ kubectl delete ns volumes
```

Correction – Partie 8 : Sécurité

Exercice 1 : RBAC

Créer le namespace rbac.

```
$ kubectl create ns rbac
```

Créer un nouveau Rôle en adaptant le fichier de configuration my-rbac-list.yml, présent dans le répertoire des sources, avec les contraintes suivantes :

- nom : my-rbac-list
- namespace : rbac
- règles :
  - apiGroup : group core
  - ressources : pods
  - opérations : list

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: rbac
  name: my-rbac-list
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["list"]
```

Appliquer la configuration. Vérifier avec les commandes adéquates.

```
$ kubectl apply -f my-rbac-list.yml
$ kubectl get role.rbac -n rbac
$ kubectl describe role.rbac -n rbac my-rbac-list
```

Créer un nouveau Rôle, nommé my-rbac-all, en adaptant le fichier my-rbac-all1.yml, présent dans le répertoire des sources, avec les mêmes contraintes que précédemment , mais en lui donnant cette fois-ci les opération get, watch et list.

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: rbac
  name: my-rbac-all
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get","watch","list"]
```

Appliquer la configuration. Vérifier avec les commandes adéquates.

```
$ kubectl apply -f my-rbac-all1.yml
$ kubectl get role.rbac -n rbac
$ kubectl describe role.rbac -n rbac my-rbac-all
```

Créer 2 servicesAccounts, user-list et user-all, dans le namespace rbac en adpatant les fichiers user-list.yml et user-all.yml, présents dans les répertoires des sources.
Le premier servira pour le Rôle avec l'opération list, le second pour celui avec les 3 opérations, list, wet et watch.

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: user-list
  namespace: rbac
```

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: user-all
  namespace: rbac
```

Appliquer les fichiers.

```
$ kubectl apply -f user-list.yml
$ kubectl apply -f user-all.yml
```

Vérifier que les Rôles et ServiceAccounts ont été correctement créés.

```
$ kubectl get roles -n rbac
$ kubectl get sa -n rbac
```

Adapter les fichiers role-binding-list.yml et role-binding-all.yml, présents dans le répertoire des sources, afin de créer 2 RoleBinding. Ces RoleBinding serviront à lier les ServiceAccount et les Roles précédemment créés :

- role-binding-all = ServiceAccount : user-all, Role : my-rbac-all
- role-binding-list = ServiceAccount : user-list, Role : my-rbac-list

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: role-binding-all
  namespace: rbac
subjects:
- kind: ServiceAccount
  name: user-all
  namespace: rbac
roleRef:
  kind: Role
  name: my-rbac-all
  apiGroup: rbac.authorization.k8s.io
```

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: role-binding-list
  namespace: rbac
subjects:
- kind: ServiceAccount
  name: user-list
  namespace: rbac
roleRef:
  kind: Role
  name: my-rbac-list
  apiGroup: rbac.authorization.k8s.io
```

Appliquer la configuration.

```
$ kubectl apply -f role-binding-list.yml
$ kubectl apply -f role-binding-all.yml
```

Vérifier que les RoleBindings ont été correctement créé et qu'ils respectent bien les contraintes.

```
$ kubectl get rolebindings -n rbac -o wide
```

Récupérer le nom des tokens générés lors de la création des serviceAccounts.

```
$ kubectl describe -n rbac sa user-list
$ kubectl describe -n rbac sa user-all
```

Créer maintenant un déploiement, basé sur une image Debian, en lui fournissant le serviceAccount user-list. Le token associé à ce ServiceAccount sera monté en tant que volume dans /etc/secrets. Nous n'oublierons pas de spécifier une commande, afin de garder le pod up.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-debian-list
  namespace: rbac
spec:
  selector:
    matchLabels:
      app: list
  template:
    metadata:
      labels:
        app: list
    spec:
      serviceAccountName: user-list
      containers:
      - name: my-debian
        image: debian
        command: ["sleep","600"]
        volumeMounts:
        - mountPath: /etc/secrets
          name: user-list-token
      volumes:
      - name: user-list-token
        projected:
          sources:
```

```
            - serviceAccountToken:
                path: user-list-token
                expirationSeconds: 7200
                audience: https://kubernetes.default.svc.cluster.local
            - configMap:
                items:
                  - key: ca.crt
                    path: ca.crt
                name: kube-root-ca.crt
```

```
$ kubectl apply -f deploy-list.yml
```

Créer maintenant un deuxième déploiement, basé sur une image Ubuntu, en lui fournissant cette fois-ci le serviceAccount user-all, et son token. Encore une fois, il nous faudra utiliser une commande pour que notre pod reste up.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-ubuntu-all
  namespace: rbac
spec:
  selector:
    matchLabels:
      app: all
  template:
    metadata:
      labels:
        app: all
    spec:
      serviceAccountName: user-all
      containers:
      - name: my-ubuntu
        image: ubuntu
        command: ["sleep","600"]
        volumeMounts:
        - mountPath: /etc/secrets
          name: user-all-token
      volumes:
      - name: user-all-token
        projected:
          sources:
            - serviceAccountToken:
                path: user-all-token
                expirationSeconds: 7200
                audience: https://kubernetes.default.svc.cluster.local
            - configMap:
                items:
                  - key: ca.crt
                    path: ca.crt
                name: kube-root-ca.crt
```

```
$ kubectl apply -f deploy-all.yml
```

Vérifier le bon fonctionnement des déploiements.

```
$ kubectl get deploy -n rbac
```

Afficher les pods.

```
$ kubectl get pods -n rbac
```

Se connecter sur le pod debian.

```
$  kubectl exec -ti -n rbac my-debian-list-$podId -- bash
```

Vérifier la présence des secrets.

```
# ls -l /etc/secrets/
```

Installer le paquet curl à l'aide de la commande apt.

```
# apt update && apt install -y curl
```

Accéder à l'api via curl et les secrets, et afficher la liste des pods à l'aide des commandes suivantes :

```
# token=$(cat /etc/secrets/user-list-token)
# curl -H "Authorization: Bearer $token" --cacert \
/etc/secrets/ca.crt \
https://kubernetes.default/api/v1/namespaces/rbac/pods
```

Essayer maintenant d'obtenir les informations d'un des pods du namespace, ainsi que ces évènements. Que constatez-vous ?

```
# curl -H "Authorization: Bearer $token" --cacert /etc/secrets/ca.crt \
https://kubernetes.default/api/v1/namespaces/rbac/pods/my-debian-list-
$podId
# curl -H "Authorization: Bearer $token" --cacert /etc/secrets/ca.crt \
https://kubernetes.default/api/v1/watch/namespaces/rbac/pods/my-debian-
list-$podId
```

Le pod ne peut accéder à ces informations. Ce qui est normal puisque nous n'avons pas donné cette opération dans les opérations permises au Rôle qui est lié à notre ServiceAccount.

Se déconnecter du pod.

```
# <CTRL-D>
```

Répéter les mêmes opérations sur le pod Ubuntu. Que constatez-vous ?

```
$ kubectl exec -ti -n rbac my-ubuntu-all-$podId -- bash
# apt update && apt install -y curl
# token=$(cat /etc/secrets/user-all-token)
# curl -H "Authorization: Bearer $token" --cacert /etc/secrets/ca.crt \
https://kubernetes.default/api/v1/namespaces/rbac/pods/
# curl -H "Authorization: Bearer $token" --cacert /etc/secrets/ca.crt \
https://kubernetes.default/api/v1/namespaces/rbac/pods/my-debian-list-
$podId
# curl -H "Authorization: Bearer $token" --cacert /etc/secrets/ca.crt \
https://kubernetes.default/api/v1/watch/namespaces/rbac/pods/my-debian-
list-$podId
```

Cette fois-ci, le pod arrive bien à accéder à l'api et afficher les informations demandées.

Editer le fichier de configuration du rôle my-rbac-all et retirer l'opération watch.

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: rbac
  name: my-rbac-all
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get","list"]
```

Appliquer les changements et vérifier l'accès à l'API depuis le pod. Que constatez-vous ?

```
$ kubectl apply -f my-rbac-all2.yml
$ kubectl exec -ti -n rbac my-ubuntu-all-$podId -- bash
# apt update && apt install curl
# token=$(cat /etc/secrets/token)
# curl -H "Authorization: Bearer $token" --cacert /etc/secrets/ca.crt \
https://kubernetes.default/api/v1/watch/namespaces/rbac/pods/my-debian-list-$podId
```

Nous n'avons plus accès à cet appel, les modifications sont donc prises directement à chaud.

Supprimer le namespace rbac.

```
$ kubectl delete ns rbac
```

Exercice 2 : Les quotas

Créer le namespace quotas.

```
$ kubectl create ns quotas
```

Adapter le fichier de configuration ma-limite.yml présent dans le répertoire des sources, pour créer une ressource LimitRange, avec les limitations suivantes :

- nom : ma-limite
- namespace : quotas
- type : pod
  - max :
  - cpu : 1
  - ram : 100Mo
- type : container
  - cpu : 0.5
  - ram : 50Mo
  - defaultRequest :
    - cpu : 0.25
    - ram : 25Mo
  - maxLimitRequestRatio :
    - cpu : 4
    - ram : 2

```
---
apiVersion: v1
kind: LimitRange
metadata:
   name: ma-limite
   namespace: quotas
spec:
   limits:
     - type: Pod
       max:
         cpu: 1
         memory: 100M
     - type: Container
       max:
         cpu: 0.5
         memory: 50M
       defaultRequest:
         cpu: 0.25
         memory: 25M
       maxLimitRequestRatio:
         cpu: 4
         memory: 2
```

Appliquer la configuration.

```
$ kubectl apply -f ma-limite.yml
```

Vérifier que la ressource a été correctement créée.

```
$ kubectl get limitranges -n quotas
```

Consulter le fichier deploy-my-nginx.yml, présent dans le répertoire des sources, puis appliquer la configuration.

```
$ kubectl apply -f deploy-my-nginx.yml
```

Vérifier les quotas appliqués sur le conteneur du pod déployé. Pourquoi ces quotas ?

```
$ kubectl describe pod -n quotas deploy-my-nginx-$podId
```

Le pod respecte bien les quotas que nous lui avons appliqués via la ressource LimitRange, à savoir en request les cpus à 0,25 et en mémoire 25Mo. Concernant les limites, là encore nous avons bien ce que la ressource prévoit : 50Mo de mémoire et 0.5 cpu.

Appliquer des requests cpu et mémoire personnalisées dans le fichier de configuration de la manière suivante, et relancer le déploiement :
cpu : 2
ram : 150Mo

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-my-nginx
  namespace: quotas
spec:
  selector:
    matchLabels:
      app: limite
  template:
    metadata:
      labels:
        app: limite
    spec:
      containers:
      - name: my-nginx
        image: nginx
        resources:
          requests:
            cpu: 2
            memory: 150M
```

```
$ kubectl apply -f deploy-my-nginx2.yml
```

Vérifier à nouveau l'état du conteneur de pod déployé, ainsi que l'état du déploiement. Que constatez-vous ?

```
$ kubectl describe pod -n quotas deploy-my-nginx-$podId
$ kubectl describe deploy -n quotas deploy-my-nginx
```

Le pod n'est pas déployé, car les requests du conteneur dépassent les limites imposées par la ressource LimitRange.

Supprimer le déploiement ainsi que la ressource LimitRange.

```
$ kubectl delete deploy -n quotas deploy-my-nginx
$ kubectl delete limitranges -n quotas ma-limite
```

Exercice 3 : ResourceQuota

Modifier le fichier de configuration deploy-rs-quotas.yml pour déployer une ressource ResourceQuota, en demandant un maximum de 4 pods, et l'utilisation de 2 cpus dans le namespace quotas.

```
---
apiVersion: v1
kind: ResourceQuota
metadata:
  name: rs-quotas
  namespace: quotas
spec:
  hard:
    pods: 4
    cpu: 2
```

Appliquer la configuration.

```
$ kubectl apply -f rs-quotas.yml
```

Créer maintenant un nouveau déploiement, avec les contraintes suivantes :

- nom : deploy-rs-quotas
- namespace : quotas
- replicas : 2
- selecteur :
  - matchLabels: app => rs-quotas
- template :
  - label : app => rs-quotas
  - conteneur :
    - nom : apache-rs-quota
    - image : httpd
    - requests :
      - cpu : 1
      - memory : 100M

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-rs-quotas
  namespace: quotas
spec:
  replicas: 2
  selector:
    matchLabels:
      app: rs-quotas
  template:
    metadata:
      labels:
        app: rs-quotas
    spec:
      containers:
      - name: apache-rs-quotas
        image: httpd
        resources:
          requests:
            cpu: 1
            memory: 100M
```

Appliquer la configuration.

```
$ kubectl apply -f deploy-rs-quotas.yml
```

Vérifier que le déploiement c'est correctement déroulé.

```
$ kubectl get deploy -n quotas
$ kubectl get pods -n quotas
```

Passer maintenant le nombre de réplicas à 4, en ligne de commande.

```
$ kubectl scale deployment -n quotas --replicas=4 deploy-rs-quotas
```

Vérifier l'état du déploiement. Que ce passe-t-il ? Pourquoi ?

```
$ kubectl get pods -n quotas
$ kubectl get deploy -n quotas
$ kubectl describe deploy -n quotas deploy-rs-quotas
```

Le déploiement a bien pris en compte notre demande de scale, mais ne l'a pas appliqué. En effet, la ressource ResourceQuota nous limite à 4 pods, ce qui n'est pas bloquant, mais également à 2 cpus. Hors, nous avons demandé 4 pods, qui contienne chacun 1 conteneur avec 1 cpu en requests. Nous dépassons donc la limite imposée.

Supprimer le namespace quotas.

```
$ kubectl delete ns quotas
```

Exercice 4 : Accès réseaux

Créer le namespace frontend.

```
$ kubectl create ns frontend
```

Créer le namespace backend.

```
$ kubectl create ns backend
```

Affecter le label type=backend au namespace backend.

```
$ kubectl label ns backend type=backend
```

Donner le label type=frontend au namespace frontend.

```
$ kubectl label ns frontend type=frontend
```

Afficher le fichier deploy-my-nginx.yml, présent dans le répertoire des sources, et vérifier qu'il respecte les contraintes suivantes :

- nom : deploy-my-nginx
- namespace : frontend
- selecteur :
  - matchLabels: app => frontend
- template :
  - label : app => frontend,auth=db
  - conteneur :
    - nom : my-nginx
    - image : nginx

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-my-nginx
  namespace: frontend
spec:
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
        auth: db
    spec:
      containers:
      - name: my-nginx
        image: nginx
```

Appliquer la configuration.

```
$ kubectl apply -f deploy-my-nginx.yml
```

Vérifier que le déploiement s'est correctement déroulé.

```
$ kubectl get deploy -n frontend
$ kubectl get pods -n frontend
```

Créer également le service nodePort associé en modifiant le fichier svc-my-nginx.yml présent dans le répertoire des sources, afin de pouvoir accéder au serveur web via le port 30001 de l'ip publique du master. Nous utiliserons le port 8001 comme port interne, et le port destination sur le pod est le port 80.

```
---
kind: Service
apiVersion: v1
metadata:
  name: svc-my-nginx
  namespace: frontend
spec:
  type: NodePort
  ports:
  - nodePort: 30001
    port: 8001
    targetPort: 80
  selector:
    app: frontend
```

Appliquer la configuration.

```
$ kubectl apply -f svc-my-nginx.yml
```

Vérifier que le service fonctionne correctement.

```
$ curl 192.168.56.31:30001
```

Afficher le fichier deploy-my-db.yml, présent dans le répertoire des sources, et vérifier qu'il respecte les contraintes suivantes :

- nom : deploy-my-db
- namespace : backend
- selecteur :
  - matchLabels: app => backend
- template :
  - label : app => backend
  - conteneur :
    - nom : my-db
    - image : mariadb
- env:
  - name: MYSQL_DATABASE
  - value: webdb
  - name: MYSQL_USER
  - value: webuser
  - name: MYSQL_ROOT_PASSWORD
  - value: mdpRoot
  - name: MYSQL_PASSWORD
  - value: mdp

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-my-db
  namespace: backend
spec:
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
      - name: my-db
        image: mariadb
        env:
        - name: MYSQL_DATABASE
          value: webdb
        - name: MYSQL_USER
          value: webuser
        - name: MYSQL_ROOT_PASSWORD
          value: mdpRoot
        - name: MYSQL_PASSWORD
          value: mdp
```

Appliquer la configuration.

```
$ kubectl apply -f deploy-my-db.yml
```

Vérifier que le déploiement fonctionne correctement. Récupérer l'adresse ip du conteneur my-db

```
$ kubectl get deploy -n backend
$ kubectl get pods -n backend -o wide
```

Se connecter au conteneur nginx à l'aide de la commande suivante :

```
$ kubectl exec -it -n frontend deploy-my-nginx-$podId-- bash
```

Effectuer une mise à jour du dépôt et installer les paquets mycli et iptutils-ping.

```
# apt update && apt install -y mycli iputils-ping
```

Vérifier que la connexion fonctionne sur la base depuis le conteneur nginx

```
# apt update && apt install -y mycli iputils-ping
# mycli -u webuser -h 10.0.235.158 webdb
```

Se déconnecter du pod.

```
# <CTRL-D>
```

Afficher le fichier deploy-my-api.yml, présent dans le répertoire des sources, et vérifier qu'il respecte les contraintes suivantes :

- nom : deploy-my-api
- namespace : frontend
- selecteur :
  - matchLabels: app => api
- template :
  - label : app => api
  - conteneur :
    - nom : my-api
    - image : debian
    - command : sleep 600

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-my-api
  namespace: frontend
spec:
  selector:
    matchLabels:
      app: api
  template:
    metadata:
      labels:
        app: api
    spec:
      containers:
      - name: my-api
        image: debian
        command: ["sleep","600"]
```

Appliquer la configuration.

```
$ kubectl apply -f deploy-my-api.yml
```

Vérifier une nouvelle fois que le déploiement a réussi.

```
$ kubectl get deploy -n frontend
$ kubectl get pods -n frontend
```

Se connecter sur le conteneur du pod my-api avec la commande suivante :

```
$ kubectl exec -ti -n frontend deploy-my-api-$podId -- bash
```

Effectuer une mise à jour et installer le paquet iputilis-ping.

```
# apt update && apt install -y iputils-ping
```

Executer un ping sur les conteneurs nginx et mariadb, ainsi que sur google.fr.

```
# ping $ipConteneurNginx
# ping  $ipConteneurMariadb
# ping google.fr
```

Modifier le fichier api-net-policy.yml, présent dans le répertoire des sources, pour mettre en place une police réseau afin de respecter les contraintes suivantes :

- appliquer sur : deploy-my-nginx
- connexions entrantes : uniquement depuis les pods du namespace ayant le label app = api sur le port 80 en TCP
- connexions sortantes : uniquement vers les pods ayant un label app=backend dans le namespace ayant le label type=backend sur le port 3306 en TCP.

```
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: api-net-policy
  namespace: frontend
spec:
  podSelector:
    matchLabels:
      app: frontend
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: api
    ports:
    - protocol: TCP
      port: 80
  egress:
  - to:
    - namespaceSelector:
        matchLabels:
          type: backend
    ports:
    - protocol: TCP
      port: 3306
```

Appliquer la configuration.

```
$ kubectl apply -f api-net-policy.yml
```

Essayer d'exécuter la commande curl sur l'ip du master sur le port 30001 du service. Que constatez-vous ?

```
$ curl 192.168.56.31:30001
```

Le curl ne fonctionne plus.

Se connecter sur le conteneur my-api à l'aide de la commande suivante :

```
$ kubectl exec -it -n frontend deploy-my-api-$podId-- bash
```

Installer le paquet curl (mettre à jour la liste des paquets si nécessaire).

```
# apt udpate && apt install -y curl
```

Exécuter un curl sur l'adresse ip du conteneur nginx. Que constatez vous ?

```
# curl $ipConteneurNginx
```

Le curl fonctionne depuis le conteneur my-api.

Se déconnecter du conteneur.

```
# <CTRL-D>
```

Re-executer la commande permettant de se connecter à la base de données du conteneur mariadb. Essayer également d'executer un ping sur l'adresse ip du conteneur mariadb. Que constatez-vous ?

```
# mycli -h  $ipConteneurBDD -u webuser
# ping $ipConteneurBDD
```

La connexion à la base fonctionne correctement. Le ping ne passe pas.

Se déconnecter du conteneur.

```
# <CTRL-D>
```

Mettre à jour le fichier mariadb-net-policy.yml, présent dans le répertoire des sources, afin de respecter les contraintes suivantes :

- appliquer sur : my-mariadb
- connexions entrantes : uniquement depuis les pods ayant un label auth=db, quelque soit le namespace
- connexions sortantes : aucunes

```
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: mariadb-net-policy
  namespace: backend
spec:
  podSelector:
    matchLabels:
      app: backend
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          type: frontend
    - podSelector:
        matchLabels:
          auth: db
```

Appliquer la configuration.

```
$ kubectl apply -f mariadb-net-policy.yml
```

Se connecter au conteneur mariadb à l'aide de la commande suivante :

```
$ kubectl exec -it -n backend deploy-my-db-$podId-- bash
```

Exécuter la commande « apt update ». Constater que cela ne fonctionne pas.

```
# apt update
```

Nous avons interdit les connexions sortantes, la commande échoue donc.

Se déconnecter du conteneur.

```
# <CTRL-D>
```

Se connecter sur le conteneur nginx et se connecter via mycli sur la base. Que constatez-vous ?

```
$ kubectl exec -ti -n frontend my-nginx-$podId -- bash
# mycli -h $ipConteneurBDD -u webuser
```

La connexion à la base fonctionne toujours normalement, puisque my-nginx est maintenant le seul pod à y avoir accès.

Se déconnecter du conteneur.

```
# <CTRL-D>
```

Supprimer les namespaces backend et frontend.

```
$ kubectl delete ns backend frontend
```