FONCTIONNEMENT ET MANIPULATION DES VOLUMES

Introduction

Permettez-moi de vous faire une petite piqûre de rappel, pour ceux qui ne le savent pas encore le concept de volume est déjà présent dans Docker (à ce sujet voici mon article sur les volumes Docker).

En effet, les données dans un conteneur sont éphémères, c'est-à-dire que lorsqu'un conteneur est supprimé tous ses fichiers le sont aussi. Cela peut donc engendrer de réels problèmes pour vos applications conteneurisées. Il a donc fallu vite trouver une solution pour sauvegarder ces données, d'où la création des volumes, qui restent la solution privilégiée quant à la persistance des données générées par les conteneurs.

C'est le même principe dans Kubernetes, lorsqu'un conteneur tombe en panne, kubelet le redémarre, et ses fichiers sont perdus. Dès lors lorsque vous exécutez un ensemble de conteneurs au sein d'un Pod, il est souvent nécessaire de **sauvegarder ou de partager des fichiers** entre ces conteneurs, et là encore les volumes résolvent ces deux problèmes. Nous allons donc dans ce chapitre **étudier et comprendre le fonctionnement et la gestion des volumes dans Kubernetes**.

Les types de volumes

Kubernetes prend en charge plusieurs types de volumes tel que le support de stockage NFS ou ceux proposés par des fournisseurs Cloud comme EBS de chez

AWS (vous retrouvez la liste des volumes acceptés par Kubernetes). Nous allons étudier quelques types de volumes dans ce chapitre.

hostPath

Dans cet exemple nous utiliserons le type de volume `hostPath` afin de vous démontrer la **simplicité de création d'un volume Kubernetes** mais également les inconvénients de ce type de volume.

Ce type de volume monte un fichier ou un répertoire du système de fichiers du nœud abritant votre pod (pour information nous avons déjà eu l'occasion de l'utiliser dans cet article).

Je vais d'ailleurs reprendre le même exemple de l'article. Nous disposerons alors de deux Pods (alpine + nginx) qui partageront un fichier grâce au système de volume que nous mettrons en place. Plus précisément le Pod alpine s'occupera de rajouter la date courante dans un fichier **index.html** dans le volume créé. Ce même fichier sera ensuite utilisé en tant que page web d'accueil par le Pod nginx.

Voici à quoi ressemblera notre template YAML :

```
apiVersion: v1
kind: Pod
metadata:
  name: multic
spec:
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
  - name: alpine
    image: alpine
    volumeMounts:
    - name: html
      mountPath: /html
    command: ["/bin/sh", "-c"]
    args:
```

```
      - date >> /html/index.html;
        while true; do
          sleep 1;
        done

  volumes:
  - name: html
    hostPath:
      path: /data
      type: DirectoryOrCreate
```

Dans ce template nous avons deux champs qui définissent la configuration de notre nouveau volume :

- `volumeMounts` : il s'agit du chemin dans le conteneur sur lequel le montage aura lieu.
- `Volume` : définit la config du volume que nous allons réclamer, dans notre cas il sera de type `hostPath` et le volume sera monté dans le dossier `/data` du nœud hébergeant les Pods.

```
type: DirectoryOrCreate
```

Dans cette partie, nous définissons le comportement de notre volume de type `hostPath`, les valeurs prises en charge par ce type de volume sont les suivantes :

Valeur	Comportement
	Une chaîne de caractères vide (par défaut) sert à la rétrocompatibilité, ce qui signifie qu'aucune vérification ne sera effectuée avant de monter le volume hostPath.
`DirectoryOrCreate`	Si rien n'existe au chemin fourni, un dossier vide y sera créé au besoin avec les permissions définies à 0755, avec le même groupe et la même possession que Kubelet.
`Directory`	Un dossier doit exister au chemin fourni

Valeur	Comportement
`FileOrCreate`	Si rien n'existe au chemin fourni, un fichier vide y sera créé au besoin avec les permissions définies à 0644, avec le même groupe et la même possession que Kubelet.
`File`	Un fichier doit exister au chemin fourni
`Socket`	Un socket UNIX doit exister au chemin fourni
`CharDevice`	Un périphérique en mode caractère doit exister au chemin fourni
`BlockDevice`	Un périphérique en mode bloc doit exister au chemin fourni

Créons désormais notre Pod avec les volumes convenus :

```
kubectl create -f pod.yaml
```

Vérifions ensuite sur quel nœud est hébergé notre pod :

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

Résultat :

```
NAME    READY   STATUS    RESTARTS   AGE    IP             NODE       NOMINATED NODE
multic   1/1     Running   0          5m2s   192.168.1.45   worker-1   <none>
```

Notre Pod est donc disponible dans le nœud `worker-1` qui dans mon cas a pour adresse IP `192.168.50.11`. Dorénavant, émettons une requête http vers notre pod `multic` :

```
ssh vagrant@192.168.50.11 curl -s  http://$(kubectl get pod multic --template={{.statu
```

Résultat :

```
Mon Aug 31 19:32:11 UTC 2019
```

Supprimons notre pod et recréons-le, de façon à vérifier la conservation des

données des conteneurs de ce pod :

```
kubectl delete pod multic && kubectl create -f pod.yaml
```

Vérifions à nouveau notre page :

```
ssh vagrant@192.168.50.11 curl -s  http://$(kubectl get pod multic --template={{.statu
```

Résultat :

```
Mon Aug 31 19:33:27 UTC 2019
Mon Aug 31 19:32:11 UTC 2019
```

Cependant, ce type de volume et cette façon de création de volumes comportent quelques inconvénients, comme :

- Dans un cluster multi-nœuds, il faut impérativement que le dossier du volume soit disponible et identique dans tous vos nœuds.
- Lorsque Kubernetes accomplit une planification en tenant compte des ressources, il ne prendra pas en compte les ressources utilisées.

Nous verrons plus tard dans ce chapitre comment éviter ces types de problèmes.

secret

`secret` est un type de volume utilisé pour **transmettre des informations sensibles**, telles que des mots de passe. Il permet de mieux contrôler la manière dont l'information est utilisée et réduit le risque d'exposition accidentelle, allons voir comment l'utiliser.

Commençons par la création d'un fichier contenant un mot de passe :

```
echo -n '1f2d1e2e67df' > ./password.txt
```

Ensuite, nous **construisons un objet k8s de type secret** en prenant en compte notre fichier **password.txt** :

```
kubectl create secret generic secret-password --from-file=./password.txt
```

Résultat :

```
secret/secret-password created
```

Par la suite, vérifions la **liste des secrets disponibles dans le cluster** :

```
kubectl get secrets
```

Résultat :

```
NAME                  TYPE                                  DATA   AGE
secret-password       Opaque                                1      76s
```

Les secrets peuvent être **décodés** via la commande `kubectl get secret`. Par exemple, pour récupérer le secret créé dans la section précédente, nous utiliserons la commande suivante :

```
kubectl get secret secret-password -o yaml
```

Résultat :

```
apiVersion: v1
data:
    password.txt: MWYyZDFlMmU2N2Rm
...
```

Utilisons la valeur fournit dans le champ `data` afin de découvrir notre mot de passe :

```
echo 'MWYyZDFlMmU2N2Rm' | base64 --decode
```

Résultat :

```
1f2d1e2e67df
```

Après avoir conçu notre secret, on peut ensuite le monter en tant que volume de données et l'utiliser das notre pod, comme suit :

```
apiVersion: v1
kind: Pod
metadata:
  name: alpine
spec:
  containers:
  - name: alpine
    image: alpine
    volumeMounts:
    - name: secret-password
      mountPath: "/etc/password"
      readOnly: true
    command: ["/bin/sh", "-c"]
    args:
      - while true; do
          sleep 1;
        done
  volumes:
  - name: secret-password
    secret:
      secretName: secret-password
```

Dans la suite, créons notre pod avec notre volume de type secret :

```
kubectl create -f pod-secret.yaml
```

Récupérons ensuite la valeur du mot de passe depuis notre conteneur `alpine` :

```
kubectl exec alpine -c alpine -- cat /etc/password/password.txt
```

Résultat :

```
1f2d1e2e67df
```

Volume persistant et revendication de volume persistant

Quand vous travaillez dans un grand environnement avec beaucoup d'utilisateurs qui déploient beaucoup de pods, les utilisateurs devront configurer à chaque fois le volume de stockage pour chaque pod. De ce fait chaque fois que des changements doivent être faits, l'utilisateur doit alors les appliquer pour tous ses pods.

Ici la solution serait de gérer le stockage de manière plus centralisée, pour ceci Kubernetes nous propose l'objet `PersistentVolume` (PV).

Un PersistentVolume est un élément de stockage dans le cluster qui a été provisionné par un administrateur, leur cycle de vie est indépendant de tout pod.

Ensuite, nous avons aussi à notre disposition le `PersistentVolumeClaim` (PVC) qui permet aux utilisateurs de revendiquer l'utilisation des ressources d'un des PersistentVolume disponible dans le cluster k8s, dans le but de consommer une partie de ses ressources.

Dans l'intention de faciliter la compréhension de ces deux nouvelles notions, nous allons imaginer le scénario suivant :

"Vous êtes l'administrateur d'un grand cluster Kubernetes, et on vous demande ainsi de mettre en place un système de persistance de données."

Votre première mission consiste donc à créer d'abord un certain nombre de PV qui contiendront les détails de stockage disponible pour les utilisateurs de votre cluster. Sans plus attendre vous commencez par **créer votre PersistentVolume** :

```

```

```
kind: PersistentVolume
apiVersion: v1
metadata:
   name: my-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /pv-data
    type: DirectoryOrCreate
```

Attention

Dans un cluster de production, en tant qu'administrateur vous n'utiliseriez pas le type `hostPath`. Au lieu de cela, mettez en place un service de partage réseau tel qu'un partage NFS, Google Compute Engine Persistent Disk, Amazon Elastic Block Store, etc ...

Nous avons donc spécifié un volume persistant de type `hostPath` nommé `my-pv` avec une capacité de stockage de 1 Gio ainsi que des droits d'accès en lecture et écriture (`ReadWriteOnce`).

Information

Actuellement, la taille de stockage est la seule ressource pouvant être définie ou demandée. Dans des futures versions Kubernetes compte inclure d'autres attributs comme le IOPS, débit, etc.

Voici la commande pour **créer votre PersistentVolume** :

```
kubectl create -f pv.yaml
```

Vous pouvez **vérifier la liste de vos PersistentVolume**, de la façon suivante :

```
kubectl get pv
```

Résultat :

```
NAME    CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM   STORAGECLASS
my-pv   1Gi        RWO            Retain           Available
```

Vous avez donc réalisé votre mission en créant un PersistentVolume. Heureux de votre exploit, vous indiquez à vos utilisateurs la consigne suivante *"Mesdames et messieurs, vous pouvez dès maintenant, réclamer jusqu'à 1 Gio de stockage"* .

Comme vous êtes gentil, vous montrez à un de vos utilisateurs **comment créer un PersistentVolumeClaim** avec le template YAML suivant :

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
   name: my-pvc
spec:
   accessModes:
      - ReadWriteOnce
   resources:
      requests:
        storage: 500Mi
```

Ici, l'utilisateur revendique une utilisation de 500 Mio d'espace de stockage en mode lecture et écriture, une fois la demande lancée, le cluster va alors analyser automatiquement la demande de l'utilisateur et lui attribuer un PersistentVolume.

Créez votre PersistentVolumeClaim avec la commande suivante :

```
kubectl create -f pvc.yaml
```

Même chose, vous pouvez **vérifiez la liste de vos PersistentVolumeClaim avec la commande suivante** :

```
kubectl get pvc
```

Résultat :

```
NAME     STATUS   VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
my-pvc   Bound    my-pv    1Gi        RWO                           94s
```

Maintenant que tout est en place, votre utilisateur peut utiliser son PersistentVolumeClaim dans son pod, de la manière suivante :

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - name: nginx-pvc
      mountPath: "/usr/share/nginx/html"
  volumes:
  - name: nginx-pvc
    persistentVolumeClaim:
      claimName: my-pvc
```

Enfin, votre utilisateur crée ensuite son Pod :

```
kubectl create -f pod-pvc.yaml
```

Désormais, testons notre volume. Comme nous avons utilisé un volume de type `hostPath`, nous allons au préalable récupérer le nœud sur lequel est hébergé notre pod :

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

Résultat :

```
NAME    READY   STATUS    RESTARTS   AGE     IP              NODE       NOMINATED NODE
nginx   1/1     Running   0          6m39s   192.168.1.190   worker-1   <none>
```

Après cela, vérifions si le dossier de notre volume a bien été crée dans notre nœud :

```
ssh vagrant@192.168.50.11 ls -l  / | grep pv-data
```

C'est bien le cas :

```
drwxr-xr-x   2 root root  4096 Sep  3 17:04 pv-data
```

Maintenant, nous allons rajouter notre propre page d'accueil. Pour ce faire nous nous connecterons directement sur notre nœud via le protocole ssh :

```
ssh vagrant@192.168.50.11
sudo su - root
echo "ceci est un test" > /pv-data/index.html
```

Après avoir rajouté notre propre page d'accueil l'étape suivante est de lancer une requête sur notre conteneur nginx :

```
ssh vagrant@192.168.50.11 curl -s  http://$(kubectl get pod nginx --template={{.status
```

Résultat :

```
ceci est un test
```

Conclusion

Il est temps de conclure ce chapitre, nous avons pu voir comment utiliser certains types de volumes supportés par kubernetes et certains de leurs inconvénients, d'ailleurs n'hésitez pas à vous entraîner sur d'autres types de volumes non vus dans ce chapitre. Par la suite, nous nous sommes mis dans la peau d'un administrateur

de cluster Kubernetes dans le but de créer un système de persistance de données, pour cela nous avons étudié et éxploité l'objet PersistentVolume et les PersistentVolumeClaim.