Lab 5 : Communiquer avec les Pods via des règles de routage

Dans l'exercice précédent, nous avons étudié les `Services` via les objets `ClusterIP` et `NodePort`. Nous avons montré que le `Service ClusterIP` ne permettait pas d'accéder aux `Pods` depuis l'extérieur du cluster et que le `Service NodePort` nécessitait de réserver un port réseau sur tout le cluster même si aucun `Pod` ne tournait. Il existe également un autre type de `Service` appelait `LoadBalancer` qui est difficilement configurable pour un cluster qui n'est pas dans le Cloud (notre cas actuellement).

Nous présentons dans cet exercice les `Ingress`, une solution qui s'appuie sur les `Services`. Un `Ingress` est une règle qui relie une URL à un objet `Service`. Par ailleurs, un `Ingress` utilise un contrôleur `Ingress` pour piloter un `Reverse Proxy` pour implémenter les règles. **Toutefois, un `Ingress` n'est pas un objet de type `Service`**. L'objectif visé par un `Ingress` est d'éviter d'utiliser un `Service` de type `NodePort` ou `LoadBalancer` pour communiquer depuis l'extérieur d'un cluster Kubernetes, seul un `Service` de type `ClusterIP` sera suffisant.

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

- Écrire une configuration `Ingress` ;
- Créer des règles de type `fanout` ou hôtes virtuels ;
- Gérer des hôtes virtuels.

Étapes à suivre

- Avant de commencer les étapes de cet exercice, assurez-vous que le `Namespace` créé dans l'exercice précédent `mynamespaceexercice3` soit supprimé.

```
$ kubectl delete namespace mynamespaceexercice3
namespace "mynamespaceexercice3" deleted
```

- Créer dans le répertoire *exercice4-ingress/* un fichier appelé *mynamespaceexercice4.yaml* en ajoutant le contenu suivant :

```
apiVersion: v1
kind: Namespace
metadata:
  name: mynamespaceexercice4
```

- Créer ce `Namespace` dans notre cluster :

```
$ kubectl apply -f exercice4-ingress/mynamespaceexercice4.yaml
namespace/mynamespaceexercice4 created
```

Dans les étapes suivantes, nous allons créer deux `Deployments` afin de simuler l'existence de deux applications différentes (microservices). Les deux `Deployment` seront basés sur la même image Docker Nginx pour déployer des `Pods`. Pour différencier les deux applications, nous modifierons la page *index.html* de chaque conteneur afin d'identifier clairement l'application visée par nos requêtes. Pour accéder à ces applications (microservices) depuis l'extérieur du cluster, nous utiliserons un `Ingress` avec deux règles pour les relier aux URL suivantes : http://<IP_NODE>/app1 pour la première application et http://<IP_NODE>/app2 pour la seconde application. Cette forme de configuration est appelée `fanout` et permet de définir une règle à base de sous-chemins.

- Créer dans le répertoire *exercice4-ingress/* un fichier appelé *app1deployment.yaml* qui décrit un `Deployment` et un `Service ClusterIP` pour la première application :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app1deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: app1pod
  template:
    metadata:
      labels:
        app: app1pod
    spec:
      containers:
      - name: app1container
        image: nginx:latest
        ports:
        - containerPort: 80
        lifecycle:
          postStart:
            exec:
              command:
                - /bin/sh
                - -c
                - >
                  mkdir /usr/share/nginx/html/app1;
                  echo App 1 fanout from $HOSTNAME >
/usr/share/nginx/html/app1/index.html;
                  echo App 1 vhosts from $HOSTNAME >
/usr/share/nginx/html/index.html
```

```
---

apiVersion: v1
kind: Service
metadata:
  name: app1service
spec:
  selector:
    app: app1pod
  type: ClusterIP
  ports:
    - protocol: TCP
      targetPort: 80
      port: 8080
```

Vous noterez dans la partie `command` la création du répertoire *app1* (`mkdir /usr/share/nginx/html/app1`). Cela est nécessaire pour que la requête puisse aboutir, problématique courante quand il est nécessaire de déployer avec des sous-chemins.

- Créer dans le répertoire *exercice4-ingress/* un fichier appelé *app2deployment.yaml* qui décrit un `Deployment` et un `Service ClusterIP` pour la seconde application :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app2deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: app2pod
  template:
    metadata:
      labels:
        app: app2pod
    spec:
      containers:
      - name: app2container
        image: nginx:latest
        ports:
        - containerPort: 80
        lifecycle:
          postStart:
            exec:
              command:
                - /bin/sh
                - -c
                - >
                  mkdir /usr/share/nginx/html/app2;
                  echo App 2 fanout from $HOSTNAME >
/usr/share/nginx/html/app2/index.html;
                  echo App 2 vhosts from $HOSTNAME >
/usr/share/nginx/html/index.html
```

```
---

apiVersion: v1
kind: Service
metadata:
  name: app2service
spec:
  selector:
    app: app2pod
  type: ClusterIP
  ports:
    - protocol: TCP
      targetPort: 80
      port: 8080
```

- Appliquer les deux configurations précédentes pour créer les `Deployments` et les `Services` dans le cluster Kubernetes :

```
$ kubectl apply -f exercice4-ingress/app1deployment.yaml -n
mynamespaceexercice4
deployment.apps/app1deployment created
service/app1service created
$ kubectl apply -f exercice4-ingress/app2deployment.yaml -n
mynamespaceexercice4
deployment.apps/app2deployment created
service/app2service created
```

- Créer dans le répertoire *exercice4-ingress/* un fichier appelé *myingressfanout.yaml* qui décrit un `Ingress` :

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: myingressfanout
spec:
  rules:
    - http:
        paths:
        - path: /app1
          pathType: Prefix
          backend:
            service:
              name: app1service
              port:
                number: 8080
        - path: /app2
          pathType: Prefix
          backend:
            service:
              name: app2service
              port:
                number: 8080
```

Deux règles sont définies. La première traite du chemin (`path`) `/app1` et s'applique au `Service app1service` et la seconde traite du chemin (`path`) `/app2` et s'applique au `Service app2service`.

- Appliquer cette configuration pour créer cet `Ingress` dans le cluster Kubernetes :

```
$ kubectl apply -f exercice4-ingress/myingressfanout.yaml -n
mynamespaceexercice4
ingress.networking.k8s.io/myingressfanout created
```

- Afficher le détail complet de cet `Ingress` via l'option `describe` pour s'assurer que tout est configuré correctement :

```
$ kubectl describe ingress -n mynamespaceexercice4 myingressfanout
Name:             myingressfanout
Labels:           <none>
Namespace:        mynamespaceexercice4
Address:          172.29.0.3,172.29.0.4,172.29.0.5
Ingress Class:    traefik
Default backend:  <default>
Rules:
  Host        Path  Backends
  ----        ----  --------
  *
              /app1   app1service:8080 (10.42.0.11:80,10.42.2.16:80)
              /app2   app2service:8080 (10.42.1.10:80,10.42.2.17:80)
Annotations:  <none>
Events:       <none>
```

Il ne reste plus qu'à tester les deux règles en effectuant des requêtes vers le nœud maître (ou nœud de travail) via l'outil **cURL**.

Via K3d

- Lister l'ensemble des conteneurs disponibles :

```
$ docker ps
CONTAINER ID   IMAGE                          COMMAND                  CREATED          STATUS
PORTS                                                                NAMES
e3156d411390   d0554070bc8c                   "/bin/sh -c nginx-pr…"   5 minutes ago    Up 5 minutes
80/tcp, 0.0.0.0:30001->30001/tcp, 0.0.0.0:62002->6443/tcp   k3d-mycluster-serverlb
b7e9d52a3d89   rancher/k3s:v1.21.7-k3s1   "/bin/k3d-entrypoint…"   16 minutes ago   Up 16 minutes
k3d-mycluster-agent-1
e4848c17c6cd   rancher/k3s:v1.21.7-k3s1   "/bin/k3d-entrypoint…"   16 minutes ago   Up 16 minutes
k3d-mycluster-agent-0
6ae3be322c8c   rancher/k3s:v1.21.7-k3s1   "/bin/k3d-entrypoint…"   16 minutes ago   Up 16 minutes
k3d-mycluster-server-0
```

- Actuellement aucun conteneur du cluster K8s ne peut répondre à une requête sur le port `80`. Modifier le cluster K3d afin d'ajouter l'écoute sur ce port :

```
$ k3d cluster edit mycluster --port-add 80:80@loadbalancer
INFO[0000] portmapping '80:80' targets the loadbalancer: defaulting to
[servers:*:proxy agents:*:proxy]
INFO[0000] Renaming existing node k3d-mycluster-serverlb to k3d-mycluster-
serverlb-gNOwY...
INFO[0000] Creating new node k3d-mycluster-serverlb...
INFO[0000] Stopping existing node k3d-mycluster-serverlb-gNOwY...
INFO[0010] Starting new node k3d-mycluster-serverlb...
INFO[0010] Starting Node 'k3d-mycluster-serverlb'
INFO[0017] Deleting old node k3d-mycluster-serverlb-gNOwY...
INFO[0017] Successfully updated mycluster
```

```
$ docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED         STATUS
PORTS                                                                                  NAMES
017f86faa708   ef33158baf49                           "/bin/sh -c nginx-pr…"   2 minutes ago   Up
About a minute   0.0.0.0:80->80/tcp, 0.0.0.0:30001->30001/tcp, 0.0.0.0:61868->6443/tcp   k3d-
mycluster-serverlb
218cfd215045   ghcr.io/k3d-io/k3d-tools:5.4.7   "/app/k3d-tools noop"    5 hours ago     Up 5
hours                                                                                  k3d-
mycluster-tools
fc1b05755fa1   rancher/k3s:v1.25.6-k3s1          "/bin/k3d-entrypoint…"   5 hours ago     Up 5
hours                                                                                  k3d-
mycluster-agent-1
881b02b75046   rancher/k3s:v1.25.6-k3s1          "/bin/k3d-entrypoint…"   5 hours ago     Up 5
hours                                                                                  k3d-
mycluster-agent-0
ccb827ca9afd   rancher/k3s:v1.25.6-k3s1          "/bin/k3d-entrypoint…"   5 hours ago     Up 5
hours                                                                                  k3d-
mycluster-server-0
```

Le port `80` du cluster K8s est maintenant exposé sur le port `80` du poste de développeur.

- Toutes requêtes sur le port `80` du poste du développeur sont transférées vers le contrôleur `Ingress` :

```
$ curl localhost:80/app1/
App 1 from app1deployment-95f49fb56-d5pj5
$ curl localhost/app2/
App 2 from app2deployment-567484c687-tbvxv
```

Les `Ingress` permettent également de gérer les hôtes virtuels pour éviter d'utiliser les sous-chemins (`fanout`). Ainsi, au lieu d'utiliser cette forme d'URL http://<IP_NODE>/app1 nous allons plutôt utiliser celle-ci http://app1.mydomain.test. Toutefois, puisque nous ne disposons pas du domaine http://mydomain.test nous allons devoir configurer le poste du développeur pour que les requêtes envoyées au cluster soient satisfaites.

Via K3d

- Éditer le fichier */etc/hosts* en ajoutant les deux lignes suivantes :

```
...
127.0.0.1 app1.mydomain.test
127.0.0.1 app2.mydomain.test
```

- Créer dans le répertoire *exercice4-ingress/* un fichier appelé *myingressvhosts.yaml* qui décrit un `Ingress` en utilisant de hôtes virtuels :

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: myingressvhosts
spec:
  rules:
    - host: "app1.mydomain.test"
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: app1service
                port:
                  number: 8080
    - host: "app2.mydomain.test"
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: app2service
                port:
                  number: 8080
```

- Appliquer cette configuration pour créer cet `Ingress` dans le cluster Kubernetes :

```
$ kubectl apply -f exercice4-ingress/myingressvhosts.yaml -n
mynamespaceexercice4
ingress.networking.k8s.io/myingressvhosts created
```

- Tester les deux règles associées à des hôtes virtuels en effectuant des requêtes via l'outil **cURL** :

```
$ curl app1.mydomain.test
App 1 vhosts from app1deployment-6887b85fc9-c2wg8
$ curl app2.mydomain.test
App 2 vhosts from app2deployment-6c8d974467-njw55
```

Bilan de l'exercice

À cette étape, vous savez :

- créer des `Ingress` ;
- créer des règles de type `fanout` ou hôtes virtuels ;
- Configurer son poste de développeur pour répondre à des requêtes de hôtes virtuels.

Pour continuer sur les concepts présentés dans cet exercice, nous proposons les expérimentations suivantes :

- créer un `Deployment` basé sur une image Docker Apache HTTP et définir trois `ReplicaSets` ;
- créer un `Service` de type `ClusterIP` pour ce `Deployment`;
- créer un `Ingress` pour s'appliquer à ce `Service ClusterIP` et pour gérer l'hôte virtuel http://apache.mydomain.test.