FACILITER LE DÉPLOIEMENTS K8S AVEC KUSTOMIZE

Introduction

Prérequis

Avant de poursuivre ce cours, vous devez au minimum avoir une compréhension de base sur la technologie Kubernetes, si vous n'avez jamais touché à Kubernetes de votre vie, alors je vous conseille grandement dire lire mon cours complet sur Kubernetes.

Pourquoi Kustomize ?

Si vous avez déjà tenté de déployer vos applications sur différents environnements Kubernetes, il est fort probable que vous ayez personnalisé une configuration Kubernetes pour chaque environnement en copiant les fichiers YAML de vos ressources k8s et en les modifiant en fonction de vos besoins pour chaque environnement. Cette approche reste très fastidieuse et répétitive car pour intégrer des améliorations vous devez passer par chaque environnement

Par exemple, imaginez que vous ayez besoin de déployer vos applications k8s dans un environnement de dev, pre-prod et prod. Vous voulez donc les utiliser ensemble, d'une manière ou d'une autre. De plus, vous souhaitez personnaliser les fichiers afin que vos instances de ressources apparaissent dans le cluster avec des Labels qui les distingue des ressources d'un collègue qui font la même chose dans le même cluster. Vous souhaitez également définir des valeurs appropriées pour le CPU, la

mémoire et le nombre de répliques pour chaque environnement.

De plus, vous aurez besoin de plusieurs variantes de l'ensemble de la configuration : une petite variante (en matière de ressources informatiques utilisées) consacrée aux tests et à l'expérimentation, et une variante beaucoup plus grande consacrée au service des utilisateurs externes en production. De même, d'autres équipes voudront leurs propres variantes.

Cela soulève toutes sortes de questions. Copiez-vous votre configuration sur plusieurs emplacements et les modifiez-vous indépendamment ? Et si vous avez des dizaines d'équipes de développement qui ont besoin de variantes légèrement différentes de la pile ? Comment maintenez-vous et mettez-vous à niveau les aspects de configuration qu'ils partagent en commun ? Les workflows utilisant kustomize apportent des réponses à ces questions.

kustomize est un outil Kubernetes qui vous permet de personnaliser les fichiers YAML bruts de vos ressources k8s d'origine à des fins multiples (ex: différents environnements, différentes variables/répliques/ressources informatique, etc ...), en laissant les fichiers YAML d'origines intacts et utilisables tel quel.

Installation

kustomize n'est pas un nouvel outil, il est en construction depuis 2017 et a été introduit en tant que sous-commande native de kubectl dans la version 1.14. Vous n'êtes pas donc pas obligé de télécharger le binaire kustomize car il est déjà intégré en tant que sous-commande `kubectl`. Mais si jamais vous souhaitez l'installer séparément de kubectl, alors voici la procédure à suivre ci-dessous.

Il existe plusieurs façons d'installer l'outil kustomize que vous retrouverez sur la page d'installation kustomize. Dans notre cas nous allons installer la dernière version depuis la source afin de profiter de la toute dernière version de l'outil. Pour ce faire, vous devrez d'abord mettre à niveau votre système et vos packages:

```
sudo apt update -y && sudo apt upgrade -y
```

Installez ensuite le package curl s'il n'est pas déjà installé:

```
sudo apt install -y curl
```

Nous sommes maintenant prêts à télécharger le binaire kustomize pour Linux depuis les sources officielles avec la commande suivante :

```
curl -s "https://raw.githubusercontent.com/kubernetes-sigs/kustomize/master/hack/inst
```

Après avoir téléchargé le binaire, nous allons le déplacer le dossier **/usr/local/bin/** afin que n'importe quel utilisateur normal puisse exécuter la commande `kustomize` :

```
sudo mv kustomize /usr/local/bin
```

Enfin la dernière étape c'est de tester l'installation:

```
kustomize --help
```

Résultat :

```
Manages declarative configuration of Kubernetes.
See https://sigs.k8s.io/kustomize

Usage:
    kustomize [command]

Available Commands:
```

```
    build    Build a kustomization target from a directory or URL.
...
```

> **Information**
>
> Au moment de la rédaction de cet article, la version actuelle de l'outil était la v4.2.0

Utilisation avec et sans kustomize

Afin de mieux **comprendre l'intérêt de kustomize**, nous allons travailler sur un projet n'utilisant pas cet outil et inversement.

Sans kustomize

Pour commencer, nous allons créer et nous placer sur un dossier d'espace de travail sans utiliser pour le moment l'outil kustomize, soit :

```
mkdir test && cd test
```

Dedans nous allons créer les fichier YAML de nos ressources k8s dans un dossier qu'on nommera `base` :

```
mkdir base
```

Créons ensuite un fichier `base/deployment.yaml` pour créer notre Deployment nginx dans le dossier créé précédemment, comme suit :

```
vim base/deployment.yaml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: http-test-kustomize
spec:
  selector:
    matchLabels:
      app: http-test-kustomize
  template:
    metadata:
      labels:
        app: http-test-kustomize
    spec:
      containers:
      - name: http-test-kustomize
        image: nginx
        ports:
        - name: http
          containerPort: 8080
          protocol: TCP
```

On fait la même chose pour le fichier YAML de notre Service k8s, avec un fichier qu'on nommera cette-fois-ci **base/service.yaml** :

```
vim base/service.yaml
```

```
apiVersion: v1
kind: Service
metadata:
  name: http-test-kustomize
spec:
  ports:
    - name: http
      port: 8080
  selector:
    app: http-test-kustomize
```

À cet instant, l'arborescence du projet ressemble à ceci :

```
|__ base
    |__ deployment.yaml
    |__ service.yaml
```

Ensuite, placez-vous à la racine de votre projet et créez vos ressources k8s avec la commande suivante:

```
kubectl apply -f base/
```

Résultat :

```
deployment.apps/http-test-kustomize created
service/http-test-kustomize created
```

Vérifions ensuite l'état de nos ressources :

```
kubectl get deploy,service
```

Résultat :

```
NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/http-test-kustomize     1/1     1            1           5s

NAME                            TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S
service/http-test-kustomize     ClusterIP      10.200.112.174   < none>        8080/
```

Pour le moment rien de nouveau, nous avons un Deployment avec 1 Pod qui utilise un Service qui écoute sur le port 8080. Voyons voir maintenant comment nous faciliter la vie avec kustomize. Avant cela, supprimons d'abord ce que nous venons de créer :

```
kubectl delete -f base/
```

Avec kustomize

Pour voir l'avantage de kustomize, nous allons supprimer les Labels et Selectors des 2 fichiers créés précédemment et nous laisserons kustomize les gérer pour nous.

Modifions donc ces 2 fichiers avec le nouveau contenu suivant:

```
vim base/deployment.yaml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: http-test-kustomize
spec:
  template:
    spec:
      containers:
      - name: http-test-kustomize
        image: nginx
        ports:
        - name: http
          containerPort: 8080
          protocol: TCP
```

Même chose pour le fichier configuration de notre Service k8s:

```
vim base/service.yaml
```

```
apiVersion: v1
kind: Service
metadata:
  name: http-test-kustomize
spec:
  ports:
    - name: http
      port: 8080
```

Pour information, ces fichiers ne seront JAMAIS modifiés, nous appliquerons simplement une personnalisation au-dessus d'eux grâce à l'outil kustomize pour créer de nouvelles définitions de nos ressources k8s.

Dans cet exemple, nous allons donc générer nos Labels et Selectors automatiquement depuis kustomize. Pour cela nous allons ajouter un nouveau fichier dans le dossier **base** nommé **base/kustomization.yaml**. Ce fichier sera le point central de notre base car c'est ici que nous déclarons les changements des ressources concernées:

```
vim base/kustomization.yaml
```

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

commonLabels:
  app: http-test-kustomize

resources:
  - service.yaml
  - deployment.yaml
```

Dans le paramètre `ressources` nous déclarons les ressources k8s à modifier et dans le paramètre `commonLabels` nous déclarons les labels à ajouter à toutes les ressources et sélecteurs.

Information

Rendez-vous sur la page les paramètres kustomize pour voir les différentes paramètres qu'il est possible d'intégrer dans le fichier **kustomization.yaml**.

À cet instant, l'arborescence du projet ressemble à ceci :

```
|__ base
    |__ deployment.yaml
    |__ service.yaml
    |__ kustomization.yaml
```

Il est possible de **visualiser le fichier YAML final qui sera généré par kustomize** depuis la commande suivante:

```
kustomize build base
```

ou bien la commande suivante si vous n'avez pas téléchargé le binaire:

```
kubectl kustomize k8s
```

Résultat :

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: http-test-kustomize
  name: http-test-kustomize
spec:
  ports:
  - name: http
    port: 8080
  selector:
    app: http-test-kustomize
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: http-test-kustomize
  name: http-test-kustomize
spec:
  selector:
    matchLabels:
      app: http-test-kustomize
  template:
    metadata:
      labels:
        app: http-test-kustomize
    spec:
      containers:
      - image: nginx
        name: app
        ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

Vous remarquerez qu'il a généré automatiquement pour nous les Labels et Selectors, quelle classe ce kustomize !!

Pour **appliquer et générer les ressources k8s depuis kustomize**, vous lancerez la commande `kubectl apply` avec l'option `-k` comme suite :

```
kubectl apply -k base
```

Résultat :

```
service/http-test-kustomize created
deployment.apps/http-test-kustomize created
```

Pour **supprimer les ressources k8s depuis kustomize**, vous lancerez la commande `kubectl delete` avec l'option `-k` comme suite :

```
kubectl apply -k base
```

Résultat :

```
service "http-test-kustomize" deleted
deployment.apps "http-test-kustomize" deleted
```

Gestion des environnements

Nommage et Namespaces

Maintenant, nous voulons personnaliser notre application pour un cas spécifique, par exemple, pour souhaitons **gérer l'environnement de développement et de production séparément** avec l'outil kustomize. À chaque étape, nous verrons comment enrichir notre base avec quelques modifications.

L'objectif principal de cet article n'est pas de couvrir l'ensemble des fonctionnalités de kustomize mais d'être un exemple standard pour vous montrer la philosophie derrière cet outil.

Pour cela, tout d'abord, nous allons créer nos deux namespaces de dev et de prod:

```
kubectl create ns dev
kubectl create ns prod
```

Ensuite, nous allons créer le dossier **dev** et **prod** à la racine du projet un avec le fichier **kustomization.yaml** dedans. Ainsi, la structure de notre projet ressemblera à ceci:

```
|__ base
|   |__ deployment.yaml
|   |__ service.yaml
|   |__ kustomization.yaml
|__ dev
|   |__ kustomization.yaml
|__ prod
    |__ kustomization.yaml
```

Commençons avec le fichier **kustomization.yaml** de l'environnement de dev en y rajoutant le contenu suivant:

```
vim dev/kustomization.yaml
```

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
bases:
- ../base/
namespace: dev
namePrefix: dev-
```

Dans le paramètre `bases` nous indiquons l'emplacement de nos fichiers sources YAML, dans le paramètre `namespace` nous indiquons le namespace à utiliser par défaut et dans le paramètre `namePrefix` nous indiquons la valeur du préfixe à ajouter aux noms de toutes les ressources.

Vérifions maintenant la version du fichier YAML final avec ces nouveaux paramètres:

```
kubectl kustomize dev
```

Résultat :

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: http-test-kustomize
  name: dev-http-test-kustomize
  namespace: dev
spec:
  ports:
  - name: http
    port: 8080
  selector:
    app: http-test-kustomize
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: http-test-kustomize
  name: dev-http-test-kustomize
  namespace: dev
spec:
  selector:
    matchLabels:
      app: http-test-kustomize
  template:
    metadata:
      labels:
        app: http-test-kustomize
    spec:
      containers:
      - image: nginx
        name: app
        ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

Vous pouvez ensuite faire la même chose pour l'environnement de production:

```
vim prod/kustomization.yaml
```

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
bases:
- ../base/
namespace: prod
namePrefix: prod-
```

Variables d'environnement

Variables d'environnement statiques

Dans notre base, nous n'avons défini aucune variable d'environnement. Nous allons maintenant ajouter cela au-dessus de notre base en utilisant la ressource k8s de type Configmap.

La première étape consiste à modifier le fichier **base/deployment.yaml** pour prendre en considération notre Configmap. Voici à va ressembler notre nouveau fichier YAML de notre Deployment:

```
vim base/deployment.yaml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: http-test-kustomize
spec:
  template:
    spec:
      containers:
      - name: http-test-kustomize
        image: nginx
        ports:
        - name: http
          containerPort: 8080
          protocol: TCP
        envFrom:
        - configMapRef:
            name: http-test-kustomize
```

Pour cette exemple, nous allons ajouter et manipuler des variables qu'on nommera `ENV` et `TEST` dans notre Configmap qui sera entièrement gérée par kustomize.

Pour cela, il faut d'abord modifier le fichier **dev/kustomization.yaml** en lui ajoutant l'option `configMapGenerator` qui permet de générer une ConfigMap à partir d'un fichier ou une liste de clé-valeur:

```
vim dev/kustomization.yaml
```

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
bases:
- ../base/
namespace: dev
namePrefix: dev-
configMapGenerator:
  - name: http-test-kustomize
    env: config.properties
```

Nous allons ensuite créer le fichier **dev/config.properties** qui sera un simple fichier de propriétés clé-valeur utilisé par kustomize pour générer notre ConfigMap:

```
vim dev/config.properties
```

```
ENV=dev
TEST=true
```

À ce stade, la structure du projet ressemble actuellement à ceci:

```
|__ base
|   |__ deployment.yaml
|   |__ service.yaml
|   |__ kustomization.yaml
|__ dev
|   |__ kustomization.yaml
|   |__ config.properties
|__ prod
    |__ kustomization.yaml
```

Vérifions maintenant la version du fichier YAML final généré par kustomize:

```
kubectl kustomize dev
```

Résultat :

```
apiVersion: v1
data:
  ENV: dev
  TEST: "true"
kind: ConfigMap
metadata:
```

```
  name: dev-http-test-kustomize-tkt445c8kg
  namespace: dev
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: http-test-kustomize
  name: dev-http-test-kustomize
  namespace: dev
spec:
  ports:
  - name: http
    port: 8080
  selector:
    app: http-test-kustomize
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: http-test-kustomize
  name: dev-http-test-kustomize
  namespace: dev
spec:
  selector:
    matchLabels:
      app: http-test-kustomize
  template:
    metadata:
      labels:
        app: http-test-kustomize
    spec:
      containers:
      - envFrom:
        - configMapRef:
            name: dev-http-test-kustomize-tkt445c8kg
        image: nginx
        name: http-test-kustomize
        ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

Enfin, nous allons appliquer les mêmes modifications pour notre environnement de production:

```
vim prod/kustomization.yaml
```

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
bases:
- ../base/
namespace: prod
namePrefix: prod-
configMapGenerator:
  - name: http-test-kustomize
    env: config.properties
```

```
vim prod/config.properties
```

```
ENV=prod
TEST=false
```

À ce moment, la structure du projet ressemble actuellement à ceci:

```
|__ base
|   |__ deployment.yaml
|   |__ service.yaml
|   |__ kustomization.yaml
|__ dev
|   |__ kustomization.yaml
|   |__ config.properties
|__ prod
    |__ kustomization.yaml
    |__ config.properties
```

Variables d'environnement dynamiques

Comme nous pouvons le voir sur l'exemple précédent, les clés et les valeurs de notre ConfigMap sont assez statiques et ne peuvent pas être remplacées facilement. Pour se faciliter la tache, nous allons **utiliser les variables d'environnement système avec kustomize**.

Dans cet exemple, nous allons passer la variable `TEST` en tant que variable d'environnement. Pour cela nous allons d'abord déclarer la clé `TEST` sans valeur dans le fichier **dev/config.properties**, comme cela:

```
vim dev/config.properties
```

```
ENV=dev
TEST
```

Ensuite, nous allons exporter la variable d'environnement avant de lancer la commande kustomize. Exemple:

```
TEST="false" kubectl kustomize dev
```

Résultat :

```
apiVersion: v1
data:
  ENV: dev
  TEST: "false"
kind: ConfigMap
metadata:
  name: dev-http-test-kustomize-4t5bm74mm6
...
```

Nombre de Replicas

Comme dans notre exemple précédent, nous allons étendre notre base pour définir des paramètres non déjà définis. Ici, nous aimerions ajouter des informations sur le nombre de répliques. Pour cela c'est simple, il suffit juste de modifier notre fichier **dev/kustomization.yaml** avec le contenu suivant:

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
bases:
- ../base/
namespace: dev
namePrefix: dev-
configMapGenerator:
  - name: http-test-kustomize
    env: config.properties

replicas:
```

```
- name: http-test-kustomize
  count: 2
```

Vérifions maintenant notre fichier YAML:

```
kubectl kustomize dev
```

Résultat :

```
...
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: http-test-kustomize
  name: dev-http-test-kustomize
  namespace: dev
spec:
  replicas: 2
```

Nous allons cette fois-ci en profiter de la ressource k8s de type "HorizontalPodAutoscaler" pour la mise échelle automatique du nombre de pods de notre Deployment en fonction de l'utilisation du processeur. Pour cela nous allons commencer par créer notre HorizontalPodAutoscaler dans notre base :

```
vim base/hpa.yaml
```

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: http-test-kustomize
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: http-test-kustomize
  minReplicas: 1
  maxReplicas: 2
  metrics:
  - type: Resource
    resource:
      name: cpu
```

```
      targetAverageUtilization: 60
```

Ici nous aurons entre 1 à 2 répliques de nos pods selon la consommation du CPU (60%).

Ensuite n'oublions pas de rajouter cette ressource dans notre fichier **base/kustomization.yaml** situé dans notre base :

```
vim base/kustomization.yaml
```

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
commonLabels:
  app: http-test-kustomize

resources:
  - service.yaml
  - deployment.yaml
  - hpa.yaml
```

Enfin, il suffit maintenant de modifier le fichier **dev/kustomization.yaml** en lui rajoutant le paramètre `patchesStrategicMerge` qui permet la prise en charge du mécanisme de correctif. Dans notre cas nous allons modifier le nombre de répliques pour l'environnement de dev:

```
vim dev/kustomization.yaml
```

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
bases:
- ../base/
namespace: dev
namePrefix: dev-
configMapGenerator:
  - name: http-test-kustomize
    env: config.properties
patchesStrategicMerge:
- hpa-dev.yaml
```

Enfin nous spécifions les nouvelles valeurs des paramètres `minReplicas` et `maxReplicas` de notre HorizontalPodAutoscaler, dans le fichier **dev/hpa-dev.yaml**, comme suit:

```
vim dev/hpa-dev.yaml
```

```
---
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: http-test-kustomize
spec:
  minReplicas: 2
  maxReplicas: 3
```

Vérifions cela sur notre fichier YAML final:

```
kubectl kustomize dev
```

Résultat :

```
...
---
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  labels:
    app: http-test-kustomize
  name: dev-http-test-kustomize
  namespace: dev
spec:
  maxReplicas: 3
  metrics:
  - resource:
      name: cpu
      targetAverageUtilization: 60
    type: Resource
  minReplicas: 2
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: dev-http-test-kustomize
```

Et on fait la même chose pour l'environnement de production:

```
vim prod/kustomization.yaml
```

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
bases:
- ../base/
namespace: prod
namePrefix: prod-
configMapGenerator:
  - name: http-test-kustomize
    env: config.properties
patchesStrategicMerge:
- hpa-prod.yaml
```

```
vim prod/hpa-prod.yaml
```

```
---
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: http-test-kustomize
spec:
  minReplicas: 3
  maxReplicas: 5
```

À ce stade, la structure du projet ressemble actuellement à ceci:

```
|__ base
|   |__ deployment.yaml
|   |__ service.yaml
|   |__ kustomization.yaml
|__ dev
|   |__ kustomization.yaml
|   |__ hpa-dev.yaml
|   |__ config.properties
|__ prod
    |__ kustomization.yaml
    |__ hpa-dev.yaml
    |__ config.properties
```

Conclusion

En conclusion, kustomize est un ajout utile à votre boîte à outils Kubernetes. Cela peut résoudre certains des problèmes de modélisation les plus fastidieux auxquels vous pouvez être confronté. Offrant ainsi quelques précieux avantages qui sont:

- **Natif à Kubectl**: pas besoin d'installer ou de gérer en tant que dépendance distincte.
- **Langage Yaml**: il n'a pas de langage de template, vous pouvez utiliser le YAML standard pour déclarer rapidement vos configurations.
- **Configurations multiples**: gère un nombre illimité de configurations différentes.
- **Réutilisabilité**: permet de réutiliser un fichier de base dans tous vos environnements (développement, production), puis de superposer des spécifications uniques pour chacun.