

GÉRER ET MANIPULER LES NAMESPACES ET LES RESOURCEQUOTAS

Introduction

C'est quoi un namespace ?

Kubernetes prend en charge plusieurs **clusters virtuels** sauvegardés par le même cluster physique. Ces clusters virtuels sont appelés namespaces.

Un namespace fournit **une étendue pour les noms des ressources Kubernetes**. Ceci est utile lorsque de nombreux utilisateurs répartis dans plusieurs équipes ou projets utilisent le même cluster et qu'il existe un risque de collision de noms. Ils sont aussi un bon moyen pour **diviser et limiter les ressources physiques du cluster** entre plusieurs utilisateurs via un système de quota que nous verrons à travers ce chapitre.

Conseils et restrictions d'utilisation :

- Il n'est pas nécessaire d'utiliser plusieurs namespaces simplement pour séparer des ressources légèrement différentes, telles que des versions différentes de la même application. Dans ce cas, préférez l'utilisation des labels de manière à distinguer les ressources.
- Les noms de ressources doivent être uniques dans un namespace, mais pas entre les différents namespaces.

- Les namespaces ne peuvent pas être imbriqués les uns dans les autres et chaque ressource Kubernetes ne peut figurer que dans un seul namespace.

Les namespaces par défauts

Vous ne le saviez peut-être pas, mais vous utilisiez des namespaces depuis le début de cours sans pour autant en être conscient. Afin de vérifier cela, nous allons **répertorier la liste des namespaces actuels de votre cluster**, en lançant la commande suivante :

```
kubectl get namespace
```

Résultat :

| NAME | STATUS | AGE |
|-----------------|--------|-----|
| default | Active | 25d |
| kube-node-lease | Active | 25d |
| kube-public | Active | 25d |
| kube-system | Active | 25d |

Comme vous pouvez le constater, notre cluster k8s utilise trois **namespaces par défaut** dans Kubernetes, soit :

- **default** : le namespace utilisé par défaut pour les objets Kubernetes dont on ne spécifie pas de namespace.
- **kube-system** : le namespace pour les objets créés et utilisés par le système Kubernetes.
- **kube-public** : ce namespace est créé automatiquement au cas où vous souhaitez que certaines de vos ressources soient visibles publiquement dans l'ensemble du cluster par tous les utilisateurs (y compris ceux non authentifiés).

Manipulation des namespaces

Créer des namespaces

Il existe **deux façons pour créer notre namespace**, soit en utilisant un template au format YAML, soit sans.

Dans cet exemple, nous allons créer un namespace nommé `dev-team` sans utiliser de template :

```
kubectl create namespace dev-team
```

Résultat :

```
namespace/dev-team created
```

Cette fois-ci, nous utiliserons la méthode avec le template au format YAML. Créez donc un fichier yaml et copiez le contenu suivant :

```
apiVersion: v1
kind: Namespace
metadata:
  name: prod-team
```

Ensuite, exécutez la commande suivante :

```
kubectl create -f namespace.yaml
```

Résultat :

```
namespace/prod-team created
```

Par la suite, vérifions si nos namespaces spécifiés plus haut ont bien été créés :

```
kubectl get namespace
```

C'est bien le cas :

```
NAME                STATUS    AGE
default             Active    25d
dev-team            Active    2m12s
kube-node-lease     Active    25d
kube-public         Active    25d
kube-system         Active    25d
prod-team           Active    8s
```

Utilisation des namespaces dans les objets Kubernetes

Là aussi, il existe deux manières pour utiliser vos ressources Kubernetes dans un namespace. Soit vous définissez dans la liste `metadata` le champ `namespace`, comme suit :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  namespace: prod-team
  labels:
    app: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```

Après cela, créez votre objet Kubernetes :

```
kubectl create -f deploy-prod.yaml
```

Soit, il suffit juste de rajouter l'option `-n` ou `--namespace` dans vos commandes kubectl, comme par exemple :

```
kubectl run alpine --image=alpine --namespace=prod-team -- sleep 1d
```

À présent, si on vérifie les pods disponibles dans notre namespace `prod-team`, nous aurons ainsi les trois pods créés précédemment :

```
kubectl get pods -n prod-team
```

Résultat :

| NAME | READY | STATUS | RESTARTS | AGE |
|------------------------|-------|---------|----------|-------|
| alpine-56b95f459-scbnw | 1/1 | Running | 0 | 17s |
| nginx-7bffc778db-272kx | 1/1 | Running | 0 | 4m24s |
| nginx-7bffc778db-m7nds | 1/1 | Running | 0 | 4m24s |

Définition de la préférence d'un namespace

Vous pouvez **enregistrer de manière permanente un namespace pour toutes les commandes kubectl**, sans la nécessité d'utiliser l'option `-n` :

```
kubectl config set-context --current --namespace=prod-team
```

Résultat :

```
Context "Kubernetes-admin@Kubernetes" modified.
```

Vous pouvez examiner le namespace par défaut grâce à la commande suivante :

```
kubectl config view | grep namespace
```

Résultat :

```
namespace: prod-team
```

Voici la **commande pour revenir au namespace par défaut** :

```
kubectl config set-context --current --namespace=default
```

Suppression d'un namespace

Si vous n'avez plus besoin de votre namespace, alors vous pouvez le **supprimer** avec la commande suivante :

```
kubectl delete namespaces prod-team
```

Définir des quotas

Introduction à l'objet Kubernetes ResourceQuotaobjet

Lorsque plusieurs utilisateurs ou équipes partagent un cluster, il se peut qu'une équipe puisse utiliser plus que sa juste part de ressources. Dans ce cas de figure, il est recommandé d'utiliser l'objet **ResourceQuotaobjet** permettant aux administrateurs de résoudre ce type problème, en limitant la consommation ou la quantité de ressources définies dans votre namespace.

Fonctionnement d'un ResourceQuotaobjet dans un namespace

1. L'administrateur crée un ResourceQuotaobjet pour chaque namespace.
2. Les utilisateurs créent des objets Kubernetes (pods, services, etc.) dans un namespace et le système de quotas surveille la consommation des ressources pour s'assurer que celle-ci ne dépasse pas les limites matérielles définies dans

le ResourceQuotaobjet.

3. Si la création ou la mise à jour d'une ressource viole une contrainte de quota, la demande échouera et l'API renvoie le code HTTP 403 FORBIDDEN avec un message expliquant la contrainte qui aurait été violée.

Voici un exemple de quelques types de ressources qui sont pris en charge :

| Nom de la ressource | Description |
|------------------------------|---|
| <code>limits.cpu</code> | Pour tous les pods dans un état non terminal, la somme des limites de la CPU ne peut pas dépasser cette valeur. |
| <code>requests.cpu</code> | Pour tous les pods dans un état non terminal, la somme des requêtes de CPU ne peut pas dépasser cette valeur. |
| <code>limits.memory</code> | Pour tous les pods dans un état non terminal, la somme des limites de mémoire ne peut pas dépasser cette valeur. |
| <code>requests.memory</code> | Pour tous les pods dans un état non terminal, la somme des requêtes en mémoire ne peut pas dépasser cette valeur. |
| <code>Pods</code> | Nombre total de pods dans un état non terminal. |
| <code>services</code> | Nombre total de services pouvant exister. |

Création d'un ResourceQuota

Pour cet exemple, nous allons mettre en place les exigences suivantes :

- Le nombre total de pods ne doit pas dépasser 2 pods.
- Le total de demandes de mémoire pour tous les conteneurs ne doit pas dépasser 1 Go.
- Le total de la limite de mémoire pour tous les conteneurs ne doit pas dépasser 2 Go.

- Le nombre total de demandes d'UC (niveau d'utilisation du processeur central) pour tous les conteneurs ne doit pas dépasser 1 UC.
- Le nombre total de processeurs pour tous les conteneurs ne doit pas dépasser 2 cpu.

Créons notre ResourceQuota avec les contraintes définies plus haut, via le template YAML suivant :

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: mem-cpu-quota
spec:
  hard:
    pods: "2"
    requests.cpu: "1"
    limits.cpu: "2"
    requests.memory: 1Gi
    limits.memory: 2Gi
```

Ensuite, nous allons **définir notre ResourceQuota dans notre namespace `dev-team`**, de la façon suivante :

```
kubectl apply -f quota.yaml -n=dev-team
```

Comme d'habitude, nous vérifions si notre ResourceQuota est disponible dans notre namespace `dev-team` :

```
kubectl get resourcequota -n dev-team
```

Résultat :

```
NAME          CREATED AT
mem-cpu-quota 2019-09-01T17:48:57Z
```

Dès à présent, nous allons créer un Pod qui demande une consommation de 700 Mio de mémoire, dans le namespace `dev-team` :

```
apiVersion: v1
kind: Pod
metadata:
  namespace: dev-team
  name: nginx1
spec:
  containers:
  - name: nginx1
    image: nginx
    resources:
      limits:
        memory: "1Gi"
        cpu: "800m"
      requests:
        memory: "700Mi"
        cpu: "400m"
```

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

Ensuite, dans le même namespace, nous créerons un second Pod avec la même consommation, soit un pod qui dépasse le quota de la mémoire disponible (700 Mio + 700 Mio > 1 Gio)

```
apiVersion: v1
kind: Pod
metadata:
  namespace: dev-team
  name: nginx2
spec:
  containers:
  - name: nginx2
    image: nginx
    resources:
      limits:
        memory: "1Gi"
        cpu: "800m"
      requests:
        memory: "700Mi"
        cpu: "400m"
```

Si, nous tentons de lancer la commande de création de ce nouveau pod, au sein de notre namespace **dev-team**, alors nous nous retrouverons avec l'erreur suivante :

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

Résultat :

```
Error from server (Forbidden): error when creating "pod.yaml": pods "nginx2" is forbidden
```

D'après le résultat, on s'aperçoit bien que nous dépassons la limite du ResourceQuota `mem-cpu-quota` défini dans le namespace `dev-team`.

Tous les objets ne sont pas dans un namespace

La plupart des objets Kubernetes se trouvent dans certains namespace. Cependant, les objets de bas niveau, tel que les nodes ou les persistentVolumes, ne figurent dans aucun namespace.

Voici la commande qui permet de **lister les objets Kubernetes qui sont et ne sont pas dans un namespace** :

```
kubectl api-resources --namespaced=true
```

Résultat :

```
NAME                SHORTNAMES  APIGROUP                NAMESPACE  KIND
...
pods                 po          core                    true        Pod
...
deployments          deploy      extensions              true        Deployment
roles                rbac.authorization.k8s.io  true                Role
```

```
kubectl api-resources --namespaced=false
```

Résultat :

```
NAME                SHORTNAMES  APIGROUP                NAMESPACE
componentstatuses  cs          core                    false
namespaces         ns          core                    false
...
```

| | | | |
|-------------------|----|----------------|-------|
| storageclasses | sc | storage.k8s.io | false |
| volumeattachments | | storage.k8s.io | false |

Conclusion

Pour conclure, les namespaces restent un très bon moyen pour gérer les différentes équipes qui utilisent votre cluster. En plus de l'objet ResourceQuota combiné avec les namespaces qui vous permettront de limiter davantage l'utilisation des ressources de vos objets Kubernetes.

Comme à mon habitude voici un aide-mémoire des commandes liées aux namespaces :

```
# Afficher la liste des namespaces
kubectl get namespaces

# Créer un namespace
kubectl create namespace <NAMESPACE NAME>
kubectl create -f <template.yaml>

# Créer un deployment dans un namespace
kubectl run nginx --image=nginx --namespace=<NAMESPACE NAME>

# Supprimer un namespace
kubectl delete namespaces <NAMESPACE NAME>

# Afficher les détails d'un namespace
kubectl describe namespaces <NAMESPACE NAME>

# Utiliser un namespace par défaut
kubectl config set-context --current --namespace=<NAMESPACE NAME>

# Examiner le namespace par défaut
kubectl config view | grep namespace

# Lister les objets Kubernetes qui sont et ne sont pas dans un namespace
kubectl api-resources --namespaced=true
kubectl api-resources --namespaced=false
```