GÉRER ET MANIPULER UN SERVICE KUBERNETES

Introduction

Dans cet article, nous aborderons les `services` Kubernetes. Nous nous pencherons sur leur création et utilisation.

C'est quoi un service Kubernetes ?

Un service peut être défini comme une abstraction par-dessus les Pods, qui attribue aux pods leurs propres adresses IP et un nom DNS unique, et peut aussi équilibrer la charge entre eux.

En effet, les pods ont une durée de vie très limité et sont à maintes reprises créés et détruits, ce qui par conséquent altère continuellement leurs adresses IP. Interviennent alors les services qui vont quant à eux permettre aux clients d'échanger de manière plus fiable avec les conteneurs s'exécutant dans le Pod à l'aide d'une adresse IP virtuelle statique grâce au travail du composant `kube-proxy` .

Il existe quatre types de services pour une utilisation particulière :

- `ClusterIP` : C'est le type par défaut. Il expose le Service sur une adresse IP interne du cluster. De ce fait, le service n'est accessible que depuis l'intérieur du cluster.
- `NodePort` : Il expose le service vers l'extérieur du cluster à l'aide du NAT (la plage de ports autorisés est entre 30000 et 32767).

Appartient à devopssec.fr - lien original ici

- `LoadBalancer` : Il utilise l'équilibreur de charge des fournisseurs de cloud. Ainsi, les services NodePort et ClusterIP sont créés automatiquement et sont acheminés par l'équilibreur de charge externe.
- `ExternalName` : Ce service effectue une simple redirection CNAME (par exemple rediriger le trafic vers le nom de domaine "example.com").

Sans transition, commençons directement par la manipulation des services.

Manipulation des Services

Dans cet article nous n'étudierons que le type `ClusterIP` et `NodePort` qui sont les plus communément utilisés.

ClusterIP

Nous allons commencer par manipuler le service `ClusterIP`, utilisé par défaut sur Kubernetes.

Pour ce chapitre, j'utiliserai ma propre image hajdaini/flask avec le tag `random` qui permet d'afficher depuis une requête http, le nom du conteneur et une chaîne de caractères aléatoires ce qui aura pour but d'analyser plus facilement l'acheminement du trafic effectué par l'agent `kube-proxy`.

Créons d'abord notre template sous le format YAML de manière à **construire notre propre service**, avec le contenu suivant :

```
apiVersion: v1
kind: Service
metadata:
  name: flask-service
spec:
  type: ClusterIP
  selector:
```

Appartient à devopssec.fr - lien original ici

```
    app: flask
  ports:
  - port: 5000
    targetPort: 5000
    name: flask-cluster-ip
```

Place maintenant aux explications :

```
type: ClusterIP
```

Sans surprise, c'est ici que nous définissons le type de notre service.

```
selector:
  app: flask
```

Ici, nous indiquons que le service `flask-service` ne sera utilisé que par les Pods ayant le label `app: flask`.

```
- port: 5000
  targetPort: 5000
  name: flask-cluster-ip
```

Dans cette phase, le `targetPort` est le port qui est utilisé par les Pods de notre Deployment et le `port` est le numéro de port qui sera utilisé par notre service afin de communiquer avec lui depuis l'intérieur de notre cluster K8.

> **Information**
>
> Sur docker ça correspond à `targetPort:port` de la commande `docker run`.

Dorénavant, nous allons **exécuter la commande suivante dans le but de créer notre Service Kubernetes** :

```
kubectl create -f service.yaml
```

Vérifions ensuite la liste des services disponibles dans notre cluster Kubernetes :

```
kubectl get svc
```

Résultat :

```
NAME              TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
flask-service     ClusterIP   10.102.220.242   <none>        5000/TCP
kubernetes        ClusterIP   10.96.0.1        <none>        443/TCP
```

Nous avons donc le service `kubernetes` qui est créé par défaut par K8s et notre service `flask-service` qui écoute sur l'IP `10.102.220.242` et sur le port `5000`.

Ensuite, nous construisons notre Deployment qui utilisera mon image `hajdaini/flask:random` en deux répliques :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flask-deployment
spec:
  selector:
    matchLabels:
      app: flask
  replicas: 2
  template:
    metadata:
      labels:
        app: flask
    spec:
      containers:
      - name: flask-random
        image: hajdaini/flask:random
```

Comme vous pouvez le constater, nous avons spécifié le label `app: flask`, afin qu'il soit pris en compte par notre Service `flask-service`. Dès lors, créons notre Deployment :

```
kubectl create -f flask.yaml
```

À présent, récupérons le nom des Pods de notre Deployment :

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

Résultat :

```
NAME                                READY   STATUS    RESTARTS   AGE     IP
flask-deployment-8cfb9777b-s8hnp    1/1     Running   0          3m58s   192.168.1.17
flask-deployment-8cfb9777b-wdxt9    1/1     Running   0          3m58s   192.168.1.17
```

Comme je l'ai précisé plus haut, le service de type ClusterIP n'est accessible que depuis l'intérieur de notre cluster. Je vais donc devoir passer par le protocole ssh depuis mon nœud dénommer `worker-1` dans l'intention de communiquer avec mon service, comme suit :

```
ssh vagrant@[WORKER_IP] curl -s http://[CLUSTER_IP]:5000
```

> **Rappel**
>
> l'Ip de mon nœud `worker-1` est 192.168.50.11.

Soit, dans mon cas :

```
ssh vagrant@192.168.50.11 curl -s http://10.102.220.242:5000
```

Résultat :

```
My name is flask-deployment-8cfb9777b-wdxt9 and I say to you "WVYVOS"
```

On remarque que c'est le Pod nommé `flask-deployment-8cfb9777b-wdxt9` qui répond à notre requête depuis le service `flask-service` sur le port 5000.

Maintenant utilisons une boucle `for` afin de vérifier comment l'agent `kube-proxy` achemine notre trafic :

Appartient à devopssec.fr - lien original ici

```
ssh vagrant@192.168.50.11 "for i in {1..5}; do curl -s http://10.102.220.242:5000 &&
```

Résultat :

```
My name is flask-deployment-8cfb9777b-s8hnp and I say to you "XJY8F9"
My name is flask-deployment-8cfb9777b-wdxt9 and I say to you "Z25FZC"
My name is flask-deployment-8cfb9777b-s8hnp and I say to you "UV41OH"
My name is flask-deployment-8cfb9777b-wdxt9 and I say to you "A4MKPX"
My name is flask-deployment-8cfb9777b-wdxt9 and I say to you "V802S9"
```

On peut s'apercevoir qu'un LoadBalancer très basique est adopté par l'agent `kube-proxy` afin de rediriger le trafic sur les différents Pods de notre Deployment.

À partir de maintenant , vous pouvez détruire et recréer vos Pods et communiquer avec eux sereinement, sans vous soucier de leur nouvelle adresse IP, puisque dorénavant vous utilisez l'adresse IP de votre ClusterIP, offrant ainsi une meilleure stabilité.

NodePort

Imaginons, que nous avons besoin cette fois-ci d'accéder à notre Pod depuis l'extérieur de notre cluster Kubernetes. Nous exploiterons ainsi cette le type `NodePort`.

Sans la nécessité de détruire notre ancien service, nous allons modifier et appliquer les changements de notre ancien template YAML, en suivant les étapes suivantes :

Premièrement modifions notre ancien template YAML, comme suit :

```
apiVersion: v1
kind: Service
metadata:
  name: flask-service
spec:
  type: NodePort
  selector:
    app: flask
```

Appartient à devopssec.fr - lien original ici

```
  ports:
  - port: 5000
    targetPort: 5000
    nodePort: 30001
    name: flask-np
```

Voici quelques explications sur nos nouveaux changements :

```
type: NodePort
```

Ici, Nous changeons d'abord le type de notre service.

```
ports:
- port: 5000
  targetPort: 5000
  nodePort: 30001
  name: flask-np
```

Ensuite, dans cette étape, nous translatons l'IP du service vers l'IP du nœud qui accueillera nos Pods et le port du service (ici 5000) vers le port 30001.

> **Information**
>
> Pour rappel la plage d'IP dans un service de type NodePort se situe entre 30000 et 32767. On ne peut donc pas dépasser cette limite dans le champ `nodePort`.

Deuxièmement, **appliquons les changements de notre service** :

```
kubectl apply -f flask.yaml
```

Résultat :

```
service/flask-service configured
```

Vérifions ensuite la liste des services disponibles dans note cluster :

```
kubectl get svc
```

Résultat :

```
NAME              TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
flask-service     NodePort    10.102.220.242   <none>        5000:3000
kubernetes        ClusterIP   10.96.0.1        <none>        443/TCP
```

Désormais, nous somme capables d'accéder à nos Pods depuis l'extérieur de notre cluster à travers l'IP de notre nœud (dans mon cas c'est le nœud nommé `worker-1`), comme suit :

```
curl -s http://192.168.50.11:30001
```

Résultat :

```
My name is flask-deployment-8cfb9777b-nwhrl and I say to you "P68NVW"
```

La commande kubectl expose

Plutôt que de rédiger un template YAML de notre service, on peut **créer un service à l'aide de la commande** `kubectl expose` afin d'exposer les pods notre Deployment.

Par exemple pour exposer notre Deployment `flask-deployment` à l'aide d'un service de type ClusterIP, comme vu précédemment, nous suivrons les étapes suivantes :

D'abord je commence par **supprimer mon service Kubernetes** `flask-service`. Pour ce faire nous lançons la commande suivante :

```
kubectl delete svc flask-service
```

Après ceci, on expose les pods de notre Deployment :

```
kubectl expose deployment flask-deployment --name flask-service \              1 ?
    --type ClusterIP --protocol TCP --port 5000 --target-port 5000 --selector='app=f
```

Voici la commande pour exposer notre Deployment avec un service de type NodePort :

```
kubectl expose deployment flask-deployment --name flask-service \
    --type NodePort --protocol TCP --port 5000 --target-port 5000 --selector='app=fla
```

Conclusion

On peut constater que les services restent à bon moyen de gérer plus efficacement la partie réseau de nos Pods. Comme pour chaque fin d'article, voici un **antisèche des commandes des services kubernetes** :

```
# Afficher la liste des Services
kubectl get service [En option <SERVICE NAME>]

# Créer un Service depuis un template
kubectl create -f <template.yaml>

# Créer un Service depuis sans template
kubectl expose deployment <DEPLOYMENT NAME>
    --name : nom du service
    --type : type du service
    --protocol : protocole à utiliser (TCP/UDP)
    --port : port utilisé par le service
    --target-port : port utilisé utilisé par le Pod
    --selector='clé=valeur': le sélecteur utilisé par service

# Supprimer un Service
kubectl delete service <SERVICE NAME>

# Appliquer des nouveaux changements à votre Service sans le détruire
kubectl apply -f <template.yaml>

# Modifier et appliquer les changements de votre Service instantanément sans le détru
kubectl edit service <SERVICE NAME>

# Afficher les détails d'un Service
kubectl describe service <SERVICE NAME>
```