Formation Kubernetes

Divers

Configure le client CLI ‘kubectl’

```
$ kubectl config set-cluster default-cluster --server=http://masterkube:8080
$ kubectl config set-context default-context --cluster=default-cluster --
user=default-admin
$ kubectl config use-context default-context
```

Installation Dashboard:

```
$ kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/master/src/deploy/recommend
ed/kubernetes-dashboard.yaml
$ kubectl proxy  --port=8001 &
http://localhost:8001/api/v1/namespaces/kube-system/services/https:kubernetes-
dashboard:/proxy/.
```

Installation metrics-server pour Prometheus: Il doit être exécuté dans le cluster pour que les métriques et les graphiques soient disponibles. Vérifier que le service est en cours d'exécution:

```
$ kubectl get services --namespace=kube-system
```

Vérifier le cluster:

vérifier la version:

```
$ kubectl version
```

Vérifiez l'emplacement et les informations d'identification :

```
$ kubectl config view
```

Vérifiez chaque composant:

```
$ kubectl get cs
```

Obtenez les noms de vos nœuds du cluster

```
$ kubectl get nodes
$ kubectl describe nodes
```

exécute kubectl en mode reverse-proxy et tester le serveur API et l'authentification :

```
$ kubectl proxy --port=8080 &
$ Curl http://localhost:8080/
```

Obtenez une liste et l'URL du reverse Proxy de l'ensemble des services demarré sur le cluster:

```
kubectl cluster-info

kubectl cluster-info dump

kubectl get all

kubectl get services --namespace=kube-system

kubectl options

kubectl -h
```

Manifeste Template:

```
apiVersion: v1
kind: Pod
metadata:
  name: monpod
  namespace: namespace1
  annotations:
    description: my frontend
  labels:
    environment: "production"
    tier: "frontend"
spec:
  restartPolicy: Always
  securityContext:
    runAsUser: 1000
    fsGroup: 2000
  containers:
  - name: container1
```

```
      image: ubuntu
      env:
      -name: envname
       value: "valenv"
      ressources:
       limits:
        memory: "200Mi"
        cpu: "1"
        ephemeral-storage: "4Gi"
       requests:
        memory: "100Mi"
        cpu: "0.5"
        ephemeral-storage: "2Gi"
      securityContext:
       allowPrivilegeEscalation: false
      livenessProbe:
       initialDelaySeconds: 15
       timeoutSeconds: 1
       httpGet:
        path: /site
        port: 8080
        httpHeaders:
        - name: X-Custom-Header
          value: Awesome
      command: ['sh','-c','echo Hello onepoint! && sleep ]
      #args:
```

NAMESPACE:

1/Lister tous les namespaces:

```
$ kubectl get namespaces
```

2/ Créer un namespace:

```
$ kubectl create namespace namespace1

$ kubectl create –f namespace1.yaml
```

```
apiVersion: v1
kind: Namespace
metadata:
  name: namespace1
```

3/ obtenir des informations sur un namespace: $ kubectl describe namespaces kube-system

4/ Définir un namespace pour un objet:

```
$ kubectl --namespace=namespace1 run nginx –image=nginx
$ kubectl --namespace=namespace1 get <type>
```

5/ Définir le namespace par defaut pour toutes les commandes kubectl: $ kubectl config set-context $(kubectl config current-context) --namespace=namespace1 $ kubectl config view | grep namespace

6/ Supprimer un namespace : kubectl delete namespace namespace1

POD

1/Création d'un fichier manifest (YAML) initial :

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: monpod
spec:
  containers:
  - name: container1
    image: nginx
```

----------Configuration impérative:----------

2/Création du Pod à partir d'un fichier manifest:

```
$ kubectl create –f mon-fichier.yaml  (--namespace=namespace1)
        (--record enregistre la commande en cours dans les annotations. Utile pour
une révision ultérieure)
```

3/ Afficher les Pods en cours d'execution:

```
$ kubectl get pod monpod -o wide
        (-o wide permet d'afficher le Node auquel le pod a été assigné)
$ kubectl get pod monpod --namespace=namespace1 -o yaml
        (-o yaml "--output=yaml" spécifie d'afficher la configuration complette de
l'objet)
```

4/ Voir des informations détaillées sur l'histoire et le status du Pod:

```
$  kubectl describe pod monpod --namespace=namespace1
```

5/Supprimez votre Pod:

```
$ kubectl delete pod monpod –-namespace=namespace1
$ kubectl delete –grace-period=0 --force pod monpod --namespace=namespace1
```

6/ Recréer plusieurs Pods avec des nom et labels différents

7/ Assigné une classe Guaranteed, Burstable et BestEffort au Pod et afficher le résultat:

```
$ kubectl get pod |grep -i qosClass
```

8/ Attribuer des labels à un Pod:

```
$ kubectl label pods monpod environment=production tier=frontend
```

9/ Afficher les labels générées pour chaque pod:

```
$ kubectl get pods --show-labels –-namespace=namespace1
```

10/ Effectuer une recherche avec le selector equality-base:

```
$ kubectl get pods -l environment=production,tier=frontend
```

11/ Effectuer une recherche avec le selector set-based:

```
$ kubectl get pods -l 'environment in (production),tier in (frontend)‘
```

12/ Supprimer des Pods avec une recherche avec le selector equality-base:

```
$ kubectl delete pods -l environment=production,tier=frontend
```

13/ Attacher une anotation à un Pod:

```
$ kubectl annotate pods monpod description='my frontend'
```

LimitRange

1/ Créer un objet LimitRange

```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-limit-range
spec:
  limits:
  - default:
      cpu: 1
    defaultRequest:
      cpu: 0.5
    type: Container
```

2/ Créez le LimitRange dans l'espace de noms

```
$ kubectl create -f cpu-defaults.yaml --namespace=namespace1
```

/3 Créer un Pod dans le namesapce et vérifier les ressources. si un conteneur est créé dans l'espace de noms namespace1 et que le conteneur ne spécifie pas ses propres valeurs pour la demande CPU et la limite de l'UC, le conteneur reçoit une demande CPU par défaut de 0,5 et une limite par défaut de 1.

Mode privilége

Configurer un contexte de sécurité pour un pod avec un volume:

- runAsUser: spécifie que pour tout Conteneur du Pod, le premier processus s'exécute avec l'ID utilisateur 1000.
- fsGroup: spécifie que l'ID de groupe 2000 est associé à tous les Conteneurs du Pod.L'ID de groupe est également associé au volume monté.

```
      spec:
        securityContext:
          runAsUser: 1000
          fsGroup: 2000
          seLinuxOptions:
           level: "s0:c123,c456"
          supplementalGroups: [5678]
        containers:
         ...
          securityContext:
            privileged: true
            runAsUser: 2000
```

```
            seLinuxOptions:
             level: "s0:c123,c456"
            allowPrivilegeEscalation: false
            capabilities:
             add: ["NET_ADMIN", "SYS_TIME"]
```

-Obtenez un shell et dressez la liste des processus en cours pour vérifier que les processus s'exécutent en tant qu'utilisateur 1000

```
$  kubectl exec -it security-context-demo -- sh
$  ps aux
```

-Créer un fichier dans le répertoire de votre volume et vérifier la valeur de l'ID de groupe -affichez les capacités du processus 1:

```
$ cd /proc/1
```

---

sondes / probes

1-Définir une sonde d'activité qui utilise une requête EXEC:

```
      spec:
        containers:
         ...
         livenessProbe:
          exec:
           command:
           - cat
           - /tmp/healthy
          initialDelaySeconds: 5
          periodSeconds: 5
```

- periodSeconds: spécifie que kubelet doit effectuer une sonde d'activité toutes les 5 secondes.
- initialDelaySeconds: indique a kubelet qu'il doit attendre 5 secondes avant d'effectuer la première sonde.
- command: kubelet exécute la commande cat /tmp/healthy dans le conteneur Lorsque le conteneur démarre, il exécute cette commande:

```
$ /bin/sh -c "touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600"
```

2-Définir une sonde d'activité qui utilise une requête HTTPGET:

```
        spec:
          containers:
           ...
            livenessProbe:
              httpGet:
                path: /monsite
                port: 8080
                httpHeaders:
                - name: X-Custom-Header
                  value: Awesome
             initialDelaySeconds: 3
             periodSeconds: 3
```

3-Définir une sonde d'activité qui utilise une requête TCP:

```
        spec:
          containers:
            ...
            - containerPort: 8080
            readinessProbe:
              tcpSocket:
                port: 8080
              initialDelaySeconds: 5
              periodSeconds: 10
            livenessProbe:
              tcpSocket:
                port: 8080
              initialDelaySeconds: 15
              periodSeconds: 20
```

- Cet exemple utilise à la fois des sondes de disponibilité (Readiness) et de vivacité (Liveness).
- kubelet envoie la première sonde de disponibilité 5 secondes après le démarrage du conteneur.
- Cela tentera de se connecter au conteneur sur le port 8080. Si la sonde réussit, le pod sera marqué comme prêt.
- Le kubelet continuera à exécuter cette vérification toutes les 10 secondes.
- kubelet lancera la première sonde de vivacité 15 secondes après le début du conteneur.
- Tout comme la sonde de disponibilité, elle tentera de se connecter au conteneur goproxy sur le port 8080.
- Si la sonde d'activité échoue, le conteneur sera redémarré.

4-Utiliser un port nommé:

Vous pouvez utiliser un ContainerPort nommé pour les contrôles d'activité HTTP ou TCP:

```
        ports:
        - name: liveness-port
          containerPort: 8080
          hostPort: 8080

        livenessProbe:
          httpGet:
            path: /healthz
            port: liveness-port
```

Pod et volume

1/ Pod et Volume éphémère:

-Définir un volume dans le Pod et le monter dans le container avec une

```
limite:
        spec:
         volumes:
         -name: monvolume
           emptyDir(): {}
         containers:
         ...
          volumeMounts:
          -name: monvolume
           mountPath: /mnt/volume
          ressources:
           limits:
            memory: "200Mi"
            ephemeral-storage: 2GiB
           requests:
            ephemeral-storage: 1GiB
```

-Créer le Pod puis dans un autre terminal, lancer un shell sur le conteneur:

```
$ kubectl exec -it container1 -- /bin/bash
```

-Dans le terminal d'origine, surveillez les modifications apportées

2/ Pod et Volume persistant (PersistentVolumeClaim):

-Créer un "PersistentVolume"

```
$  mkdir /mnt/data
```

Dans le répertoire /mnt/data , créez un fichier index.html :

```
$ echo 'test volume persistant' > /mnt/data/index.html
```

-Créer un objet PersistentVolume à partir d'un nouveau manifest :

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: mypersvol
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

Afficher des informations (status) sur le PersistentVolume:

```
$ kubectl get pv task-pv-volume
```

-Crée un PersistentVolumeClaim, qui sera automatiquement lié au PersistentVolume et qui demande un volume d'au moins trois gibibytes pouvant fournir un accès en r/w pour au moins un node:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: mypersvolclaim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

-Regardez à nouveau le PersistentVolume et vérifier que la sortie montre que PersistentVolumeClaim est lié à votre PersistantVolume:

```
$  kubectl get pv task-pv-volume
```

3-Créer un pod qui utilise PersistentVolumeClaim comme volume de stockage.

```
...
        spec:
          volumes:
          - name: volume1
            persistentVolumeClaim:
            claimName: mypersvolclaim
          containers:
           ...
            volumeMounts:
            - mountPath: "/var/www/monsite"
              name: volume1
```

-Obtenez un shell sur le conteneur et vérifiez le montage:

```
$  kubectl exec -it task-pv-pod -- /bin/bash
$  kubectl get pod task-pv-pod
```

Les secrets

-Créer un secret Méthode 1: -Create files containing the username and password:

```
$ echo -n "admin" > ./username.txt
$ echo -n "azerty" > ./password.txt
```

-Package these files into secrets:

```
$ kubectl create secret generic user --from-file=./username.txt
$ kubectl create secret generic pass --from-file=./password.txt
```

-Créer un secret Méthode 2:

```
$ kubectl create secret generic monsecret --from-literal=username='my-app' --from-
literal=password='39528$vdg7Jb'
```

-Créer un secret Méthode 3:

-Créer l'object secret à partir du fichier yaml:

```
        apiVersion: v1
        kind: Secret
        metadata:
          name: monsecret
        data:
          username: admin
          password: azerty

-Afficher des informations sur le secret:
        $  kubectl get secret
        $  kubectl describe secret monsecret -o yaml


-Créer un pod qui accède aux secret via un volume (tous les fichiers créés sur
montage secret auront l'autorisation 0400):

```yaml
        ...
        spec:
          containers:
            ...
            volumeMounts:
            - name: secret-volume
              mountPath: /mnt/secret-volume
              readOnly: true
          volumes:
            - name: secret-volume
              secret:
               secretName: monsecret
               defaultMode: 256
```

-Spécifier un chemin particulier pour un item (/mnt/secret-volume/my-group/my-username à la place de /mnt/secret-volume/username) et spécifier des autorisations différentes pour différents fichiers (ici, la valeur d'autorisation de 0777):

```
        volumes:
          - name: foo
            secret:
              secretName: mysecret
              items:
              - key: username
                path: my-group/my-username
                mode: 511
```

- Si spec.volumes[].secret.items est utilisé, seules les clés spécifiées dans les items sont projetées.
- Pour consommer toutes les clés du secret, elles doivent toutes être répertoriées dans le champ des items.
```

- Toutes les clés listées doivent exister dans le secret correspondant. Sinon, le volume n'est pas créé.

-Créer un pod qui a accès aux secret via des variables d'environnement:

```
spec:
  containers:
   ...
   env:
   - name: ENVSECRET1
     valueFrom:
      secretKeyRef:
       name: usersecret
       key: username
   - name: ENVSECRET2
     valueFrom:
       secretKeyRef:
       name: passsecret
       key: password
```

Projected Volume

Créer un pod pour utiliser un "Projected Volume" et pour monter les secrets (ci-dessus) dans le même répertoire partagé:

```
        spec:
          containers:
          ...
            volumeMounts:
            - name: all-in-one
              mountPath: "/projected-volume"
              readOnly: true
          volumes:
          - name: all-in-one
            projected:
              sources:
              - secret:
                  name: user
              - secret:
                  name: pass
```

-Observez les modifications apportées au pod:

```
$  kubectl get --watch pod test-projected-volume
```

-Dans un autre terminal vérifiez que le répertoire contient vos sources projetées

```
$ kubectl exec -it test-projected-volume -- /bin/sh
$ ls /projected-volume/
```

Affecter un pod à un node particulier du cluster

-Répertoriez les nodes du cluster:

```
$ kubectl get nodes
```

-Choisissez le node et ajoutez-y une étiquette:

```
$ kubectl label nodes <your-node-name> label1=var1
```

-Vérifiez que le node que vous avez choisi possède le nouveau label :

```
$  kubectl get nodes --show-labels
```

-Créer un pod planifié sur le node choisi grace au sélecteur de node (node qui a un label label1=var1):

```
        spec:
          containers:
          ...
          nodeSelector:
            label1: var1
```

-Vérifiez que le pod est bien en cours d'exécution sur le node choisi:

```
$ kubectl get pods --output=wide
```

---

comptes de service

Configurer les comptes de service pour les pods: Un compte de service fournit une identité pour les processus qui 'exécutent dans un pod. Lorsque vous (un humain) accédez au cluster (par exemple, en utilisant ubectl), vous êtes authentifié par l'apiserver comme un compte utilisateur particulier (actuellement, il s'agit généralement d'un admin , sauf si votre administrateur de cluster a personnalisé votre cluster). Les rocessus dans des conteneurs à l'intérieur des Pods peuvent également contacter l'apiserver. Lorsqu'ils le font, ils sont authentifiés en tant que compte de service particulier (par exemple, default).

Utilisez le compte de service par défaut pour accéder au serveur API. Lorsque vous créez un pod, si vous ne spécifiez pas de compte de service, e compte de service default lui est automatiquement affecté dans le même espace de noms. Les autorisations API d'un compte de service dépendent du plug-in d'autorisation et de la stratégie utilisée.

vous pouvez désactiver les informations d'identification de l'API automounting pour un compte de service en définissant automountServiceAccountToken: false sur le compte de service:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-robot
automountServiceAccountToken: false
...
```

vous pouvez également désactiver les informations d'identification de l'API automounting pour un pod particulier:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  serviceAccountName: build-robot
  automountServiceAccountToken: false
  ...
```

La spécification de pod a priorité sur le compte de service si les deux spécifient une valeur automountServiceAccountToken .

Utilisez plusieurs comptes de service: Chaque espace de noms a une ressource de compte de service par default appelée default . lister toutes les ressources de serviceAccount dans l'espace de noms avec cette commande:

```
  $ kubectl get serviceAccounts
```

Vous pouvez créer des objets ServiceAccount supplémentaires comme suit:

```
$ cat > /tmp/serviceaccount.yaml <<EOF
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-robot
EOF
```

```
$ kubectl create -f /tmp/serviceaccount.yaml
serviceaccount "build-robot" created
```

Pour utiliser un compte de service autre que celui par défaut, définissez simplement le champ spec.serviceAccountName d'un pod sur le nom du compte de service que vous souhaitez utiliser. Le compte de service doit exister au moment de la création du module, sinon il sera rejeté. Vous ne pouvez pas mettre à jour le compte de service d'un pod déjà créé. Vous pouvez nettoyer le compte de service de cet exemple comme ceci: $ kubectl delete serviceaccount/build-robot

Créez manuellement un jeton d'API de compte de service: Supposons que nous ayons un compte de service existant nommé "build-robot" comme mentionné ci-dessus, et nous créons un nouveau secret manuellement.

```
$ cat > /tmp/build-robot-secret.yaml <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: build-robot-secret
  annotations:
    kubernetes.io/service-account.name: build-robot
type: kubernetes.io/service-account-token
EOF
$ kubectl create -f /tmp/build-robot-secret.yaml
secret "build-robot-secret" created
```

Vous pouvez maintenant confirmer que le nouveau secret généré contient un jeton d'API pour le compte de service "build-robot". Tous les jetons pour les comptes de service inexistants seront nettoyés par le contrôleur de jeton.

```
$ kubectl describe secrets/build-robot-secret
```

---

image

Pull une image d'un registre privé: créer un pod qui utilise un secret pour extraire une image d'un registre Docker privé ou d'un référentiel.

vous devez vous authentifier auprès d'un registre afin d'extraire une image privée:

```
$  docker login
```

Le processus de connexion crée ou met à jour un fichier config.json contenant un jeton d'autorisation.

Créer un secret dans le cluster qui contient votre jeton d'autorisation Créez ce secret, en le nommant regcred :

```
$ kubectl create secret docker-registry regcred --docker-server=<your-registry-
server> --docker-username=<your-name> --docker-password=<your-pword> --docker-
email=<your-email>
```

Vous avez correctement défini vos informations d'identification Docker dans le cluster sous la forme d'un secret appelé regcred .

consulter le format Secret au format YAML:

```
$ kubectl get secret regcred --output=yaml
```

La valeur du champ .dockerconfigjson est une représentation base64 de vos informations d'identification Docker.

Pour comprendre ce qui se trouve dans le champ .dockerconfigjson , convertissez les données secrètes dans un format lisible:

```
$ kubectl get secret regcred --output="jsonpath={.data.\.dockerconfigjson}" |
base64 -d
```

Notez que les données secrètes contiennent le jeton d'autorisation similaire à votre fichier local `~/.docker/config.json`.

Créer un pod qui utilise votre secret: Voici un fichier de configuration pour un pod qui doit avoir accès à vos informations d'identification Docker dans regcred :

```
apiVersion: v1
kind: Pod
metadata:
  name: private-reg
spec:
  containers:
  - name: private-reg-container
    image: <your-private-image>
  imagePullSecrets:
  - name: regcred
```

remplacez par le chemin d'accès à une image dans un registre privé tel que: janedoe/jdoe-private:v1 Le champ imagePullSecrets du fichier de configuration spécifie que Kubernetes doit obtenir les informations d'identification d'un secret nommé regcred .

Créez un pod qui utilise votre secret et vérifiez que le pod est en cours d'exécution: $ kubectl create -f my-private-reg-pod.yaml $ kubectl get pod private-reg

Init Container

Configurer l'initialisation du pod: comment utiliser un Init Container pour initialiser un Pod avant l'exécution d'un conteneur d'application. créez un pod avec un conteneur d'applications et un conteneur d'initialisation. Le conteneur init est exécuté avant le démarrage du conteneur d'application.

```
apiVersion: v1
kind: Pod
metadata:
  name: init-demo
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
    volumeMounts:
    - name: workdir
      mountPath: /usr/share/nginx/html
  # These containers are run during pod initialization
  initContainers:
  - name: install
    image: busybox
    command:
    - wget
    - "-O"
    - "/work-dir/index.html"
    - http://kubernetes.io
    volumeMounts:
    - name: workdir
      mountPath: "/work-dir"
  dnsPolicy: Default
  volumes:
  - name: workdir
    emptyDir: {}
```

vous pouvez voir que le pod a un volume que le conteneur init et le conteneur d'application partagent. Le conteneur init monte le volume partagé dans le `/usr/share/nginx/html /work-dir` , et le conteneur d'application monte le volume partagé dans `/usr/share/nginx/html`. Le conteneur init exécute la commande suivante, puis se termine: wget -O /work-dir/index.html http://kubernetes.io Notez que le conteneur init écrit le fichier index.html dans le répertoire racine du serveur nginx.

Handlers

Attacher des gestionnaires aux événements du cycle de vie des conteneurs: (Handlers) Kubernetes prend en charge les événements postStart et preStop. Kubernetes envoie l'événement postStart immédiatement après le démarrage d'un conteneur et envoie l'événement preStop immédiatement avant la fin du conteneur. créez un pod avec un conteneur. Le conteneur a des gestionnaires pour les événements postStart et preStop:

```
apiVersion: v1
kind: Pod
metadata:
  name: lifecycle-demo
spec:
  containers:
  - name: lifecycle-demo-container
    image: nginx
    lifecycle:
      postStart:
        exec:
          command: ["/bin/sh", "-c", "echo Hello from the postStart handler >
/usr/share/message"]
      preStop:
        exec:
          command: ["/usr/sbin/nginx","-s","quit"]
```

vous pouvez voir que la commande postStart écrit un fichier de message dans le `/usr/share` du conteneur. La commande preStop arrête nginx avec élégance. Ceci est utile si le conteneur est arrêté à cause d'un échec. Le gestionnaire postStart s'exécute de manière asynchrone par rapport au code du conteneur, mais la gestion par Kubernetes du conteneur se bloque jusqu'à la fin du gestionnaire postStart. Le statut du conteneur n'est pas défini sur RUNNING tant que le gestionnaire post-démarrage n'est pas terminé. Kubernetes envoie uniquement l'événement preStop lorsqu'un module est terminé . Cela signifie que le hook preStop n'est pas appelé lorsque le pod est terminé .

ConfigMap

Configurer un pod pour utiliser un fichier ConfigMap: ConfigMaps vous permet de découpler les artefacts de configuration du contenu de l'image pour garder les applications conteneurisées portables. comment créer des fichiers ConfigMaps et configurer des modules en utilisant des données stockées dans ConfigMaps. créer des configmaps à partir de répertoires , de fichiers ou de valeurs littérales :

```
$ kubectl create configmap <map-name> <data-source>
```

- map-name est le nom que vous souhaitez attribuer à ConfigMap
- data-source est le répertoire, le fichier ou la valeur littérale dans laquelle les données doivent être dessinées.

La source de données correspond à une paire clé-valeur dans ConfigMap, où key = le nom du fichier ou la clé que vous avez fournie sur la ligne de commande, et value = le contenu du fichier ou la valeur littérale que vous avez fournie sur la ligne de commande. Vous pouvez utiliser kubectl describe ou kubectl get pour kubectl get informations sur un ConfigMap.

Créer des ConfigMaps à partir de répertoires ou à partir de plusieurs fichiers du même répertoire.

```

```

```
$ kubectl create configmap game-config --from-
file=https://k8s.io/docs/tasks/configure-pod-container/configmap/kubectl
$ kubectl describe configmaps game-config
$ kubectl get configmaps game-config -o yaml
```

```
apiVersion: v1
data:
  game.properties: |
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:52:05Z
  name: game-config
  namespace: default
  resourceVersion: "516"
  selfLink: /api/v1/namespaces/default/configmaps/game-config
  uid: b4952dc3-d670-11e5-8cd0-68f728db1985
```

Créer des ConfigMaps à partir de fichiers: créer un fichier ConfigMap à partir d'un fichier individuel ou de plusieurs fichiers.

```
$ kubectl create configmap game-config-2 --from-
file=https://k8s.io/docs/tasks/configure-pod-
container/configmap/kubectl/game.properties
```

Vous pouvez passer l'argument --from-file plusieurs fois pour créer un fichier ConfigMap à partir de plusieurs sources de données.

Utilisez l'option --from-env-file pour créer un --from-env-file ConfigMap à partir d'un fichier env, par exemple:

```
$ kubectl create configmap game-config-env-file --from-env-
file=docs/tasks/configure-pod-container/game-env-file.properties
```

A VOIR !! https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/

Partage des espaces de noms de processus entre des conteneurs

Partage des espaces de noms de processus entre des conteneurs dans un pod: Lorsque le partage d'espace de noms de processus est activé, les processus d'un conteneur sont visibles par tous les autres conteneurs de ce Pod. Vous pouvez utiliser cette fonctionnalité pour configurer des conteneurs coopérants, tels qu'un conteneur sidecar de gestionnaire de journaux, ou pour résoudre les images de conteneur qui n'incluent pas les utilitaires de débogage comme un shell. préreqquis: Une porte de fonction alpha spéciale PodShareProcessNamespace doit être définie sur true sur le système: --feature-gates=PodShareProcessNamespace=true .

Le partage de l'espace de nom de processus est activé à l'aide du champ ShareProcessNamespace:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  shareProcessNamespace: true
  containers:
  - name: nginx
    image: nginx
  - name: shell
    image: busybox
    securityContext:
      capabilities:
        add:
        - SYS_PTRACE
    stdin: true
    tty: true
```

Créer le Pod et attacher un shell Vous pouvez signaler les processus dans d'autres conteneurs. Il est même possible d'accéder à une autre image conteneur en utilisant le lien `/proc/$pid/root` .

head /proc/8/root/etc/nginx/nginx.conf

Les processus sont visibles pour les autres conteneurs du module. Cela inclut toutes les informations visibles dans /proc Les systèmes de fichiers du conteneur sont visibles par les autres conteneurs du conteneur via le lien `/proc/$pid/root` .

Traduire un fichier Docker Compose

Traduire un fichier Docker Compose en ressources Kubernetes Kubernetes + Compose = Kompose C'est un outil de conversion pour tout ce qui compose (à savoir Docker Compose) pour les orchestrateurs de

conteneurs (Kubernetes ou OpenShift). http://kompose.io/

En trois étapes simples, nous vous emmènerons de Docker Compose à Kubernetes.

1. Prenez un exemple de fichier docker-compose.yaml
2. Exécutez "kompose up" dans le même répertoire
3. Et commencez sur Kubernetes

Alternativement, vous pouvez exécuter kompose convert et déployer avec kubectl: $ kompose --file docker-voting.yml convert $ kompose --provider openshift --file docker-voting.yml convert

Installation kompose: curl -L https://github.com/kubernetes/kompose/releases/download/v1.1.0/kompose-linux-amd64 -o kompose chmod +x kompose sudo mv ./kompose /usr/local/bin/kompose

A voir: https://kubernetes.io/docs/tools/kompose/user-guide/

Injecter des données dans une application

---

commande (Entrypoint) et des arguments (Cmd)

Définir une commande (Entrypoint) et des arguments (Cmd) pour un conteneur: Pour définir une commande, incluez le champ de command dans le fichier de configuration. Pour définir les arguments de la commande, incluez le champ args dans le fichier de configuration. La commande et les arguments que vous définissez ne peuvent pas être modifiés après la création du Pod. La commande et les arguments que vous définissez dans le fichier de configuration remplacent la commande par défaut et les arguments fournis par l'image Si vous définissez des arguments, mais ne définissez pas de commande, la commande par défaut est utilisée avec vos nouveaux arguments.

```
apiVersion: v1
kind: Pod
metadata:
  name: command-demo
  labels:
    purpose: demonstrate-command
spec:
  containers:
  - name: command-demo-container
    image: debian
    command: ["printenv"]
    args: ["HOSTNAME", "KUBERNETES_PORT"]
  restartPolicy: OnFailure
```

Pour voir la sortie de la commande exécutée dans le conteneur, affichez les journaux du Pod:

```
$ kubectl logs command-demo
```

---

variables d'environnement

Utiliser les variables d'environnement pour définir les arguments: Au lieu de fournir directement des chaînes (arguments), vous pouvez définir des arguments en utilisant des variables d'environnement:

```
env:
- name: MESSAGE
  value: "hello world"
command: ["/bin/echo"]
args: ["$(MESSAGE)"]
```

Cela signifie que vous pouvez définir un argument pour un Pod en utilisant l'une des techniques disponibles pour définir les variables d'environnement, y compris ConfigMaps et Secrets .

Exécuter une commande dans un shell:

```
args: ["-c", "while true; do echo hello; sleep 10;done"]
```

- Si vous ne fournissez pas de command ou d' args pour un conteneur, les valeurs par défaut définies dans l'image Docker sont utilisées.
- Si vous fournissez une command mais pas d' args pour un conteneur, seule la command fournie est utilisée. Le paramètre EntryPoint par défaut et le paramètre Cmd par défaut défini dans l'image Docker sont ignorés.
- Si vous ne fournissez que des args pour un conteneur, le paramètre Entrypoint par défaut défini dans l'image Docker est exécuté avec les args que vous avez fournis.
- Si vous fournissez une command et des args , l'Entrypoint par défaut et le Cmd par défaut définis dans l'image Docker sont ignorés. Votre command est exécutée avec vos args .

---

variable d'environnement pour un conteneur

Définir une variable d'environnement pour un conteneur: Pour définir les variables d'environnement, incluez le champ env ou envFrom dans le fichier de configuration.

```
apiVersion: v1
kind: Pod
metadata:
  name: envar-demo
  labels:
    purpose: demonstrate-envars
spec:
  containers:
  - name: envar-demo-container
    image: gcr.io/google-samples/node-hello:1.0
    env:
    - name: DEMO_GREETING
```

```
      value: "Hello from the environment"
    - name: DEMO_FAREWELL
      value: "Such a sweet sorrow"
```

Créer le Pod et connectez vous dessus . Dans votre shell, exécutez la commande "printenv" pour lister les variables d'environnement.

Les variables d'environnement définies à l'aide du champ env ou envFrom remplacent toutes les variables d'environnement spécifiées dans l'image du conteneur.

---

champs Pod comme valeurs pour les variables d'environnement

Utiliser les champs Pod comme valeurs pour les variables d'environnement: comment un Pod peut utiliser des variables d'environnement pour exposer des informations sur lui-même aux Conteneurs s'exécutant dans le Pod. Les variables d'environnement peuvent exposer les champs Pod et les champs Conteneur.

Downward API: Il existe deux façons d'exposer les champs Pod et Conteneur à un conteneur en cours d'exécution: -Variables d'environnement -DownwardAPIVolumeFiles ces deux façons d'exposer les champs Pod et Conteneur s'appellent l' API Downward .

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-envars-fieldref
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "sh", "-c"]
      args:
      - while true; do
          echo -en '\n';
          printenv MY_NODE_NAME MY_POD_NAME MY_POD_NAMESPACE;
          printenv MY_POD_IP MY_POD_SERVICE_ACCOUNT;
          sleep 10;
        done;
      env:
        - name: MY_NODE_NAME
          valueFrom:
            fieldRef:
              fieldPath: spec.nodeName
        - name: MY_POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: MY_POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        - name: MY_POD_IP
```

```
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
        - name: MY_POD_SERVICE_ACCOUNT
          valueFrom:
            fieldRef:
              fieldPath: spec.serviceAccountName
  restartPolicy: Never
```

vous pouvez voir cinq variables d'environnement. Le champ env est un tableau d' EnvVars . Le premier élément du tableau spécifie que la variable d'environnement MY_NODE_NAME tire sa valeur du champ spec.nodeName du Pod. De même, les autres variables d'environnement obtiennent leurs noms des champs Pod.

Créer le Pod et afficher les journaux du conteneur:

```
$  kubectl logs dapi-envars-fieldref
```

---

champs Conteneur comme valeurs pour les variables d'environnement

Utiliser les champs Conteneur comme valeurs pour les variables d'environnement:

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-envars-resourcefieldref
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox:1.24
      command: [ "sh", "-c"]
      args:
      - while true; do
          echo -en '\n';
          printenv MY_CPU_REQUEST MY_CPU_LIMIT;
          printenv MY_MEM_REQUEST MY_MEM_LIMIT;
          sleep 10;
        done;
      resources:
        requests:
          memory: "32Mi"
          cpu: "125m"
        limits:
          memory: "64Mi"
          cpu: "250m"
      env:
        - name: MY_CPU_REQUEST
          valueFrom:
```

```
            resourceFieldRef:
              containerName: test-container
              resource: requests.cpu
        - name: MY_CPU_LIMIT
          valueFrom:
            resourceFieldRef:
              containerName: test-container
              resource: limits.cpu
        - name: MY_MEM_REQUEST
          valueFrom:
            resourceFieldRef:
              containerName: test-container
              resource: requests.memory
        - name: MY_MEM_LIMIT
          valueFrom:
            resourceFieldRef:
              containerName: test-container
              resource: limits.memory
  restartPolicy: Never
```

vous pouvez voir quatre variables d'environnement. Le champ env est un tableau d' EnvVars . Le premier élément du tableau spécifie que la variable d'environnement MY_CPU_REQUEST tire sa valeur du champ MY_CPU_REQUEST d'un conteneur nommé test-container . De même, les autres variables d'environnement obtiennent leurs valeurs à partir des champs Container.

stocké des champs Pod dans un DownwardAPIVolumeFile: comment un Pod peut utiliser un DownwardAPIVolumeFile pour exposer des informations sur lui-même à des Conteneurs s'exécutant dans le Pod: Un DownwardAPIVolumeFile peut exposer les champs Pod et les champs Conteneur

Stocker les champs Pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: kubernetes-downwardapi-volume-example
  labels:
    zone: us-est-coast
    cluster: test-cluster1
    rack: rack-22
  annotations:
    build: two
    builder: john-doe
spec:
  containers:
    - name: client-container
      image: k8s.gcr.io/busybox
      command: ["sh", "-c"]
      args:
      - while true; do
          if [[ -e /etc/podinfo/labels ]]; then
            echo -en '\n\n'; cat /etc/podinfo/labels; fi;
```

```
          if [[ -e /etc/podinfo/annotations ]]; then
            echo -en '\n\n'; cat /etc/podinfo/annotations; fi;
          sleep 5;
        done;
      volumeMounts:
        - name: podinfo
          mountPath: /etc/podinfo
          readOnly: false
  volumes:
    - name: podinfo
      downwardAPI:
        items:
          - path: "labels"
            fieldRef:
              fieldPath: metadata.labels
          - path: "annotations"
            fieldRef:
              fieldPath: metadata.annotations
```

vous pouvez voir que le pod a un volume /etc/podinfo et que le conteneur monte le volume dans /etc/podinfo . Regardez le tableau des items sous downwardAPI . Chaque élément du tableau est un DownwardAPIVolumeFile . Le premier élément spécifie que la valeur du champ metadata.labels du Pod doit être stockée dans un fichier nommé labels . Le second élément spécifie que la valeur du champ d' annotations du Pod doit être stockée dans un fichier nommé annotations .

vous stockez les champs Conteneur dans un DownwardAPIVolumeFile:

```
apiVersion: v1
kind: Pod
metadata:
  name: kubernetes-downwardapi-volume-example-2
spec:
  containers:
    - name: client-container
      image: k8s.gcr.io/busybox:1.24
      command: ["sh", "-c"]
      args:
      - while true; do
          echo -en '\n';
          if [[ -e /etc/podinfo/cpu_limit ]]; then
            echo -en '\n'; cat /etc/podinfo/cpu_limit; fi;
          if [[ -e /etc/podinfo/cpu_request ]]; then
            echo -en '\n'; cat /etc/podinfo/cpu_request; fi;
          if [[ -e /etc/podinfo/mem_limit ]]; then
            echo -en '\n'; cat /etc/podinfo/mem_limit; fi;
          if [[ -e /etc/podinfo/mem_request ]]; then
            echo -en '\n'; cat /etc/podinfo/mem_request; fi;
          sleep 5;
        done;
      resources:
        requests:
```

```
          memory: "32Mi"
          cpu: "125m"
        limits:
          memory: "64Mi"
          cpu: "250m"
      volumeMounts:
        - name: podinfo
          mountPath: /etc/podinfo
          readOnly: false
  volumes:
    - name: podinfo
      downwardAPI:
        items:
          - path: "cpu_limit"
            resourceFieldRef:
              containerName: client-container
              resource: limits.cpu
          - path: "cpu_request"
            resourceFieldRef:
              containerName: client-container
              resource: requests.cpu
          - path: "mem_limit"
            resourceFieldRef:
              containerName: client-container
              resource: limits.memory
          - path: "mem_request"
            resourceFieldRef:
              containerName: client-container
              resource: requests.memory
```

vous pouvez voir que le pod a un volume /etc/podinfo et que le conteneur monte le volume dans /etc/podinfo . Regardez le tableau des items sous downwardAPI . Chaque élément du tableau est un DownwardAPIVolumeFile. Le premier élément spécifie que dans le conteneur nommé client-container , la valeur du champ limits.cpu doit être stockée dans un fichier nommé cpu_limit .

Capacités de l'API Downward: Les informations suivantes sont disponibles pour les conteneurs via les variables d'environnement et DownwardAPIVolumeFiles:

- Le nom du nœud
- L'adresse IP du nœud
- Le nom du Pod
- L'espace de noms du Pod
- L'adresse IP du Pod
- Le nom du compte de service du pod
- L'UID du Pod
- Limite du processeur d'un conteneur
- Demande de CPU d'un conteneur
- Limite de mémoire d'un conteneur
- Demande de mémoire d'un conteneur

En outre, les informations suivantes sont disponibles via DownwardAPIVolumeFiles.

- Les étiquettes du Pod
- Les annotations du Pod

PodPreset

Injecter des informations dans des pods à l'aide d'un PodPreset: Vous pouvez utiliser un objet podpreset pour injecter des informations telles que des secrets, des montages de volume et des variables d'environnement, etc. dans des pods au moment de la création.

Créer un préréglage de pod:

```
apiVersion: settings.k8s.io/v1alpha1
kind: PodPreset
metadata:
  name: allow-database
spec:
  selector:
    matchLabels:
      role: frontend
  env:
    - name: DB_PORT
      value: "6379"
  volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
    - name: cache-volume
      emptyDir: {}
```

Ceci est un exemple pour montrer comment une spécification de Pod est modifiée par le préréglage de Pod qui définit une variable ConfigMap pour les variables d'environnement:

Spécification de pod soumise par l'utilisateur:

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
spec:
  containers:
    - name: website
      image: nginx
      ports:
        - containerPort: 80
```

Utilisateur soumis ConfigMap :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: etcd-env-config
data:
  number_of_members: "1"
  initial_cluster_state: new
  initial_cluster_token: DUMMY_ETCD_INITIAL_CLUSTER_TOKEN
  discovery_token: DUMMY_ETCD_DISCOVERY_TOKEN
  discovery_url: http://etcd_discovery:2379
  etcdctl_peers: http://etcd:2379
  duplicate_key: FROM_CONFIG_MAP
  REPLACE_ME: "a value"
```

Exemple de préréglage de pod:

```
apiVersion: settings.k8s.io/v1alpha1
kind: PodPreset
metadata:
  name: allow-database
spec:
  selector:
    matchLabels:
      role: frontend
  env:
    - name: DB_PORT
      value: "6379"
    - name: duplicate_key
      value: FROM_ENV
    - name: expansion
      value: $(REPLACE_ME)
  envFrom:
    - configMapRef:
        name: etcd-env-config
  volumeMounts:
    - mountPath: /cache
      name: cache-volume
    - mountPath: /etc/app/config.json
      readOnly: true
      name: secret-volume
  volumes:
    - name: cache-volume
      emptyDir: {}
    - name: secret-volume
      secret:
         secretName: config-details
```

Pod spec après admission controller:

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
  annotations:
    podpreset.admission.kubernetes.io/podpreset-allow-database: "resource version"
spec:
  containers:
    - name: website
      image: nginx
      volumeMounts:
        - mountPath: /cache
          name: cache-volume
        - mountPath: /etc/app/config.json
          readOnly: true
          name: secret-volume
      ports:
        - containerPort: 80
      env:
        - name: DB_PORT
          value: "6379"
        - name: duplicate_key
          value: FROM_ENV
        - name: expansion
          value: $(REPLACE_ME)
      envFrom:
        - configMapRef:
            name: etcd-env-config
  volumes:
    - name: cache-volume
      emptyDir: {}
    - name: secret-volume
      secret:
         secretName: config-details
```

Suppression d'un préréglage de pod:

```
$ kubectl delete podpreset allow-database podpreset "allow-database" deleted
```

##RUN APPLICATION

objet Kubernetes Deployment

exécuter une application à l'aide d'un objet Kubernetes Deployment. Vous pouvez exécuter une application en créant un objet Déploiement Kubernetes et vous pouvez décrire un déploiement dans un fichier YAML.

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template: # create pods using pod definition in this template
    metadata:
      # unlike pod-nginx.yaml, the name is not included in the meta data as a
unique name is
      # generated from the deployment name
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Créez un déploiement basé sur le fichier YAML:

```
$ kubectl apply -f https://k8s.io/docs/tasks/run-application/deployment.yaml
```

Afficher des informations sur le déploiement:

```
$ kubectl describe deployment nginx-deployment
```

Répertoriez les modules créés par le déploiement:

```
$  kubectl get pods -l app=nginx
```

Afficher des informations sur un pod:

```
$  kubectl describe pod <pod-name>
```

Mise à jour du déploiement: Vous pouvez mettre à jour le déploiement en appliquant un nouveau fichier YAML.

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.8 # Update the version of nginx from 1.7.9 to 1.8
        ports:
        - containerPort: 80
```

Appliquez le nouveau fichier YAML:

```
$ kubectl apply -f https://k8s.io/docs/tasks/run-application/deployment-
update.yaml
```

Regardez le déploiement créer des pods avec de nouveaux noms et supprimer les anciens pods:

```
$  kubectl get pods -l app=nginx
```

Scaling de l'application en augmentant le nombre de réplicas: Vous pouvez augmenter le nombre de pods dans votre déploiement en appliquant un nouveau fichier YAML.

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 4 # Update the replicas from 2 to 4
  template:
    metadata:
```

```
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.8
        ports:
        - containerPort: 80
```

appliquer les changements et vérifier les résultat:

```
$ kubectl get pods -l app=nginx
```

Supprimer un déploiement:

```
$  kubectl delete deployment nginx-deployment
```

Le moyen préféré de créer une application répliquée consiste à utiliser un déploiement, qui à son tour utilise un ReplicaSet. Avant que le déploiement et ReplicaSet ont été ajoutés à Kubernetes, les applications répliquées ont été configurées à l'aide d'un ReplicationController .

Stateful à instance unique

Exécuter une application Stateful à instance unique: exécuter une application stateful à instance unique dans Kubernetes en utilisant un volume PersistentVolume et un déploiement.

```
apiVersion: v1
kind: Service
metadata:
  name: mysql
spec:
  ports:
  - port: 3306
  selector:
    app: mysql
  clusterIP: None
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
spec:
  accessModes:
    - ReadWriteOnce
```

```
    resources:
      requests:
        storage: 20Gi
---
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - image: mysql:5.6
        name: mysql
        env:
          # Use secret in real usage
        - name: MYSQL_ROOT_PASSWORD
          value: password
        ports:
        - containerPort: 3306
          name: mysql
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: mysql-persistent-storage
        persistentVolumeClaim:
          claimName: mysql-pv-claim
```

lister les pods:

```
$ kubectl get pods -l app=mysql
```

Inspect the PersistentVolumeClaim:

```
$ kubectl describe pvc mysql-pv-claim
```

Accéder à l'instance MySQL: Le fichier YAML précédent crée un service qui permet à d'autres modules du cluster d'accéder à la base de données. L'option de service clusterIP: None permet au service DNS de se

résoudre directement en adresse IP du pod. C'est optimal quand vous n'avez qu'un Pod derrière un Service et que vous n'avez pas l'intention d'augmenter le nombre de Pods.

Exécutez un client MySQL pour vous connecter au serveur:

```
$  kubectl run -it --rm --image=mysql:5.6 --restart=Never mysql-client -- mysql -h
mysql -ppassword
```

Cette commande crée un nouveau Pod dans le cluster qui exécute un client MySQL et le connecte au serveur via le Service. S'il se connecte, vous savez que votre base de données MySQL avec état est en cours d'exécution. Waiting for pod default/mysql-client-274442439-zyp6i to be running, status is Pending, pod ready: false If you don't see a command prompt, try pressing enter. mysql>

*Utiliser la strategy: type: Recreate dans le fichier YAML de configuration de déploiement. Cela indique à Kubernetes de ne pas utiliser les mises à jour tournantes. Les mises à jour tournantes ne fonctionneront pas, car vous ne pouvez pas faire tourner plus d'un pod à la fois. La stratégie Recreate arrêtera le premier pod avant d'en créer un nouveau avec la configuration mise à jour.

Supprimer un déploiement:

```
$ kubectl delete deployment,svc mysql
$ kubectl delete pvc mysql-pv-claim
```

Si vous avez manuellement provisionné un PersistentVolume, vous devez également le supprimer manuellement et libérer la ressource sous-jacente. Si vous avez utilisé un provisionneur dynamique, il supprime automatiquement le volume PersistentVolume lorsqu'il voit que vous avez supprimé PersistentVolumeClaim.

---

application Stateful répliquée

Exécuter une application Stateful répliquée: Cette page montre comment exécuter une application dynamique répliquée à l'aide d'un contrôleur StatefulSet . L'exemple est une topologie à un seul maître MySQL avec plusieurs esclaves exécutant une réplication asynchrone.

L'exemple de déploiement de MySQL se compose d'un fichier ConfigMap, de deux services et d'un StatefulSet. Créez le fichier ConfigMap :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql
  labels:
    app: mysql
data:
```

```
  master.cnf: |
    # Apply this config only on the master.
    [mysqld]
    log-bin
  slave.cnf: |
    # Apply this config only on slaves.
    [mysqld]
    super-read-only
```

Ce ConfigMap fournit des substitutions my.cnf qui vous permettent de contrôler indépendamment la configuration sur le maître MySQL et les esclaves. Dans ce cas, vous souhaitez que le maître puisse servir les journaux de réplication aux esclaves et que vous souhaitiez que les esclaves rejettent les écritures qui ne proviennent pas de la réplication.

Chaque Pod détermine la portion à regarder dans le configmap lors de l'initialisation, en fonction des informations fournies par le contrôleur StatefulSet.

Créez les services :

```
# Headless service for stable DNS entries of StatefulSet members.
apiVersion: v1
kind: Service
metadata:
  name: mysql
  labels:
    app: mysql
spec:
  ports:
  - name: mysql
    port: 3306
  clusterIP: None
  selector:
    app: mysql
---
# Client service for connecting to any MySQL instance for reads.
# For writes, you must instead connect to the master: mysql-0.mysql.
apiVersion: v1
kind: Service
metadata:
  name: mysql-read
  labels:
    app: mysql
spec:
  ports:
  - name: mysql
    port: 3306
  selector:
    app: mysql
```

Le service Headless héberge les entrées DNS que le contrôleur StatefulSet crée pour chaque pod faisant partie de l'ensemble. Comme le service Headless est nommé mysql , les pods sont accessibles en résolvant .mysql depuis n'importe quel autre pod du même cluster et espace-noms Kubernetes.

Le service client, appelé mysql-read , est un service normal avec son propre IP de cluster qui distribue les connexions sur tous les pods MySQL déclarant être prêts. L'ensemble des points d'extrémité potentiels comprend le maître MySQL et tous les esclaves.

Notez que seules les requêtes en lecture peuvent utiliser le service client à charge équilibrée. Parce qu'il n'y a qu'un seul maître MySQL, les clients doivent se connecter directement au Pod principal MySQL (via son entrée DNS dans le Headless Service) pour exécuter des écritures.

```
StatefulSet:
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  serviceName: mysql
  replicas: 3
  template:
    metadata:
      labels:
        app: mysql
    spec:
      initContainers:
      - name: init-mysql
        image: mysql:5.7
        command:
        - bash
        - "-c"
        - |
          set -ex
          # Generate mysql server-id from pod ordinal index.
          [[ `hostname` =~ -([0-9]+)$ ]] || exit 1
          ordinal=${BASH_REMATCH[1]}
          echo [mysqld] > /mnt/conf.d/server-id.cnf
          # Add an offset to avoid reserved server-id=0 value.
          echo server-id=$((100 + $ordinal)) >> /mnt/conf.d/server-id.cnf
          # Copy appropriate conf.d files from config-map to emptyDir.
          if [[ $ordinal -eq 0 ]]; then
            cp /mnt/config-map/master.cnf /mnt/conf.d/
          else
            cp /mnt/config-map/slave.cnf /mnt/conf.d/
          fi
        volumeMounts:
        - name: conf
          mountPath: /mnt/conf.d
        - name: config-map
```

```
          mountPath: /mnt/config-map
      - name: clone-mysql
        image: gcr.io/google-samples/xtrabackup:1.0
        command:
        - bash
        - "-c"
        - |
          set -ex
          # Skip the clone if data already exists.
          [[ -d /var/lib/mysql/mysql ]] && exit 0
          # Skip the clone on master (ordinal index 0).
          [[ `hostname` =~ -([0-9]+)$ ]] || exit 1
          ordinal=${BASH_REMATCH[1]}
          [[ $ordinal -eq 0 ]] && exit 0
          # Clone data from previous peer.
          ncat --recv-only mysql-$(($ordinal-1)).mysql 3307 | xbstream -x -C
/var/lib/mysql
          # Prepare the backup.
          xtrabackup --prepare --target-dir=/var/lib/mysql
        volumeMounts:
        - name: data
          mountPath: /var/lib/mysql
          subPath: mysql
        - name: conf
          mountPath: /etc/mysql/conf.d
      containers:
      - name: mysql
        image: mysql:5.7
        env:
        - name: MYSQL_ALLOW_EMPTY_PASSWORD
          value: "1"
        ports:
        - name: mysql
          containerPort: 3306
        volumeMounts:
        - name: data
          mountPath: /var/lib/mysql
          subPath: mysql
        - name: conf
          mountPath: /etc/mysql/conf.d
        resources:
          requests:
            cpu: 500m
            memory: 1Gi
        livenessProbe:
          exec:
            command: ["mysqladmin", "ping"]
          initialDelaySeconds: 30
          periodSeconds: 10
          timeoutSeconds: 5
        readinessProbe:
          exec:
            # Check we can execute queries over TCP (skip-networking is off).
            command: ["mysql", "-h", "127.0.0.1", "-e", "SELECT 1"]
```

```
          initialDelaySeconds: 5
          periodSeconds: 2
          timeoutSeconds: 1
      - name: xtrabackup
        image: gcr.io/google-samples/xtrabackup:1.0
        ports:
        - name: xtrabackup
          containerPort: 3307
        command:
        - bash
        - "-c"
        - |
          set -ex
          cd /var/lib/mysql

          # Determine binlog position of cloned data, if any.
          if [[ -f xtrabackup_slave_info ]]; then
            # XtraBackup already generated a partial "CHANGE MASTER TO" query
            # because we're cloning from an existing slave.
            mv xtrabackup_slave_info change_master_to.sql.in
            # Ignore xtrabackup_binlog_info in this case (it's useless).
            rm -f xtrabackup_binlog_info
          elif [[ -f xtrabackup_binlog_info ]]; then
            # We're cloning directly from master. Parse binlog position.
            [[ `cat xtrabackup_binlog_info` =~ ^(.*?)[[:space:]]+(.*?)$ ]] || exit
1
            rm xtrabackup_binlog_info
            echo "CHANGE MASTER TO MASTER_LOG_FILE='${BASH_REMATCH[1]}',\
                  MASTER_LOG_POS=${BASH_REMATCH[2]}" > change_master_to.sql.in
          fi

          # Check if we need to complete a clone by starting replication.
          if [[ -f change_master_to.sql.in ]]; then
            echo "Waiting for mysqld to be ready (accepting connections)"
            until mysql -h 127.0.0.1 -e "SELECT 1"; do sleep 1; done

            echo "Initializing replication from clone position"
            # In case of container restart, attempt this at-most-once.
            mv change_master_to.sql.in change_master_to.sql.orig
            mysql -h 127.0.0.1 <<EOF
          $(<change_master_to.sql.orig),
            MASTER_HOST='mysql-0.mysql',
            MASTER_USER='root',
            MASTER_PASSWORD='',
            MASTER_CONNECT_RETRY=10;
          START SLAVE;
          EOF
          fi

          # Start a server to send backups when requested by peers.
          exec ncat --listen --keep-open --send-only --max-conns=1 3307 -c \
            "xtrabackup --backup --slave-info --stream=xbstream --host=127.0.0.1 -
-user=root"
        volumeMounts:
```

```
        - name: data
          mountPath: /var/lib/mysql
          subPath: mysql
        - name: conf
          mountPath: /etc/mysql/conf.d
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
      volumes:
      - name: conf
        emptyDir: {}
      - name: config-map
        configMap:
          name: mysql
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 10Gi
```

Vous pouvez regarder la progression du démarrage en exécutant: $ kubectl get pods -l app=mysql --watch

Ce manifeste utilise une variété de techniques pour gérer les Pods dynamiques dans le cadre d'un StatefulSet.

Compréhension de l'initialisation dynamique du Pod: Le contrôleur StatefulSet démarre les Pods une à la fois, dans l'ordre par leur index ordinal. Il attend que chaque Pod indique être prêt avant de commencer le suivant.De plus, le contrôleur attribue à chaque Pod un nom unique et stable de la forme - .

Générer la configuration:

https://kubernetes.io/docs/tasks/run-application/run-replicated-stateful-application/

Contrôle d'accès:

Le stockage configuré avec un ID de groupe (GID) permet d'écrire uniquement par Pods en utilisant le même GID. Les GID non concordants ou manquants provoquent des erreurs d'autorisation refusées. Pour réduire le besoin de coordination avec les utilisateurs, un administrateur peut annoter un PersistentVolume avec un GID. Ensuite, le GID est automatiquement ajouté à tout Pod qui utilise le PersistentVolume.

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: pv1
  annotations:
    pv.beta.kubernetes.io/gid: "1234"
```

Quand un Pod consomme un PersistentVolume qui a une annotation GID, le GID annoté est appliqué à tous les Conteneurs dans le Pod de la même manière que les GID spécifiés dans le contexte de sécurité du Pod.