Les variables d'environnement dans Kubernetes

Dans ce chapitre, nous verrons les différentes façons d'injecter des variables d'environnement dans les applications Kubernetes

1) Introduction

Lorsque vous construisez votre pile d'application pour fonctionner sur Kubernetes, généralement la configuration de base du pod est généralement effectuée en définissant différentes variables d'environnement. Si par exemple si vous souhaitez créer un pod utilisant l'image wordpress, vous aurez alors besoin de surcharger certaines variables d'environnement, tels que l'adresse, l'utilisateur et le mot de passe de la base de données. Par ailleurs vous souhaiteriez peut-être aussi protéger le mot de passe de la base de données en évitant de l'afficher en clair. De plus, parfois, vous voudriez aussi définir tout un ensemble de variables d'environnement pouvant être partagées par plusieurs conteneurs de votre pods.

Afin de répondre à tous ces besoins, nous verrons donc, dans cet article, **les différentes façons d'injecter des variables d'environnement dans les applications Kubernetes**.

2) Les types de variables d'environnement

2.1) Le mode basique

Lorsque vous créez un pod, vous pouvez définir des variables d'environnement pour les conteneurs qui s'exécutent dans le pod en incluant le champ `env` dans le fichier de configuration du pod.

Dans cet exercice, nous créerons un pod qui exécute un conteneur basé sur l'image alpine contenant une variable d'environnement nommé `DEMO_ENV` et sa valeur "Coucou". Voici donc à quoi va ressembler le fichier de configuration du pod :

```
apiVersion: v1
kind: Pod
metadata:
  name: alpine
spec:
  containers:
  - name: alpine
    image: alpine
    env:
    - name: DEMO_ENV
      value: "Coucou"
    command: ["/bin/sh", "-c"]
    args:
      - while true; do
          sleep 1;
        done
```

Copier

Créez ensuite le pod basé sur le fichier de configuration YAML:

```
kubectl apply -f pod.yaml
```

Copier

Par la suite, nous allons exécuter la commande `printenv` afin de **répertorier les variables d'environnement disponible dans le conteneur de notre pod** :

```
kubectl exec -it alpine -- printenv | grep DEMO_ENV
```

Copier

Résultat :

```
DEMO_ENV=Coucou
```

2.2) ConfigMap

Dans le cas où vous utilisez les mêmes variables d'environnement dans plusieurs pods, il est plus intéressant de songer à utiliser l'objet `ConfigMap` .

Comment ça marche ? Premièrement vous définissez vos clés et vos valeurs votre objet ConfigMap et deuxièmement vous les réutilisez dans le champ `env` de vos Pods. Dès lors, quand vous souhaiteriez modifier la valeur votre variable d'environnement, vous n'aurez plus qu'à modifier sa valeur que dans votre ConfigMap. À l'inverse du mode basique, où vous auriez rectifié la valeur de votre variable d'environnement pour chaque pod.

Pour créer une ConfigMap, soit vous utilisez soit la commande `kubectl` soit un fichier yaml.

2.2-1) Depuis la commande kubectl

Voici à quoi ressemble le prototype de la commande de création de ConfigMap avec l'outil kubectl :

```
kubectl create configmap <confiMap-name> --from-literal=<key>=<value>
```

Copier

Dans cet exemple nous allons créer une ConfigMap nommée `maconfigmap` avec comme clé `db-name` et comme valeur `test` , soit :

```
kubectl create configmap maconfigmap --from-literal=db-name=test
```

Copier

Voici la commande qui permet de **lister les ConfigMaps disponibles dans votre cluster** :

```
kubectl get configmaps
```
Copier

Résultat :

```
NAME          DATA   AGE
maconfigmap   1      2m26s
```

Si vous désirez **divulguer la valeur de votre configMap**, on exécutera la commande ci-dessous :

```
kubectl describe configmap maconfigmap
```
Copier

Résultat :

```
...
Data
====
db-name:
----
test
Events:  <none>
```

Si vous avez plusieurs clés et valeurs à déclarer, il serait plus intéressant de rajouter toutes les données dans un fichier. Pour ce faire, créez un fichier *values.txt* et complétez les données suivantes :

```
cle-un:valeur-un
cle-deux:valeur-deux
```

Pour utiliser notre fichier, nous utiliserons l'argument **--from-file** , comme suit :

```
kubectl create configmap maconfigmap2 --from-file=values.txt
```
Copier

Vérifions les données de notre `configmap2` :

```
kubectl describe configmaps maconfigmap2
```
Copier

Résultat :

```
...
Data
====
values.txt:
----
cle-un:valeur-un
cle-deux:valeur-deux

Events:  <none>
```

2.2-2) Utilisation d'un fichier YAML

L'autre manière de créer votre configMap, est d'utiliser un fichier YAML. Dans cet exemple, nous créerons une ConfigMap avec comme clé `ma-cle` et comme valeur `ma-valeur`.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: test-config
data:
  ma-cle: ma-valeur
```

Copier

Créez ensuite votre objet ConfigMap avec la commande suivante :

```
kubectl create -f configmapc.yaml
```

Copier

Enfin, vérifions si notre nouveau ConfigMap est bien disponible dans notre cluster :

```
kubectl get configmaps
```

Copier

Résultat :

```
NAME           DATA   AGE
maconfigmap    1      20m
maconfigmap2   1      13m
test-config    1      3m26s
```

2.2-3) Utiliser votre ConfigMap dans une variable d'environnement

Dans le fichier YAML de notre pod, nous aurons besoin du champ `valueFrom` afin de récolter une valeur depuis un objet Kubernetes et le champ `configMapKeyRef` pour indiquer à notre master qu'on souhaite récolter cette valeur depuis un objet de type ConfigMap, ce qui nous donne le fichier YAML suivant :

```
apiVersion: v1
kind: Pod
metadata:
  name: alpine
spec:
  containers:
  - name: alpine
    image: alpine
    env:
    - name: DB-NAME
      valueFrom:
        configMapKeyRef:
          name: maconfigmap
          key: db-name
    command: ["/bin/sh", "-c"]
    args:
      - while true; do
          sleep 1;
        done
```

Copier

Créez votre pod à l'aide de la commande suivante :

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

Copier

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

Affichons dorénavant, la valeur de notre variable d'environnement `DB-NAME` en exécutant la commande `printenv` :

```
kubectl exec -it alpine -- printenv | grep DB-NAME
```
Copier

Résultat :

```
DB-NAME=test
```

2.3) les Secrets

Les `Secret` sont des objets Kubernetes qui vous permettent de **stocker et de gérer des informations sensibles**, telles que des mots de passe, des jetons OAuth et des clés ssh.

Le fonctionnement est similaire à l'objet ConfigMap, vous déclarez vos données dans un objet de type Secret et par la suite vous les réutilisez dans votre Pod.

2.3-1) Depuis la commande kubectl

Comme pour l'objet ConfigMap, vous pouvez utiliser la commande `kubectl` de façon à **créer votre objet Secret**.

Voici le prototype de la commande `kubectl` :

```
kubectl create secret generic <secret-name> --from-literal=<key>=<value>
```
Copier

Dans cet exemple nous allons créer un secret nommé `monsecret` avec comme clé `db-password` et comme valeur `passw0rd`, soit :

```
kubectl create secret generic monsecret --from-literal=db-password=passw0rd
```
Copier

Voici la commande qui permet de **lister les secrets disponibles dans votre cluster** est la suivant :

```
kubectl get secrets
```
Copier

Résultat :

```
NAME                  TYPE                                  DATA   AGE
default-token-5hwrq   kubernetes.io/service-account-token   3      12m
monsecret             Opaque                                1      9s
```

Pour information, les données secrètes sont codées en **base64**, vous pouvez récupérer le hash depuis la commande suivante :

```
kubectl get secrets monsecret -o yaml
```
Copier

Résultat :

```
apiVersion: v1
data:
  db-password: cGFzc3cwcmQ=
```

```
...
```

Une fois le hash récupéré (ici `cGFzc3cwcmQ=`), **décodez votre secret** avec la commande suivante :

```
echo "cGFzc3cwcmQ=" | base64 -d
```
Copier

Résultat :

```
passw0rd
```

> **Information**
>
> Le codage base64 n'est pas une méthode de chiffrement et est considéré comme identique au texte brut.

Comme pour les ConfigMaps, vous pouvez utiliser un fichier pour rajouter toutes vos données secrètes. Dans cet exemple nous allons créer un fichier nommé *login.txt* contenant les données suivantes :

```
db-user:admin
db-password:p@ssw0rd
```

Pour créer notre objet secret depuis un fichier, nous utiliserons cette fois-ci l'argument **--from-file**, comme suit :

```
kubectl create secret generic loginsecret --from-file=login.txt
```
Copier

Récupérons la liste des secrets de notre cluster :

```
kubectl get secrets
```
Copier

Notre nouveau secret est bien présent :

```
NAME                  TYPE                                  DATA   AGE
default-token-5hwrq   kubernetes.io/service-account-token   3      53m
loginsecret           Opaque                                1      2m59s
monsecret             Opaque                                1      41m
```

2.3-2) Utilisation d'un fichier YAML

Dans cet exemple, nous créerons à l'aide d'un fichier yaml, un objet secret nommé `db-login` avec deux clés, soit la clé `db-user` avec la valeur `root` et la clé `db-password` avec la valeur `p@ssw0rd`.

Avant toute chose, nous devons d'abord **convertir nos valeurs en base64**, soit :

```
echo "root" | base64 && echo "p@ssw0rd" | base64
```
Copier

Résultat :

```
cm9vdAo=
cEBzc3cwcmQK
```

Une fois le hash obtenu, on peut commencer par la création de notre template YAML :

```
apiVersion: v1
kind: Secret
metadata:
  name: db-login
data:
  db-user: cm9vdAo=
  db-password: cEBzc3cwcmQK
```

Copier

Créez ensuite votre secret à l'aide la commande suivante :

```
kubectl create -f secret.yaml
```

Copier

Avant d'utiliser notre objet secret dans un pod, vérifions d'abord si il est bien disponible dans notre cluster :

```
kubectl get secrets
```

Copier

c'est bien le cas :

```
NAME                  TYPE                                  DATA   AGE
default-token-5hwrq   kubernetes.io/service-account-token   3      53m
loginsecret           Opaque                                1      2m59s
monsecret             Opaque                                1      45m
db-login              Opaque                                2      53s
```

2.3-3) Utiliser votre secret dans une variable d'environnement

```
apiVersion: v1
kind: Pod
metadata:
  name: alpine
spec:
  containers:
  - name: alpine
    image: alpine
    env:
    - name: DB-USER
      valueFrom:
        secretKeyRef:
          name: db-login
          key: db-user
    - name: DB-PASSWORD
      valueFrom:
        secretKeyRef:
          name: db-login
          key: db-password
    command: ["/bin/sh", "-c"]
    args:
      - while true; do
          sleep 1;
        done
```

Copier

Créez votre pod :

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

Dorénavant, affichons les valeurs de nos variables d'environnement `DB-USER` et `DB-PASSWORD` en exécutant la commande `printenv` :

```
kubectl exec -it alpine -- printenv | grep 'DB-USER\|DB-PASSWORD'
```
Copier

Résultat :

```
DB-USER=root
DB-PASSWORD=p@ssw0rd
```

3) Conclusion

Kubernetes nous offre selon nos besoins différentes façon de gérer et manipuler nos variables d'environnement. Pour vous exercez essayez de créer un Pod basé sur l'image wordpress.

Voici un **aide-mémoire des commandes liées aux variables d'environnement de Kubernetes** :

```
################## ConfigMap ################

# Créer d'une ConfigMap
kubectl create configmap <confiMap-name> --from-literal=<key>=<value>

# Créer d'un ConfigMap depuis un fichier
kubectl create configmap <confiMap-name> --from-file=<filaname>

# Lister les ConfigMaps
kubectl get configmaps

# Récupérer la valeur d'une ConfigMap
kubectl describe configmap <confiMap-name>


################## Secrets ################

# Créer d'un Secret
kubectl create secret generic <secret-name> --from-literal=<key>=<value>

# Créer d'un Secret depuis un fichier
kubectl create secret generic <secret-name> --from-file=<filaname>

# Lister les Secrets
kubectl get secrets

# Récupérer la valeur d'un Secret
kubectl get secrets <confiMap-name> -o yaml
echo "<hash>" | base64 -d

# Convertir une valeur en base64
echo "<value>" | base64
```
Copier

← Chapitre précédent Chapitre suivant →

Faire un don newsletter Convertir en pdf

Si vous aimez l'article, n'hésitez pas à mettre un like 2

k8s orchestrateur conteneur

12/09/2019

article rédigé par **ajdaini-hatim**

Partager sur :

← Lire d'autres articles

Des problèmes, des erreurs sur l'article ? Signalez-les !

Espace commentaire

Écrire un commentaires

vous devez être inscrit pour poster un message !

1 commentaire

14/09/19

@zammerrenu

bonjour

Autres articles

Création d'une fenêtre et d'un rendu sur la SDL 2

Dans cet article, nous allons aborder en détail la gestion et fonctionnement de la fenêtre et du rendu sur la SDL 2.

05/06/2019 par guerriernumerique

Déploiement et Backup automatique d'une application Symfony 4

Cet article vous explique comment déployer et sauvegarder votre application Symfony automatiquement, rapidement et avec (...)

16/05/2019 par ajdaini-hatim

Conclusion du cours Terraform

Clap de fin ! Vous connaissez dès à présent la plupart des concepts de base (...)

20/06/2020 par ajdaini-hatim

La supervision dans kubernetes

Dans ce chapitre nous allons parler du monitoring d'un cluster Kubernetes afin de surveiller la (...)

11/09/2019 par ajdaini-hatim

Gérer et manipuler les ReplicaSets Kubernetes

Dans ce chapitre, nous verrons comment gérer les répliques de vos Pods grâce aux ReplicaSets k8s.

14/08/2019 par ajdaini-hatim

Introduction du cours pour apprendre la technologie Docker

Ce cours complet pour débutants sur la technologie Docker vous expliquera pas à pas les différentes notions de Docker.

31/05/2019 par ajdaini-hatim

Utilisation de la stack ELK sur les logs Apache

Dans ce chapitre, nous allons apprendre à utiliser la stack ELK en analysant en temps réel les logs d'accès Apache.

13/07/2020 par ajdaini-hatim

Construire une infrastructure AWS hautement disponible

Dans ce chapitre, nous verrons comment construire une infrastructure AWS hautement disponible pour une application (...)

26/05/2020 par ajdaini-hatim