GÉRER ET MANIPULER LES PODS KUBERNETES

Introduction

Dans ce chapitre, nous allons commencer à réellement manipuler notre cluster k8s, de façon précise, nous nous préoccuperons de **la plus petite unité dans Kubernetes**, à savoir les Pods.

Qu'est-ce qu'un pod?

Nous avons déjà défini les pods dans cet article, mais permettez-moi tout de même de vous faire une petite piqûre de rappel. Promis elle ne vous fera pas mal .

Un Pod est tout simplement un groupe d'un ou plusieurs conteneurs, avec un stockage et un réseau partagé. Par conséquence, ces conteneurs communiquent plus efficacement entre eux et assurent une localisation de la donnée.

De ce fait, il existe deux types de Pods :

- **Pod à conteneur simple**.
- **Pod à conteneur multiple**.

Nous allons donc étudier ces deux types de Pods.

Création du Pod

Si vous avez suivi les chapitres précédents, nous avons en notre disposition un cluster Kubernetes avec un nœud master nommé `master` avec l'ip `192.168.50.10` et un nœud worker `worker-1` et `192.168.50.11` en adresse IP. Nous avons aussi en notre présence l'outil de ligne de commande `kubectl` qui est configuré pour communiquer avec notre cluster K8S.

> **Information**
>
> Si vous n'avez pas ces prérequis, merci de regarder mes articles précédents de Kubernetes qui vous expliquent en détail comment les mettre en place.

Nous utiliserons un template au format YAML afin de créer la configuration de notre pod, puis on exécutera la commande `kubectl create` qui permet de **créer un objet Kubernetes**, dans notre cas ça sera un objet de type Pod.

Primo, il faut savoir que pour chaque objet Kubernetes que vous souhaiterez créer, vous devez définir des valeurs pour les champs suivants :

- `apiVersion` : la version de l'API Kubernetes que vous utilisez pour créer cet objet .
- `kind` : le type d'objet k8s que vous comptez créer.
- `metadata` : données de type clé valeur permettant d'identifier de manière unique l'objet.
- `spec` : contient la spécification avec des champs imbriqués propres à chaque objet k8S. Le format est donc différent pour chaque objet Kubernetes.

Pod à conteneur unique

Dans l'exemple ci-dessous nous créerons un Pod contenant un conteneur issu de l'image nginx.

Commençons donc par créer un fichier yaml nommé **nginx-pod.yaml** et remplissons dedans les champs requis pour la création de notre Pod :

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    type: web
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
```

Quelques explications s'imposent, de façon à comprendre l'étendue du fichier :

```
apiVersion: v1
kind: Pod
```

Ici, nous utilisons l'api en version 1 et nous créons un objet Kubernetes de type Pod.

```
metadata:
  labels:
    type: web
  name: nginx
```

Dans cette partie, nous désignons un label avec une clé nommée `type` et sa valeur `web` (vous pouvez rajouter n'importe quelle clé et valeur), cette donnée n'est pas obligatoire mais sachez juste qu'elle peut nous servir plus tard dans d'autres objets Kubernetes afin de sélectionner et manipuler comme bon nous semble tous les pods contenant ce label.

Par la suite on ne fait que nommer notre Pod `nginx`.

```
spec:
  containers:
  - image: nginx
    name: nginx
```

Dans cette phase, nous indiquons la configuration de notre Pod, plus précisément nous spécifions la configuration du conteneur du Pod en question. Très exactement nous désignons un conteneur utilisant l'image nginx et qu'on nomme nginx (très original).

Maintenant, nous allons **créer notre pod Kubernetes** avec la commande suivante :

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

Résultat :

```
pod/nginx created
```

Vérifions maintenant **la liste des pods disponibles dans notre cluster Kubernetes** :

```
kubectl get pods
```

Résultat :

```
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          2m36s
```

Bon ok, nous voyons bien qu'il existe seulement un seul pod disponible dans notre cluster k8s. Cela dit, il serait plus intéressant d'en plus, **récupérer l'IP du pod mais aussi le nœud qui héberge ce pod**. Pour cela, nous rajouterons l'option `-o wide` :

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

Résultat :

```
NAME    READY   STATUS    RESTARTS   AGE    IP             NODE       NOMINATED NODE
nginx   1/1     Running   0          5m1s   192.168.1.13   worker-1   <none>
```

D'après le résultat, notre Pod `nginx` est situé dans le nœud `worker-1` et possède l'ip `192.168.1.13`.

Il faut savoir que l'ip du Pod n'est accessible qu'à partir du notre nœud sur lequel il est hébergé (dans notre cas c'est le nœud `worker-1`). Nous verrons dans la partie dédiée aux `Services`, à comment rendre notre pod accessible depuis l'extérieur de notre cluster.

Essayons alors d'accéder à la page web de notre pod `nginx` depuis notre nœud `worker-1` qui pour rappel, possède l'adresse IP `192.168.50.11` :

```
ssh vagrant@192.168.50.11 curl -s http://192.168.1.13
```

Résultat :

```
<!DOCTYPE html>
<html>
...
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Passons dès à présent, à la partie mutli-conteneurs.

Pod à conteneur multiple

Dans cette partie, nous allons nous aventurer un peu plus en profondeur en créant deux conteneurs partageant le même volume afin de les faire communiquer ensemble au sein d'un même pod.

Nous garderons le même squelette que le fichier yaml précédent, mais nous ferons quelques rajouts. Ce qui nous donnera :

```
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
```

Place maintenant à l'explication de ces nouveaux rajouts. Nous avons alors :

```
volumeMounts:
- name: html
  mountPath: /usr/share/nginx/html

volumeMounts:
- name: html
  mountPath: /html
```

Comme sur Docker les données dans un conteneur sont éphémères, c'est-à-dire quand un conteneur est supprimé ses données sont détruites avec. Il faut donc trouver un moyen pour les rendre **persistantes**, de ce fait nous utiliserons les volumes.

Dans cet exemple, un volume nommé `html` est créé dont nous spécifions sa configuration plus tard dans le fichier yaml. Ce volume sera monté sur le dossier `/usr/share/nginx/html` du conteneur `nginx` et sur le dossier `/html` du conteneur `apline`.

```
command: ["/bin/sh", "-c"]
args:
  - date >> /html/index.html;
    while true; do
      sleep 1;
    done
```

Ici, nous spécifions la commande qui sera lancée au moment de la création du conteneur `alpine` , cette commande va tout simplement rajouter la date courante dans le fichier `index.html` , ce même fichier est utilisé aussi en tant que page web d'accueil par le conteneur `nginx`. Ce partage est possible grâce au volume `html`.

Par la suite nous rajoutons une boucle infinie afin que le conteneur soit en permanence active, puisqu'il faut toujours un processus qui tourne en premier plan pour que le conteneur soit toujours à l'état running.

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

Enfin, nous définissons la configuration de notre volume `html`.

Déjà, notre volume est de type `hostPath`, ce type de volume monte un fichier ou un répertoire du système de fichiers du nœud hôte dans lequel le pod s'exécutera.

Nous précisons aussi le type de `hostPath`, ici il est de type `DirectoryOrCreate`, qui signale que si le chemin indiqué dans la valeur du champ `path` n'existe pas, alors un répertoire vide y sera créé , avec le droit en 0755, ayant le même groupe et le même propriétaire que l'outil Kubelet.

Désormais, créons notre pod :

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

Résultat :

```
pod/multic created
```

Visitons maintenant notre serveur web avec la commande suivante :

```
ssh vagrant@192.168.50.11 curl -s  http://$(kubectl get pod multic --template=&lcub;&
```

Résultat :

```
Mon Aug 12 18:32:11 UTC 2019
```

> **Information**
>
> Nous avons utilisé `$(kubectl get pod multic --template=&lcub;&lcub;.status.podIP}})` afin de récupérer automatiquement l'ip de notre Pod.

Nous allons **supprimer notre pod et le recréer avec la commande suivante**, de façon à vérifier que nos données sont bien conservées :

```
kubectl delete pod multic
```

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

Vérifions de nouveau notre page :

```
ssh vagrant@192.168.50.11 curl -s  http://$(kubectl get pod multic --template=&lcub;&
```

Résultat :

```
Mon Aug 12 18:32:11 UTC 2019
Mon Aug 12 19:01:24 UTC 2019
```

On peut remarquer d'après le résultat, que les données du fichier **index.html** sont bien sauvegardées dans le volume `html`.

Avertissement

Faites attention lorsque vous utilisez ce type de volume, car :

- Dans un cluster multi-nœuds, vos pods peuvent se comporter différemment sur différents nœuds en raison de données différentes sur les différents nœuds .
- lorsque Kubernetes accomplit une planification en tenant compte des ressources, il ne prendra pas en compte les ressources utilisées par un volume de type `hostPath` .

Nous verrons dans un futur chapitre une meilleure méthode pour créer nos volumes.

Gardez à l'esprit, bien que vous puissiez héberger une application mutli-conteneurs dans un même pod, la méthode recommandée consiste à utiliser des pods distincts pour chaque application tier. La raison en est simple, vous pouvez scaler vos conteneurs de manière indépendante et les répartir sur les nœuds du cluster.

Autres commandes

Débogage

Voici d'autres commandes utiles, qui peuvent notamment vous aider à **debuger vos pods**.

Nous avons premièrement, **la commande qui permet de vérifier les logs des conteneurs de vos pods** :

```
kubectl logs multic -c nginx
```

Résultat :

```
192.168.1.1 - - [12/Aug/2019:19:02:19 +0000] "GET / HTTP/1.1" 200 87 "-" "curl/7.58.0
```

Cette commande vient avec quatres options très utiles :

- `-f` : suivre en permanence les logs du conteneur (correspond à un tail -f sur Linux).
- `--tail` : nombre de lignes les plus récentes à afficher.
- `--since=1h` : afficher tous les logs du conteneur au cours de la dernière heure .
- `--timestamps` : afficher la date et l'heure de réception des logs d'un conteneur.

Ensuite vous avez la **commande qui permet de décrire et de récupérer les informations détaillées de votre pod** :

```
kubectl describe <KUBERNETES OBJECT> <OBJECT NAME>
```

Soit :

```
kubectl describe pod multic
```

Une des informations intéressantes que vous pouvez **récupérer les différents événements qu'a subis votre objet Kubernetes**.

Ces événements sont créés automatiquement lorsque votre objet Kubernetes s'expose à des changements d'état. Voici par exemple les événements de notre pod `multic` :

```
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  169m  default-scheduler  Successfully assigned default/multic to
  Normal  Pulling    169m  kubelet, worker-1  Pulling image "nginx"
  Normal  Pulled     169m  kubelet, worker-1  Successfully pulled image "nginx"
  Normal  Created    169m  kubelet, worker-1  Created container nginx
  Normal  Started    169m  kubelet, worker-1  Started container nginx
  Normal  Pulling    169m  kubelet, worker-1  Pulling image "alpine"
  Normal  Pulled     169m  kubelet, worker-1  Successfully pulled image "alpine"
  Normal  Created    169m  kubelet, worker-1  Created container alpine
  Normal  Started    169m  kubelet, worker-1  Started container alpine
```

On peut ainsi connaître toutes les étapes supervisées par notre nœud master, qui ont mené à la création de notre pod. Par exemple on sait qu'il a d'abord décidé sur quel nœud le pod s'exécutera et par la suite il a respectivement téléchargé et instancié les images voulues.

Si un événement venait à interrompre le déroulement du pod, comme par exemple la spécification dans le fichier yaml d'une image inexistante dans le Docker Hub, vous percevriez alors l'information directement dans les événements du pod.

Vous pouvez aussi **exécuter une commande directement dans le conteneur de votre Pod**, grâce à la commande suivante :

```
kubectl exec multic -c alpine cat  /html/index.html
```

Résultat :

```
Mon Aug 12 18:30:46 UTC 2019
Mon Aug 12 18:32:11 UTC 2019
Mon Aug 12 19:01:24 UTC 2019
Tue Aug 13 14:44:22 UTC 2019
```

Comme sur Docker vous pouvez utiliser les options `-t` et `-i` afin de vous **connecter directement sur le shell du conteneur de votre Pod** :

```
kubectl exec -ti multic -c alpine /bin/sh
```

Résultat :

```
/ #
```

Appliquer des changements sans détruire le Pod

Dans cet exemple, j'ai volontairement rajouté une image invalide dans mon template YAML, ce qui nous donne l'état du pod suivant :

```
kubectl get pods
```

Résultat :

```
NAME     READY   STATUS             RESTARTS   AGE
multic   1/2     InvalidImageName   0          28m
```

Sans la nécessité de supprimer et recréer votre Pod, vous pouvez **appliquer des changements d'un Pod depuis votre template YAML** avec la commande ci-dessous :

```
kubectl apply -f nginx-pod.yaml
```

Résultat :

```
pod/multic configured
```

Maintenant si je revérifie l'état de mon pod, j'obtiens :

```
kubectl get pods
```

Résultat :

```
NAME     READY   STATUS    RESTARTS   AGE
multic   2/2     Running   1          34m
```

Conclusion

Dans ce chapitre, nous avons vu comment créer, gérer, supprimer, déboguer nos Pods. Mais c'est encore insuffisant, car nous devons étudier dans les chapitres suivants d'autres objets Kubernetes dans l'intention d'exploiter au maximum nos Pods.

Je partage avec vous, un **aide-mémoire des commandes de manipulation des Pods** :

```
# Afficher la liste des pods
kubectl get pods [En option <POD NAME>]
    -o : format de sortie
        wide : afficher l'ip du pod et le nœud qui l'héberge
        yaml : afficher encore plus d'informations sur un pod sous format YAML
        json : afficher encore plus d'informations sur un pod sous format JSON
    --template : récupérer des informations précises de la sortie de la commande
        Récupérer l'IP d'un pod : kubectl get pod  <POD NAME> --template=&lcub;&lcub;

# Créer un Pod
kubectl create -f <filename.yaml>

# Supprimer un pod
kubectl delete pods <POD NAME>

# Appliquer des nouveaux changements à votre Pod sans le détruire
```

```
kubectl apply -f <filename.yaml>

# Afficher les détails d'un pod
kubectl describe pods <POD NAME>

# Exécuter une commande d'un conteneur de votre pod
kubectl exec <POD NAME> -c <CONTAINER NAME> <COMMAND>
    -t : Allouer un pseudo TTY
    -i : Garder un STDIN ouvert

# Afficher les logs d'un conteneur dans un pod
kubectl logs <POD NAME> -c <CONTAINER NAME>
    -f : suivre en permanence les logs du conteneur
    --tail : nombre de lignes les plus récentes à afficher
    --since=1h : afficher tous les logs du conteneur au cours de la dernière heure
    --timestamps : afficher la date et l'heure de réception des logs d'un conteneur
```