Lab 6 : Conserver les données

Les données des `Pods` sont volatibles, c'est-à-dire qu'à chaque destruction d'un `Pod` toutes les données qui ont pu être générées seront perdues. Les types données qui sont gérés par un `Pod` peuvent être des données applicatives (issues d'une base de données), des logs, des fichiers de configuration, des fichiers de partage, etc. Nous adressons la problématique suivante dans cet exercice : comment s'assurer que si un `Pod` est recréé, les données précédentes soient restaurées. Pour y répondre, nous introduisons le concept de `Volume`. Un `Volume` représente un espace de stockage contenant des données et accessible à travers plusieurs conteneurs d'un même `Pod`ou de `Pods` différents. Différents types de `Volume` existent selon le besoin de stockage à gérer. Nous étudierons les `Volumes` de type `hostPath` et `emptyDir` pour l'accès à un stockage local (accessible uniquement dans un même nœud) puis `NFS` pour des `Volumes` basés sur des dossiers distants (accessible à travers plusieurs nœuds) gérés par un serveur NFS (Network File System).

À noter que cet exercice ne se veut pas être exhaustif pour présenter les différents types de `Volumes` basés sur des dossiers distants (CIFS, `PersistentVolumeClaim`, etc.). Une liste exhaustive est disponible sur la documentation de Kubernetes : https://kubernetes.io/docs/concepts/storage/volumes.

Quelque soit le type d'installation choisi pour la mise en place de votre cluster Kubernetes, toutes les commandes ci-dessous devraient normalement fonctionner. Nous considérons qu'il existe un fichier `k3s.yaml` à la racine du dossier `microservices-kubernetes-gettingstarted-tutorial/`, si ce n'est pas le cas, merci de reprendre la mise en place d'un cluster Kubernetes. Il est important ensuite de s'assurer que la variable `KUBECONFIG` soit initialisée avec le chemin du fichier d'accès au cluster Kubernetes (`export KUBECONFIG=$PWD/k3s.yaml`).

But

- Créer un `Volume` local de type `hostPath`
- Créer un `Volume` local de type `emptyDir`
- Créer un `Volume` distant de type `NFS`

Étapes à suivre

- Avant de commencer les étapes de cet exercice, assurez-vous que le `Namespace` créé dans l'exercice précédent `mynamespaceexercice4` soit supprimé.

```
$ kubectl delete namespace mynamespaceexercice4
namespace "mynamespaceexercice4" deleted
```

- Créer dans le répertoire *exercice5-volumes/* un fichier appelé *mynamespaceexercice5.yaml* en ajoutant le contenu suivant :

```
apiVersion: v1
kind: Namespace
metadata:
  name: mynamespaceexercice5
```

- Créer ce `Namespace` dans notre cluster :

```
$ kubectl apply -f exercice5-volumes/mynamespaceexercice5.yaml
namespace/mynamespaceexercice5 created
```

Nous commençons par le `Volume` de type `hostPath` qui permet de monter une ressource (dossier ou un fichier) depuis le système de fichiers du nœud hôte du `Pod`. Les cas d'usage courants sont l'accès aux éléments internes du nœud (*/var/lib/dock* ou */sys*) dans le cas où vous souhaitez faire du Docker dans Docker ou l'accès à un répertoire distant monté sur les systèmes hôtes de tous les nœuds. Toutefois, certaines précautions doivent être prises pour l'utilisation de ce type de `Volume`. La première est la difficulté d'utiliser un `Volume` de type `hostPath` sur un environnement multi-nœuds et la seconde est qu'il n'est pas possible de choisir le nœud d'un `Pod`.

- Créer dans le répertoire *exercice5-volumes/* un fichier appelé *myhostpath.yaml* qui décrit un `Deployment`, un `Service NodePort` et un `Volume` de type `hostPath` :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mydeploymentwithhostpath
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mypodforhostpath
  template:
    metadata:
      labels:
        app: mypodforhostpath
    spec:
      containers:
      - name: mynginx
        image: nginx:latest
        ports:
        - containerPort: 80
        volumeMounts:
          - mountPath: /usr/share/nginx/html
            name: myhostpathvolume
      volumes:
        - name: myhostpathvolume
          hostPath:
            path: /myhostpath
            type: DirectoryOrCreate
```

```
---

kind: Service
apiVersion: v1
metadata:
  name: mypodforhostpathservice
spec:
  selector:
    app: mypodforhostpath
  type: NodePort
  ports:
    - protocol: TCP
      targetPort: 80
      port: 8080
      nodePort: 30001
```

Le `Volume` est déclaré dans la partie `volumes` où il est identifié par un nom `myhostpathvolume`. Le type de `Volume` est ensuite précisé puis suivent des paramètres spécifiques à `hostPath`. Le paramètre `path` détaille le répertoire sur le système de fichiers où seront stockés les ressources à partager. Le paramètre `type` définit une stratégie de création du répertoire qui pour `DirectoryOrCreate` forcera la création du répertoire *myhostpath* si celui-ci n'existe pas. La partie `volumeMounts` configure le `Volume` du côté du conteneur. Le paramètre `mountPath` détaille le chemin où ce `Volume` sera accessible depuis le `Pod`. Dans cet exemple, le répertoire (*/usr/share/nginx/html*) qui contient la page web par défaut de Nginx est monté avec le dossier */myhostpath* qui se trouve sur le nœud hôte du `Pod`. Vous remarquerez que trois `Pods` ont été créés (`replicas: 3`), mais cela ne veut pas forcément dire que trois nœuds seront utilisés.

- Appliquer la configuration précédente pour créer le `Deployment` et le `Service` dans le cluster Kubernetes :

```
$ kubectl apply -f exercice5-volumes/myhostpath.yaml -n mynamespaceexercice5
deployment.apps/mydeploymentwithhostpath created
service/mypodforhostpathservice created
```

- Examiner sur quel nœud les `Pods` sont déployés :

```
$ kubectl get Pods -n mynamespaceexercice5 -o wide
NAME                                    READY   STATUS    RESTARTS   AGE   IP           NODE
mydeploymentwithhostpath-5744565654-tdq5x   1/1     Running   0          7s    10.42.1.11   k3d-
mycluster-agent-0
mydeploymentwithhostpath-5744565654-kbp6r   1/1     Running   0          7s    10.42.0.12   k3d-
mycluster-server-0
mydeploymentwithhostpath-5744565654-ptrdr   1/1     Running   0          7s    10.42.2.18   k3d-
mycluster-agent-1
```

Kubernetes utilise les trois nœuds pour déployer les trois `Pods`. Un répertoire */myhostpath* devrait exister sur les trois nœuds, puisque la stratégie de création du répertoire est `DirectoryOrCreate`.

- Vérifier le contenu du système de fichiers à la racine / de chaque nœud :

Via K3d

```
$ docker exec -it k3d-mycluster-server-0 ls /
bin  dev  etc  k3d  lib  myhostpath  output  proc  run       sbin  sys  tmp
       usr  var

$ docker exec -it k3d-mycluster-agent-0 ls /
bin  dev  etc  k3d  lib  myhostpath  proc  run       sbin  sys  tmp usr
var

$ docker exec -it k3d-mycluster-agent-1 ls /
bin  dev  etc  k3d  lib  myhostpath  proc  run       sbin  sys  tmp usr
var

$ docker exec -it k3d-mycluster-agent-1 ls /myhostpath
```

Un dossier */myhostpath* existe sur chacun des trois nœuds, mais son contenu est vide (vérifié sur le second nœud de travail).

- Exécuter une requête via l'utilisation du `Service NodePort` pour vérifier qu'il n'existe pas encore de contenu :

Via K3d

```
$ curl localhost:30001
<html>
<head><title>403 Forbidden</title></head>
<body>
<center><h1>403 Forbidden</h1></center>
<hr><center>nginx/1.23.3</center>
</body>
</html>
```

Ne pas oublier que le `Service NodePort` va distribuer aléatoirement la requête sur tous les nœuds du cluster où le `Pod` est déployé.

- Ajouter un fichier `index.html` dans le dossier *myhostpath* du nœud maître :

Via K3d

```
$ docker exec k3d-mycluster-server-0 sh -c "echo 'Bonjour depuis le noeud
Master' > /myhostpath/index.html"
```

À cet instant sur les trois nœuds disponibles de notre cluster Kubernetes, seul le nœud maître possède un contenu.

- Faire autant de requêtes sur le cluster Kubernetes pour obtenir le contenu suivant `Bonjour depuis le noeud Master` :

Via K3d

```
$ curl localhost:30001
<html>
<head><title>403 Forbidden</title></head>
<body>
<center><h1>403 Forbidden</h1></center>
<hr><center>nginx/1.23.3</center>
</body>
</html>
$ curl localhost:30001
Bonjour depuis le noeud Master
```

Le résultat attendu `Bonjour depuis le noeud Master` est obtenu ici en deux requêtes. Une manière détournée de `hostPath` pour partager une ressoource commune (dossier ou fichier) consisterait à monter sur chaque nœud un dossier distant comme par exemple NFS ou CIFS. Toutefois, nous vous recommandons de consulter la liste des volumes existants avant d'entreprendre des manipulations compliquées qui pourraient être réalisées simplement.

- Pour continuer l'exercice, supprimer les précédents objets `Deployment` et `Service` :

```
$ kubectl delete service -n mynamespaceexercice5 mypodforhostpathservice
service "mypodforhostpathservice" deleted
$ kubectl delete deployments.apps -n mynamespaceexercice5 mydeploymentwithhostpath
deployment.apps "mydeploymentwithhostpath" deleted
```

Le deuxième type de `Volume` étudié est `emptyDir`. Ce `Volume` permet le partage des données entre les conteneurs d'un même `Pod`. À la différence de `hostPath`, le `Volume emptyDir` est non peristant. Son contenu est vide à sa création. Les données peuvent être stockées sur disque ou en mémoire, mais ne seront pas conservées à la destruction du `Volume`. Les cas d'usage courants sont le partage d'un espace de travail commun entre différents conteneurs d'un même `Pod` et la mise à disposition d'un point de reprise après un arrêt non souhaité (crash) d'un traitement long. C'est le premier cas d'usage relatif au partage d'un espace de travail que nous allons présenter dans la suite.

La suite de l'exercice va consister à utiliser un `Volume emptyDir` pour y placer le contenu d'un dépôt Git. Un premier conteneur récupérera le contenu d'un dépôt Git contenant un site web statique et le second conteneur fournira un serveur web Nginx pour mettre en ligne le site web.

- Créer dans le répertoire *exercice5-volumes/* un fichier appelé *myemptydir.yaml* qui décrit un `Deployment`, un `Service NodePort` et un `Volume` de type `emptyDir` :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mydeploymentwithemptydir
spec:
  replicas: 2
  selector:
    matchLabels:
      app: mypodforemptydir
  template:
    metadata:
      labels:
        app: mypodforemptydir
    spec:
      containers:
      - name: mynginx
        image: nginx:latest
        ports:
        - containerPort: 80
        volumeMounts:
          - mountPath: /usr/share/nginx/html
            name: myemptydirvolume
      initContainers:
      - name: mygit
        image: alpine/git
        args:
          - clone
          - --
          - https://github.com/cloudacademy/static-website-example
          - /data
        volumeMounts:
          - mountPath: /data
            name: myemptydirvolume
      volumes:
        - name: myemptydirvolume
          emptyDir:
            medium: Memory

---

kind: Service
apiVersion: v1
metadata:
  name: mypodforemptydirservice
spec:
  selector:
    app: mypodforemptydir
  type: NodePort
  ports:
    - protocol: TCP
      targetPort: 80
      port: 8080
      nodePort: 30001
```

Ce fichier de configuration contient deux conteneurs appelés `mynginx` et `mygit`. Le premier est déclaré via le paramètre `containers` tandis que le second utilise le paramètre `initContainers`. Les conteneurs déclarés dans `initContainers` sont appelés des conteneurs d'initialisation. Ils sont démarrés les uns après les autres (l'ordre à une importance) et chaque conteneur d'initialisation doit se terminer avec succès avant que le prochain conteneur d'initialisation ne démarre. Quand tous les conteneurs d'initialisation sont terminés (sans erreur), les conteneurs déclarés dans le paramètre `containers` peuvent démarrer. Dans l'exemple que nous traitons, un conteneur d'initialisation s'occupera d'aller chercher dans un dépôt Git des fichiers d'une page web statique. Ces fichiers seront placés dans un `Volume emptyDir` qui est partagé avec le conteneur `mynginx` (même `Volume` identifié par `myemptydirvolume`). Veuillez noter que nous avons configuré le `Volume emptyDir` pour que le stockage se fasse en mémoire.

- Appliquer la configuration précédente pour créer le `Deployment` et le `Service` dans le cluster Kubernetes :

```
$ kubectl apply -f exercice5-volumes/myemptydir.yaml -n mynamespaceexercice5
deployment.apps/mydeploymentwithemptydir created
service/mypodforemptydirservice created
```

Il ne reste plus qu'à tester ce `Deployment` en utilisant votre navigateur préféré pour afficher le contenu.

Via K3d

- Ouvrir un navigateur et saisir l'adresse http://localhost:30001

Cette solution à base de `Volume emptyDir` nécessite de télécharger le dépôt Git autant de fois qu'il y a de `Pod` à créer. Autant dire que cette façon de faire est un cas d'école et qu'il ne s'agit pas d'une bonne pratique pour déployer un microservice stateless qui doit retourner une page web statique. Pour améliorer ce point, un dossier distant est monté sur tous les `Pods` créés. Le contenu du dossier distant sera partagé et nous n'aurons besoin que de télécharger le dépôt Git qu'une seule fois. Nous allons donc nous intéresser dans la fin de cet exercice à mettre en place un dossier distant NFS (Network File System) qui sera accessible depuis Kubernetes via un `Volume` de type `NFS`.

- Pour continuer l'exercice, supprimer les précédents objets `Deployment` et `Service` :

```
$ kubectl delete service -n mynamespaceexercice5 mypodforemptydirservice
service "mypodforemptydirservice" deleted
$ kubectl delete deployments.apps -n mynamespaceexercice5
mydeploymentwithemptydir
deployment.apps "mydeploymentwithemptydir" deleted
```

Avant de montrer la configuration pour manipuler un `Volume` de type `NFS`, nous allons devoir configurer un serveur NFS pour les besoins de notre exercice.

La mise en place d'un `Volume` de type `NFS` avec la distribution K3d n'est pas complète. Je n'ai pas encore trouvé de solution élégante pour tout intégrer dans des conteneurs Docker. Il subsiste également un problème quand on souhaite configurer un Volume NFS à K3d, la connexion est refusée quoi qu'il arrive.

Via K3d

TBA

- Créer dans le répertoire *exercice5-volumes/* un fichier appelé *mynfs.yaml* qui décrit un `Deployment`, un `Service NodePort` et un `Volume` de type `NFS` :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mydeploymentwithnfs
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mypodfornfs
  template:
    metadata:
      labels:
        app: mypodfornfs
    spec:
      containers:
      - name: mynginx
        image: nginx:latest
        ports:
        - containerPort: 80
        volumeMounts:
          - mountPath: /usr/share/nginx/html
            name: mynfsvolume
      volumes:
      - name: mynfsvolume
        nfs:
          server: <IP_NFS-SERVER>
          path: "/nfs"

---

kind: Service
apiVersion: v1
metadata:
  name: mypodfornfsservice
spec:
  selector:
    app: mypodfornfs
  type: NodePort
  ports:
    - protocol: TCP
      targetPort: 80
      port: 8080
      nodePort: 30001
```

- Remplacer dans ce fichier de configuration `<IP_NFS-SERVER>` par l'adresse IP de la machine virtuelle (`192.168.64.13`).
- Appliquer la configuration précédente pour créer le `Deployment` et le `Service` dans le cluster Kubernetes :

```
$ kubectl apply -f exercice5-volumes/mynfs.yaml -n mynamespaceexercice5
deployment.apps/mydeploymentwithnfs created
service/mypodfornfsservice created
```

Il ne reste plus qu'à tester ce `Deployment` en utilisant votre navigateur préféré pour afficher le contenu.

Via K3d

TBA

Bilan de l'exercice

À cette étape, vous savez :

- créer un `Volume` local de type `hostPath` ;
- créer un `Volume` local de type `emptyDir` ;
- créer un conteneur d'initialisation ;
- créer un serveur NFS ;
- créer un `Volume` distant de type `NFS`.

Pour continuer sur les concepts présentés dans cet exercice, nous proposons de continuer avec les manipulations suivantes :

- créer un objet `PersistentVolume` dont le `StorageClass` est de type `NFS` ;
- créer un objet `PersistentVolumeClaim` ;
- créer un `Deployment` basé sur une image Docker Apache HTTP avec une configuration via le précédent `PersistentVolumeClaim` ;
- créer un `Service` de type `NodePort` pour ce `Deployment`.