

TP4 - Kubernetes



TP sur Kubernetes

Sommaire

- [TP4 - Kubernetes](#)
 - [Sommaire](#)
 - [Création d'un cluster Kubernetes](#)
 - [Introduction](#)
 - [Mise en place de l'environnement](#)
 - [Déploiement de l'application](#)
 - [Déploiement de MongoDB](#)
 - [Connexion à MongoDB](#)
 - [1\) Déploiement de l'API](#)
 - [2\) Connexion de l'API à MongoDB](#)
 - [3\) Exposer l'API à l'interieur du cluster à l'aide d'un service](#)
 - [4\) Augmenter le nombre de réplicas de l'API](#)
 - [5\) Installer l'Ingress Controller NGINX](#)
 - [6\) Exposer l'API publiquement en utilisant un Ingress](#)
 - [7\) Réalisation du schéma réseau](#)
 - [Annexe](#)
 - [Installation de kubectl](#)
 - [Identifiant du cluster](#)
 - [Résolution de l'erreur "ingress does not contain a valid IngressClass"](#)

Création d'un cluster Kubernetes

Introduction

Dans ce TP nous devons créer un cluster Kubernetes qui contient une API. Le déploiement se fait sur le Cloud Provider Azure en utilisant le service Azure Kubernetes Service (AKS).

Mise en place de l'environnement

Pour commencer, nous devons créer un cluster Kubernetes sur Azure. Pour cela, nous allons utiliser l'outil [Azure CLI](#). Pour l'installer, il suffit de suivre les instructions sur le lien ci-dessus.

Il faut également l'outil [kubectl](#). Il permet de gérer un cluster Kubernetes.

Pour la visualisation du cluster nous allons utiliser [Lens](#). Il permet de visualiser les ressources d'un cluster Kubernetes.

Je passe les explications sur la création du cluster, car elles sont déjà présentes dans le sujet du TP.

Déploiement de l'application

Déploiement de MongoDB

Pour déployer MongoDB, nous allons utiliser l'outil [Helm](#). Helm est un gestionnaire de paquets pour Kubernetes. Il permet de déployer des applications sur Kubernetes.

Installer Helm avec la commande suivante :

```
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 |  
bash
```

Vérifier que Helm est bien installé avec la commande suivante :

```
helm version
```

Note à savoir :

Un chart Helm est un package qui contient les ressources Kubernetes nécessaires pour déployer une application sur un cluster Kubernetes. Un chart peut être créé manuellement en utilisant des modèles Go, ou en utilisant un chart existant maintenu par la communauté. Un chart Helm est essentiellement une liste de ressources Kubernetes définies sous forme de modèles Go qui peuvent être utilisés pour déployer une application sur un cluster Kubernetes.

Il existe un chart Helm permettant de déployer MongoDB sur Kubernetes.

Il faut dans un premier temps configurer le registry où est stocké le Chart avec la commande suivante :

```
helm repo add bitnami https://charts.bitnami.com/bitnami
```

Il faut ensuite installer une [release](#) de ce Chart en utilisant la commande suivante :

```
helm install mongodb bitnami/mongodb -n mongodb --create-namespace
```

Cette commande va installer le Chart Helm dans le namespace [mongodb](#) et créer toutes les ressources Kubernetes nécessaires. Dans Kubernetes, un [namespace](#) est un espace logique permettant de stocker des ressources Kubernetes. C'est pratique pour organiser les ressources et gérer les privilèges. Par défaut, si le namespace n'est pas spécifié, la ressource Kubernetes sera créée dans le namespace [default](#).

Les ressources créées sont les suivantes :

- Un [Deployment](#) : cette ressource définit la configuration de l'application et comment elle doit être exécutée sur le cluster. Le déploiement Kubernetes gère également la mise à l'échelle et la redémarrage de l'application en cas de panne.

- Un **Service** : cette ressource expose l'application sur le réseau du cluster et permet aux autres applications de se connecter à celle-ci.
- Un **Persistent Volume** : cette ressource permet de stocker les données de l'application de manière permanente, même si l'instance de l'application est redémarrée ou mise à l'échelle.
- Des **Secrets** : cette ressource permet de stocker et de gérer des informations sensibles, telles que les mots de passe et les clés de chiffrement, de manière sécurisée.
- Des annotations de chart Helm : cette ressource permet de stocker des informations sur le déploiement de l'application, telles que la version du chart Helm utilisée et les valeurs de configuration utilisées.

Connexion à MongoDB

Pour se connecter à MongoDB, il faut utiliser le port **27017** du service **mongodb** du namespace **mongodb**. Pour cela, il faut utiliser la commande suivante :

```
kubectl port-forward service/mongodb -n mongodb 27017:27017
```

On peut récupérer le mot de passe de l'utilisateur **root** avec la commande suivante :

```
kubectl get secret --namespace mongodb mongodb -o jsonpath="{.data.mongodb-root-password}" | base64 --decode
```

1) Déploiement de l'API

Pour déployer l'API, nous allons utiliser le fichier **api-deployment.yaml**. Ce fichier contient les ressources Kubernetes nécessaires pour déployer l'API.

Voici le contenu du fichier :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-deployment
  labels:
    app: api
spec:
  replicas: 1
  selector:
    matchLabels:
      app: api
  template:
    metadata:
      labels:
        app: api
```

```
spec:
  containers:
    - name: api
      image: yrez/ynov-api:latest
      ports:
        - containerPort: 3000
```

Les mots-clés principaux du fichier sont les suivants :

- **apiVersion** : Version de l'API Kubernetes utilisée. Ici, nous utilisons la version **apps/v1** qui est utilisée pour les objets **Deployment**.
- **kind** : Type de ressource Kubernetes. Ici, nous utilisons le type **Deployment** qui permet de gérer les déploiements d'applications.
- **metadata** : Métadonnées de la ressource Kubernetes. Ici, nous utilisons le nom **api-deployment** pour le déploiement et le label **app: api** pour le déploiement et les pods.
- **spec** : Spécification de la ressource Kubernetes. Ici, nous utilisons le nombre de réplicas à 1, le label **app: api** pour le déploiement et les pods, le nom de l'image Docker **yrez/ynov-api:latest** et le port **3000** du container.

Pour déployer l'API, il faut utiliser la commande suivante :

```
kubectl apply -f api-deployment.yaml
```

2) Connexion de l'API à MongoDB

A ce stade, l'API est déployée mais ne peut pas se connecter à MongoDB. Il faut créer un **secret** Kubernetes qui contient les informations de connexion à MongoDB.

Pour cela, on utilise la commande suivante :

```
kubectl create secret generic mongodb-uri --from-  
literal=MONGODB_URI=mongodb://root:JWjvXqosfn@mongodb.mongodb:27017
```

Ici on utilise **mongodb.mongodb** comme nom d'hôte car le service **mongodb** du namespace **mongodb** est accessible à l'intérieur du cluster Kubernetes avec ce nom d'hôte.

Il faut ensuite modifier le fichier **api-deployment.yaml** pour ajouter le **secret** dans le container de l'API.

Voici le contenu du fichier :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-deployment
  labels:
    app: api
```

```
spec:
  replicas: 1
  selector:
    matchLabels:
      app: api
  template:
    metadata:
      labels:
        app: api
    spec:
      containers:
        - name: api
          image: yrez/ynov-api:latest
          ports:
            - containerPort: 3000
          env:
            - name: MONGODB_URI
              valueFrom:
                secretKeyRef:
                  name: mongodb-uri
                  key: MONGODB_URI
                  optional: false
```

Les mots-clés principaux ajoutés sont les suivants :

- **env** : Liste des variables d'environnement du container. Ici, nous utilisons la référence au secret **mongodb-uri** pour récupérer la variable d'environnement **MONGODB_URI**. De plus, **optional: false** permet de rendre la variable d'environnement obligatoire.

Il ne reste plus qu'à déployer l'API avec la commande suivante :

```
kubectl apply -f api-deployment.yaml
```

3) Exposer l'API à l'intérieur du cluster à l'aide d'un service

Dans cette étape nous allons exposer l'API à l'intérieur du cluster à l'aide d'un service Kubernetes.

Pour cela, nous allons utiliser le fichier **api-service.yaml**. Ce fichier contient les ressources Kubernetes nécessaires pour exposer l'API à l'intérieur du cluster.

Voici le contenu du fichier :

```
apiVersion: v1
kind: Service
metadata:
  name: api-service
spec:
  selector:
    app: api
```

```
ports:
  - protocol: TCP
    port: 3000
```

Les mots-clés principaux du fichier sont les suivants :

- **apiVersion** : Version de l'API Kubernetes utilisée. Ici, nous utilisons la version **v1** qui est utilisée pour les objets **Service**.
- **kind** : Type de ressource Kubernetes. Ici, nous utilisons le type **Service** qui permet de gérer les services d'applications.
- **metadata** : Métadonnées de la ressource Kubernetes. Ici, nous utilisons le nom **api-service** pour le service.
- **spec** : Spécification de la ressource Kubernetes. Ici, nous utilisons le label **app: api** pour le service et les pods, le port **3000** du container et le protocole **TCP**.

4) Augmenter le nombre de réplicas de l'API

Pour augmenter le nombre de réplicas de l'API, il faut modifier le fichier **api-deployment.yaml** et changer la valeur de **replicas** à 5.

Ne pas oublier de déployer l'API avec la commande suivante :

```
kubectl apply -f api-deployment.yaml
```

5) Installer l'Ingress Controller NGINX

Dans cette étape, nous allons installer l'Ingress Controller NGINX qui permet d'exposer des services à l'extérieur du cluster Kubernetes.

Pour cela, nous allons utiliser Helm.

Voici les étapes à suivre :

- Ajouter le dépôt Helm de NGINX :

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
```

- Mettre à jour les dépôts Helm :

```
helm repo update
```

- Installer l'Ingress Controller NGINX :

```
helm install ingress-nginx ingress-nginx/ingress-nginx
```

Les ressources Kubernetes suivantes sont créées :

- Un **Deployment** qui définit la configuration du pod exécutant l'Ingress Controller Nginx
- Un **Service** qui expose l'Ingress Controller Nginx sur un port spécifique du cluster
- Un **ConfigMap** qui contient la configuration de Nginx utilisée par l'Ingress Controller
- Un **Ingress** qui définit les règles de routage entrantes vers les services de votre application
- Un **Secret** qui contient les certificats SSL utilisés pour chiffrer les communications entrantes vers l'Ingress Controller

Je récupère l'adresse IP du LoadBalancer avec la commande suivante :

```
kubectl get svc ingress-nginx-controller -o  
jsonpath='{.status.loadBalancer.ingress[0].ip}'
```

Puis je teste la connexion avec la commande suivante :

```
curl 51.138.216.98
```

On tombe sur la page 404 Not Found de NGINX. C'est normal, nous n'avons pas encore exposé l'API publiquement.

6) Exposer l'API publiquement en utilisant un Ingress

Maintenant nous allons exposer l'API publiquement en utilisant un Ingress.

Pour cela, nous allons utiliser le fichier **api-ingress.yaml**. Ce fichier contient les ressources Kubernetes nécessaires pour exposer l'API publiquement.

En sachant que les Ingress fonctionnent avec des noms de domaine, nous allons utiliser le service nip.io qui permet de créer des noms de domaine à partir d'adresses IP.

Le nom de domaine que nous allons utiliser est le suivant : **api.51.138.216.98.nip.io**.

Voici le contenu du fichier :

```
apiVersion: networking.k8s.io/v1  
kind: Ingress  
metadata:  
  name: api-ingress  
spec:  
  ingressClassName: nginx  
  rules:  
    - host: 'api.51.138.216.98.nip.io'  
      http:  
        paths:  
          - pathType: Prefix
```

```
path: '/'
backend:
  service:
    name: api-service
    port:
      number: 3000
```

Les mots-clés principaux du fichier sont les suivants :

- **apiVersion** : Version de l'API Kubernetes utilisée. Ici, nous utilisons la version **v1** qui est utilisée pour les objets **Ingress**.
- **kind** : Type de ressource Kubernetes. Ici, nous utilisons le type **Ingress** qui permet de gérer les Ingress.
- **metadata** : Métadonnées de la ressource Kubernetes. Ici, nous utilisons le nom **api-ingress** pour l'Ingress.
- **spec** : Spécification de la ressource Kubernetes. Ici, nous utilisons le nom de domaine **api.51.138.216.98.nip.io** pour l'Ingress, le chemin **/** pour l'Ingress et le port **3000** du service **api-service**.

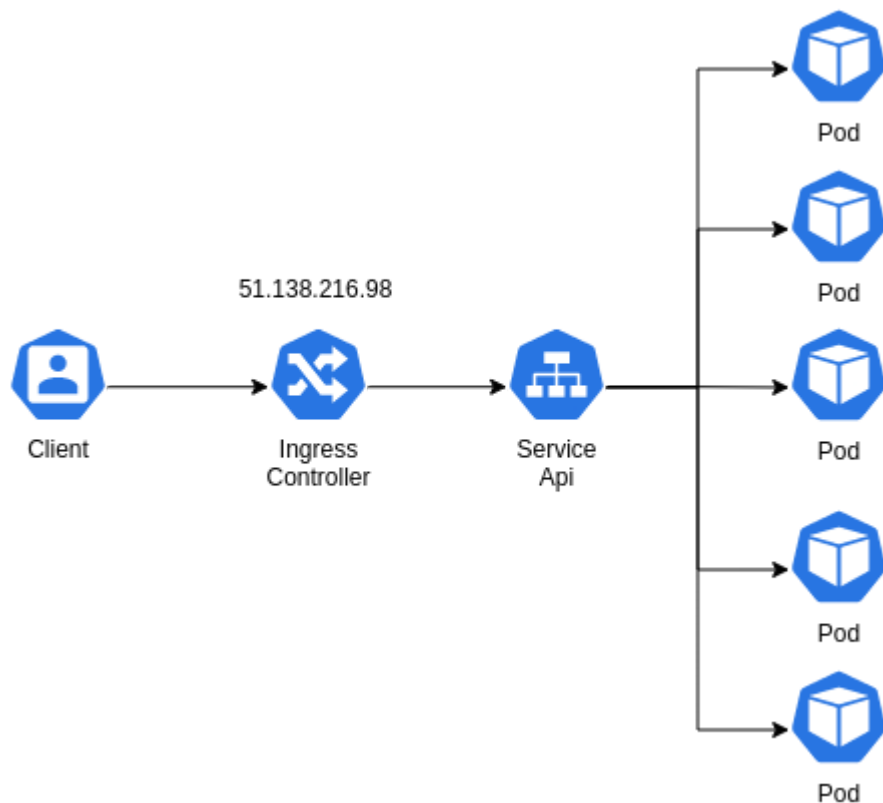
Pour une raison inconnue, j'ai eu l'erreur suivante :

```
ingress does not contain a valid IngressClass
```

La solution a été de rajouter **ingressClassName: nginx** dans le fichier **api-ingress.yaml**.

7) Réalisation du schéma réseau

Voici le schéma réseau de notre application :



Ici le client se connecte à l'Ingress Controller NGINX en utilisant le nom de domaine `api.51.138.216.98.nip.io` et le chemin `/`. L'Ingress Controller NGINX redirige la requête vers le service `api-service` qui est exposé sur le port `3000`. Le service `api-service` redirige la requête vers un pod `api-deployment-xxxxx` qui est exposé sur le port `3000`.

Annexe

Installation de kubectl

- [Install and Set Up kubectl on Linux | Kubernetes](#)

Identifiant du cluster

```
/subscriptions/f9d87673-ab83-4fcf-a151-22964e496b40/resourcegroups/rg-tp4-kubernetes/providers/Microsoft.ContainerService/managedClusters/aks-tp4
```

Résolution de l'erreur "ingress does not contain a valid IngressClass"

- [Kubernetes IngressClass | error="ingress does not contain a valid IngressClass" | by Sam Thomas | Medium](#)