Administration d’applications Kubernetes

Déploiement d’application

1.Manipuler notre premier pod

Création en mode impératif

```
ludo@kubernetes:~$ kubectl run nginx-ludo –image=nginx:latest
```

Exposer le pod sur extérieur avec un service NodePort

```
ludo@kubernetes:~$ kubectl expose pod nginx-ludo --type NodePort --port 80

ludo@kubernetes:~$ kubectl get service
NAME                TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
nginx    NodePort    10.99.252.217   <none>                  80:31070/TCP
```

Afficher les Endpoint du service

```
ludo@kubernetes:~$ kubectl get pod nginx-ludo -o wide
NAME    READY   STATUS    RESTARTS   AGE   IP               NODE
nginx   1/1     Running   0          68s   192.168.196.92   worker04

ludo@kubernetes:~$ kubectl get endpoints nginx
NAME    ENDPOINTS           AGE
nginx   192.168.196.92:80   5m6s
```

On « requête » le pod

On rentre dans le pod et on modifie le ficher html

```
ludo@kubernetes:~$ kubectl exec -it nginx-ludo -- bash root@nginx-
ludo:/# echo "<h1>Bienvenue sur le serveur de Ludo</h1>" \
   /usr/share/nginx/html/index.html
```

On supprime le pod et le service

```
ludo@kubernetes:~$ kubectl delete pod
nginx pod "nginx" deleted

ludo@kubernetes:~$ kubectl delete service
nginx service "nginx" deleted
```

2.Gérer les pods avec des manifestes

Se rendre dans le répertoires $HOME/formation/pod , ouvrir le fichier nginx-pod.yaml et on change le nom du pod.

```
ludo@kubernetes:/pod$ vim nginx-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-ludo
  labels:
    app: web
spec:
  containers:
  - image: nginx
    name: nginx
```

On applique le manifeste à l'api server

```
ludo@kubernetes:/pod$ kubectl appply -f nginx-pod.yaml
```

Exposer un pod avec un manifeste

```
ludo@kubernetes:/pod$ vim service-nginx.yaml
apiVersion: v1
kind: Service
metadata:
  name: nodeport
spec:
  type: NodePort
  selector:
    app: web
  ports:
    - port: 80
      targetPort: 80
```

On applique

```
ludo@kubernetes:/pod$ kubectl apply -f service-nginx.yaml
```

La gestion des sondes Kubernetes

La LivenessProbe

Ouvrir le fichier kuard-pod-liveness.yaml

```
ludo@kubernetes:/pod$ vim kuard-pod-liveness.yaml
apiVersion: v1
kind: Pod
metadata:
  name: kuard-ludo
  labels:
    app: liveness-ludo
spec:
  containers:
  - image: gcr.io/kuar-demo/kuard-amd64:1
    name: kuard
    livenessProbe:
      httpGet:
        path: /healthy
        port: 8080
      initialDelaySeconds:
      5 timeoutSeconds: 1
      periodSeconds: 10
      failureThreshold: 2
    ports:
    - containerPort: 8080
      name: http
```

On change le nom du pod, les labels et on applique

```
ludo@kubernetes:/pod$ kubectl apply -f kuard-pod-liveness.yaml
pod/kuard-ludo created
```

On expose l'application

```
ludo@kubernetes:/pod$ vim service-kuard.yaml
apiVersion: v1
kind: Service
metadata:
  name: kuard-ludo
spec:
  type: NodePort
  selector:
    app: demo-ludo
  ports:
    - port: 8080
      targetPort: 8080
```

On applique le manifeste

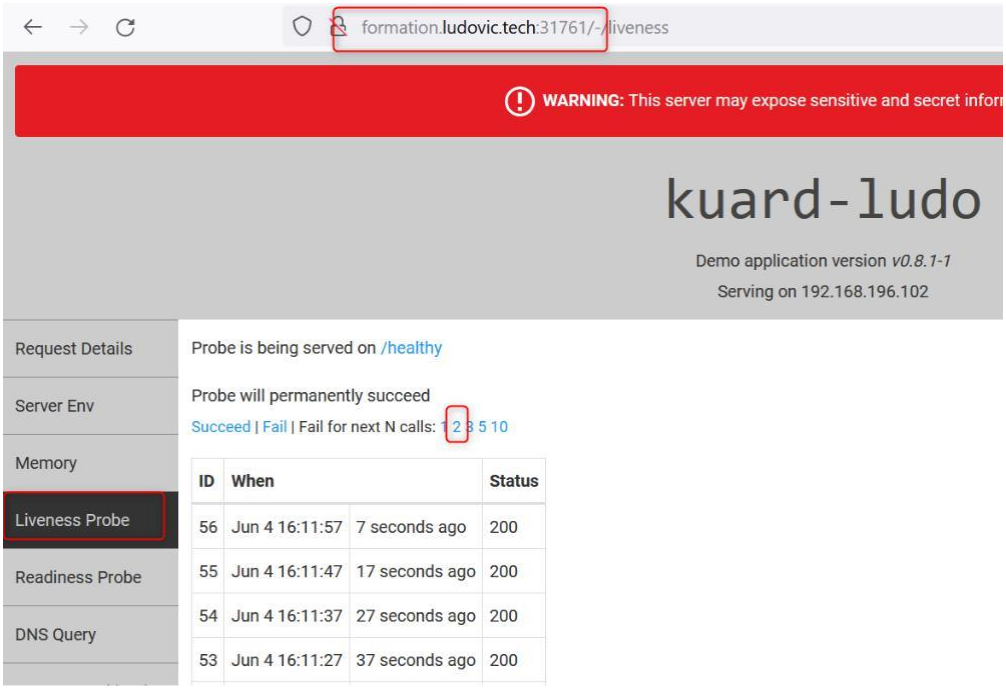
```
ludo@kubernetes:/pod$ kubectl apply -f service-kuard.yaml
service/kuard-ludo created
```

On liste les ressources

```
ludo@kubernetes:/pod$ kubectl get all
NAME             READY   STATUS    RESTARTS   AGE
pod/kuard-ludo   1/1     Running   0          2m10s

NAME                 TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
AGE
service/kuard-ludo   NodePort    10.111.216.187   <none>        8080:30101/TCP
10s
```

On teste la sonde




```
ludo@kubernetes:/pod$ kubectl get pod kuard-ludo

Every 2.0s: kubectl get pod
kubernetes.ludovic.tech: Sun Jun  4 18:14:13 2023

NAME         READY   STATUS    RESTARTS   AGE
kuard-ludo   1/1     Running   0          11m

NAME         READY   STATUS    RESTARTS      AGE
kuard-ludo   1/1     Running   1 (9s ago)    12m
```

On supprime tout

```
ludo@kubernetes:/pod$ kubectl delete all --selector app=demo-ludo
```

La sonde ReadinessProbe

Ouvrir le fichier kuard-pod-readiness.yaml

```
ludo@kubernetes:/pod$ vim kuard-pod- readiness.yaml
apiVersion: v1
kind: Pod
metadata:
  name: kuard
  labels:
    app: demo
spec:
  containers:
  - image: gcr.io/kuar-demo/kuard-amd64:1
    name: kuard
    ports:
    - containerPort: 8080
      name: http
      protocol: TCP
   readinessProbe:
      httpGet:
        path: /ready
        port: 8080
      initialDelaySeconds: 30
      timeoutSeconds: 1
      periodSeconds: 10
      failureThreshold: 1
```

On change le nom du pod, les labels et on applique

```
ludo@kubernetes:/pod$ kubectl apply -f kuard-pod- readiness.yaml
pod/kuard-ludo created
```

```
ludo@kubernetes:/pod$  kubectl get all --show-labels
NAME        READY   STATUS    RESTARTS   AGE   LABELS
pod/kuard   0/1     Running   0          7s    app=demo-ludo


ludo@kubernetes:/pod$  kubectl get all
NAME        READY   STATUS    RESTARTS   AGE
pod/kuard   1/1     Running   0          47s
```

On expose l'application

```
ludo@kubernetes:/pod$ vim service-kuard.yaml
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: kuard-ludo
spec:
  type: NodePort
  selector:
    app: demo-ludo
  ports:
    - port: 8080
      targetPort: 8080
```

On applique le manifeste

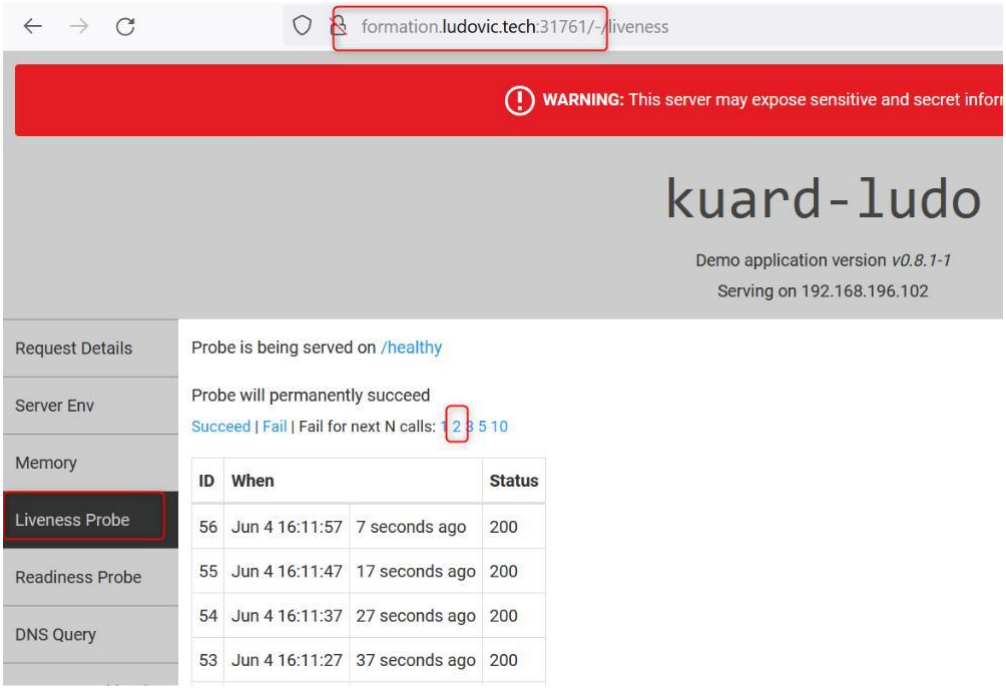
```
ludo@kubernetes:/pod$ kubectl apply -f service-kuard.yaml
service/nodeport created
```

On liste les ressources

```
ludo@kubernetes:/pod$ kubectl get all
NAME             READY   STATUS    RESTARTS   AGE
pod/kuard-ludo   1/1     Running   0          2m10s

NAME                 TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
AGE
service/kuard-ludo   NodePort    10.111.216.187   <none>        8080:30101/TCP
10s
```

On teste la sonde



```
ludo@kubernetes:/pod$  watch kubectl get pod kuard-ludo

Every 2.0s: kubectl get pod
kubernetes.ludovic.tech: Sun Jun  4 18:14:13 2023

NAME         READY   STATUS    RESTARTS   AGE
kuard-ludo   1/1     Running   0          11m

NAME         READY   STATUS    RESTARTS      AGE
kuard-ludo   1/1     Running   1 (9s ago)   12m
```

La gestion des ressources.

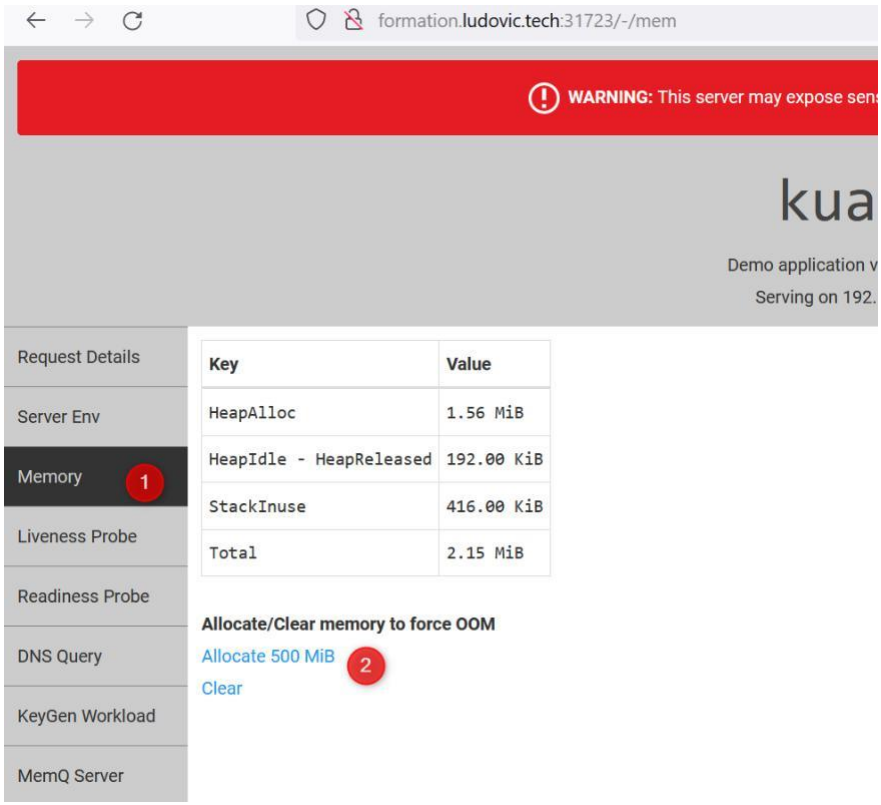
On ouvre le fichier kuard-pod-full.yaml

```
ludo@kubernetes:/pod$ vim kuard-pod-full.yaml
apiVersion: v1
kind: Pod
metadata:
  name: kuard #nom du pod
  labels:
   app: demo
  namespace: forma-ludo # nom de votre namescpace
spec:
  containers:
    - image: gcr.io/kuar-demo/kuard-amd64:1
      name: kuard
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
      resources:
        requests:
          cpu: "500m"
          memory: "128Mi"
        limits:
          cpu: "1000m"
          memory: "256Mi"
 ....
```

On applique après modification

```
ludo@kubernetes:/pod$ kubectl apply -f kuard-pod-full.yaml
pod/kuard configured
```

On test



Le pod redémarre

```
ludo@kubernetes:/pod$ watch kubectl get pod
Every 2.0s: kubectl get pod                  kubernetes.ludovic.tech: Mon Jun
5 18:11:33 2023

NAME    READY   STATUS    RESTARTS      AGE
kuard   1/1     Running   1 (74s ago)   113s
```

TP – L'InitContainer.

Comprendre les init containers

Un Pod peut avoir plusieurs conteneurs exécutant des applications mais peut aussi avoir un ou plusieurs init containers, qui sont exécutés avant que les conteneurs d'application ne démarrent.

Les init containers se comportent comme les conteneurs réguliers, avec quelques différences :

- Les init containers s'exécutent toujours jusqu'à la complétion.
- Chaque init container doit se terminer avec succès avant que le prochain ne démarre.

Si le init container d'un Pod échoue, Kubernetes redémarre le Pod à répétition jusqu'à ce que le init container se termine avec succès. Cependant, si le Pod a une restartPolicy à "Never", Kubernetes ne redémarre pas le Pod.

Afin de spécifier un init container pour un Pod, il faut ajouter le champ initContainers dans la spécification du Pod. Le statut des init containers est retourné dans le champ .status.initContainerStatuses comme un tableau des statuts du conteneur (comparable au champ .status.containerStatuses)

Les init containers supportent tous les champs et fonctionnalités des conteneurs d'application incluant les limites de ressources, les volumes et les paramètres de sécurité.

On ouvre le fichier initcontaines.yaml, dans le répertoire formation/initcontainers

```
ludo@kubernetes:/initcontainers$ vim initcontainers.yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app.kubernetes.io/name: MyApp
spec:
  containers:
  - name: myapp-container
    image: busybox:1.28
    command: ['sh', '-c', 'echo "L''app s''exécute!" && sleep 3600']
  initContainers:
  - name: init-myservice
    image: busybox:1.28
    command: ['sh', '-c', "until nslookup myservice.$(cat
/var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local; do
echo en attente de myservice; sleep 2; done"]
  - name: init-mydb
```

```
    image: busybox:1.28
    command: ['sh', '-c', "until nslookup mydb.$(cat
/var/run/secrets/kubernetes.io/serviceaccount/namespace).svc.cluster.local; do
echo en attente de mydb; sleep 2; done"]
```

Cet exemple définit un simple Pod possédant deux init containers. Le premier attend myservice et le second attend mydb. Une fois que les deux init containers terminent leur exécution, le Pod exécute le conteneur d'application décrit dans sa section spec.

On applique le manifest à l'API :

```
ludo@kubernetes:/initcontainers$ kubectl apply -f initcontaines.yaml
pod/myapp-pod created
```

Et vérifier son statut :

```
ludo@kubernetes:/initcontainers: kubectl get -f initcontaines.yaml
NAME        READY     STATUS     RESTARTS   AGE
myapp-pod   0/1       Init:0/2   0


ludo@kubernetes:/initcontainers:  kubectl describe -f initcontaines.yaml

Name:          myapp-pod
Namespace:     default
[...]
Labels:        app.kubernetes.io/name=MyApp
Status:        Pending
[...]
Init Containers:
  init-myservice:
[...]
    State:         Running
[...]
  init-mydb:
[...]
    State:         Waiting
      Reason:      PodInitializing
    Ready:         False
[...]
Containers:
  myapp-container:
[...]
    State:         Waiting
      Reason:      PodInitializing
    Ready:         False
```

Pour voir les logs des init containers dans ce Pod, exécuter :

```
ludo@kubernetes:/initcontainers:$ kubectl logs myapp-pod -c init-myservice
# Inspecter le premier init container
ludo@kubernetes:/initcontainers:$ kubectl logs myapp-pod -c init-mydb
# Inspecter le second init container
```

À ce stade, ces init containers attendent de découvrir les services nommés mydb et myservice.

On créer les Services avec le manifest services.yaml :

```
ludo@kubernetes:/initcontainers:$ kubectl apply services.yaml
service/myservice created
service/mydb created
```

Pour voir les logs des init containers dans ce Pod, exécuter :

```
ludo@kubernetes:/initcontainers:$ kubectl logs myapp-pod -c init-myservice
# Inspecter le premier init container
ludo@kubernetes:/initcontainers:$ kubectl logs myapp-pod -c init-mydb
# Inspecter le second init container
```

```
ludo@kubernetes:/initcontainers:$ kubectl get pod
NAME        READY     STATUS    RESTARTS   AGE
myapp-pod   1/1       Running   0          9m
```

Comportement détaillé

Pendant le démarrage d'un Pod, chaque init container démarre en ordre, après que le réseau et les volumes ont été initialisés. Chaque conteneur doit se terminer avec succès avant que le prochain ne démarre. Si un conteneur n'arrive pas à démarrer à cause d'un problème d'exécution ou se termine avec un échec, il est redémarré selon la restartPolicy du Pod. Toutefois, si la restartPolicy du Pod est configurée à "Always", les init containers utilisent la restartPolicy "OnFailure".

Un Pod ne peut pas être Ready tant que tous les init containers ne se sont pas exécutés avec succès. Les ports d'un init container ne sont pas agrégés sous un Service. Un Pod qui s'initialise est dans l'état Pending mais devrait avoir une condition Initialized configurée à "true".

Si le Pod redémarre ou est redémarré, tous les init containers doivent s'exécuter à nouveau.

Les changements aux spec d'un init containers sont limités au champ image du conteneur. Changer le champ image d'un init container équivaut à redémarrer le Pod.

Puisque les init containers peuvent être redémarrés, réessayés ou ré-exécutés, leur code doit être idempotent. En particulier, le code qui écrit dans des fichiers sur EmptyDirs devrait être préparé à la possibilité qu'un fichier de sortie existe déjà.

Débogage des Pods

Débogage des Pods

La première étape du débogage d'un module consiste à l'examiner. Vérifiez l'état actuel du pod et les événements récents à l'aide de la commande suivante :

```
ludo@kubernetes:$ kubectl describe pods ${POD_NAME}
```

Regardez l'état des conteneurs dans le pod. Sont-ils tous en cours d'exécution ? Y a-t-il eu des redémarrages récents ?

Poursuivre le débogage en fonction de l'état des pods.

Mon pod reste en attente (Pending)

Si un pod est bloqué en attente, pending, cela signifie qu'il ne peut pas être ordonnancé sur un nœud. En général, c'est parce qu'il n'y a pas assez de ressources d'un type ou d'un autre, ce qui empêche la planification. Regardez la sortie de la commande kubectl describe ... ci-dessus. Il devrait y avoir des messages de l'ordonnanceur expliquant pourquoi il ne peut pas programmer votre pod. Les raisons sont les suivantes :

Vous n'avez pas assez de ressources : Dans ce cas, vous devez supprimer des pods, ajuster les demandes de ressources ou ajouter de nouveaux nœuds à votre cluster. Voir le document Compute Resources pour plus d'informations.

Vous utilisez hostPort : Lorsque vous liez un pod à un hostPort, le nombre d'endroits où ce pod peut être planifié est limité. Dans la plupart des cas, hostPort n'est pas nécessaire, essayez d'utiliser un objet Service pour exposer votre Pod. Si vous avez besoin d'un hostPort, vous ne pouvez programmer qu'autant de pods qu'il y a de nœuds dans votre cluster Kubernetes.

Mon pod reste en attente (Waiting)

Si un Pod est bloqué dans l'état Waiting, c'est qu'il a été planifié sur un nœud de travail, mais qu'il ne peut pas s'exécuter sur cette machine. Là encore, les informations fournies par kubectl describe ... devraient être utiles. La cause la plus fréquente des pods en attente est l'échec de l'extraction de l'image. Il y a trois choses à vérifier :

Assurez-vous que le nom de l'image est correct.
Avez-vous poussé l'image dans le registre ?
Essayez d'extraire manuellement l'image pour voir si elle peut être extraite. Par exemple, si vous utilisez Docker sur votre PC, exécutez docker pull <image>.

Mon pod se bloque ou n'est pas en bonne santé

Une fois que votre pod a été planifié, les méthodes décrites dans Debug Running Pods sont disponibles pour le débogage.
Mon module s'exécute mais ne fait pas ce que je lui ai dit de faire

Si votre pod ne se comporte pas comme vous l'aviez prévu, il se peut qu'il y ait une erreur dans la description du pod (par exemple dans le fichier mypod.yaml sur votre machine locale), et que cette erreur ait été ignorée silencieusement lorsque vous avez créé le pod. Souvent, une section de la description du pod est mal imbriquée, ou le nom d'une clé est mal saisi, et la clé est donc ignorée. Par exemple, si vous avez mal orthographié "command" en "commnd", le module sera créé mais n'utilisera pas la ligne de commande que vous vouliez qu'il utilise.

La première chose à faire est de supprimer votre pod et d'essayer de le créer à nouveau avec l'option --validate. Par exemple, lancez

```
ludo@kubernetes:$ kubectl apply --validate -f mypod.yaml
```

Si vous avez mal orthographié command comme commnd, vous obtiendrez une erreur comme celle-ci :

```
I0805 10:43:25.129850   46757 schema.go:126] unknown field: commnd
I0805 10:43:25.129973   46757 schema.go:129] this may be a false alarm, see
https://github.com/kubernetes/kubernetes/issues/6842
pods/mypod
```

Les événements tels que ceux que vous avez vus à la fin de kubectl describe pod sont conservés dans etcd et fournissent des informations de haut niveau sur ce qui se passe dans le cluster. Pour lister tous les événements, vous pouvez utiliser

```
ludo@kubernetes:$  kubectl get events
LAST SEEN   TYPE      REASON                 OBJECT
MESSAGE
59m         Normal    Scheduled              pod/busy
Successfully assigned formation/busy to node2.ludovic.io
59m         Normal    Pulling                pod/busy
Pulling image "busybox"
```

Pour lister tous les événements dans un naespace, vous pouvez utiliser :

```
ludo@kubernetes:$   kubectl get events -n cilium-test
LAST SEEN   TYPE      REASON             OBJECT
   MESSAGE
35s         Warning   FailedScheduling   pod/echo-external-node-669b679d-6pzp8
  0/4 nodes are available: 4 node(s) didn't match Pod's node
affinity/selector. preemption: 0/4 nodes are available: 4 Preemption is not
helpful for scheduling..
```

En plus de kubectl describe pod, une autre façon d'obtenir des informations supplémentaires sur un pod (au-delà de ce qui est fourni par kubectl get pod) est de passer l'option -o yaml output format à kubectl get pod. Cela vous donnera, au format YAML, encore plus d'informations que kubectl describe pod--essentiellement toutes les informations que le système possède sur le pod. Ici, vous verrez des choses comme les annotations (qui sont des métadonnées clé-valeur sans les restrictions d'étiquette, qui sont utilisées en interne par les composants du système Kubernetes), la politique de redémarrage, les ports et les volumes.

```
ludo@kubernetes:$  kubectl get pod nginx-deployment-1006230814-6winp -o yaml
apiVersion: v1
kind: Pod
metadata:

....
....
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: "2022-02-17T21:51:01Z"
    status: "True"
    type: Initialized
  - lastProbeTime: null
    lastTransitionTime: "2022-02-17T21:51:06Z"
    status: "True"
    type: Ready
 containerStatuses:
  - containerID:
containerd://5403af59a2b46ee5a23fb0ae4b1e077f7ca5c5fb7af16e1ab21c00e0e616462a
    image: docker.io/library/nginx:latest
    imageID:
docker.io/library/nginx@sha256:2834dc507516af02784808c5f48b7cbe38b8ed5d0f4837f
16e78d00deb7e7767
    lastState: {}
    name: nginx
    ready: true
    restartCount: 0
    started: true
    state:
      running:
        startedAt: "2022-02-17T21:51:05Z"
  hostIP: 192.168.0.113
  phase: Running
  podIP: 10.88.0.3
  podIPs:
  - ip: 10.88.0.3
  - ip: 2001:db8::1
  qosClass: Guaranteed
  startTime: "2022-02-17T21:51:01Z"
    ....
```

Examen des journaux de pods

Commencez par consulter les journaux du conteneur concerné :

```
ludo@kubernetes:$ kubectl logs ${POD_NAME} ${NOM_DU_CONTENEUR}
```

Si votre conteneur s'est déjà « crashé », vous pouvez accéder au journal du conteneur précédent avec :

```
ludo@kubernetes:$ kubectl logs --previous ${POD_NAME} ${CONTAINER_NAME}
```

Débogage avec un conteneur de débogage éphémère

Les conteneurs éphémères sont utiles pour le dépannage interactif lorsque kubectl exec est insuffisant parce qu'un conteneur s'est chrashé ou qu'une image de conteneur n'inclut pas d'utilitaires de débogage.

Exemple de débogage à l'aide de conteneurs éphémères

Vous pouvez utiliser la commande kubectl debug pour ajouter des conteneurs éphémères à un pod en cours d'exécution. Tout d'abord, créez un pod pour l'exemple :

```
ludo@kubernetes:$ kubectl run ephemeral-demo --image=registry.k8s.io/pause:3.1
--restart=Never
```

Les exemples de cette section utilisent l'image du conteneur pause car elle ne contient pas d'utilitaires de débogage, mais cette méthode fonctionne avec toutes les images de conteneur. Si vous essayez d'utiliser kubectl exec pour créer un shell, vous verrez une erreur car il n'y a pas de shell dans cette image de conteneur.

```
ludo@kubernetes:$ kubectl exec -it ephemeral-demo – sh

error: Internal error occurred: error executing command in container: failed
to exec in container: failed to start exec OCI runtime exec failed: exec
failed: unable to start container process: exec: "sh": executable file not
found in $PATH: unknown
```

Vous pouvez alors ajouter un conteneur de débogage en utilisant kubectl debug. Si vous spécifiez l'argument -i ou --interactive, kubectl s'attachera automatiquement à la console du conteneur éphémère.

```

ludo@kubernetes:$ kubectl debug -it ephemeral-demo --image=busybox:1.28 --
target=ephemeral-demo
```

Le nom du conteneur de débogage est par défaut debugger-8xzrl.

Si vous ne voyez pas d'invite de commande, essayez d'appuyer sur Entrée. Cette commande ajoute un nouveau conteneur busybox et s'y attache. Le paramètre --target cible l'espace de noms de processus d'un autre conteneur. Il est nécessaire ici car kubectl run n'active pas le partage de l'espace de noms de processus dans le pod qu'il crée. Le paramètre --target doit être supporté par le Container Runtime. S'il n'est pas pris en charge, le conteneur éphémère peut ne pas être démarré, ou il peut être démarré avec un espace de noms de processus isolé de sorte que la commande ps ne révèle pas les processus dans d'autres conteneurs.

Vous pouvez visualiser l'état du conteneur éphémère nouvellement créé en utilisant kubectl describe :

```
ludo@kubernetes:$ kubectl describe pod ephemeral-demo

...
Ephemeral Containers:
  debugger-8xzrl:
    Container ID:
docker://b888f9adfd15bd5739fefaa39e1df4dd3c617b9902082b1cfdc29c4028ffb2eb
    Image:          busybox
    Image ID:       docker-
pullable://busybox@sha256:1828edd60c5efd34b2bf5dd3282ec0cc04d47b2ff9caa0b6d4f0
7a21d1c08084
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Wed, 12 Feb 2020 14:25:42 +0100
    Ready:          False
    Restart Count:  0
    Environment:    <none>
    Mounts:         <none>
```

Utilisez kubectl delete pour supprimer le pod lorsque vous avez terminé :

```
ludo@kubernetes:$ kubectl delete pod ephemeral-demo
```

Débogage à l'aide d'une copie du Pod

Parfois, les options de configuration du pod rendent difficile le dépannage dans certaines situations. Par exemple, vous ne pouvez pas exécuter kubectl exec pour dépanner votre conteneur si votre image de conteneur n'inclut pas de shell ou si votre application se bloque au démarrage. Dans ces situations, vous pouvez utiliser kubectl debug pour créer une copie du Pod avec des valeurs de configuration modifiées pour faciliter le débogage.

Copier un Pod lors de l'ajout d'un nouveau conteneur

L'ajout d'un nouveau conteneur peut être utile lorsque votre application fonctionne mais ne se comporte pas comme vous le souhaitez et que vous aimeriez ajouter des utilitaires de dépannage supplémentaires au Pod.

Par exemple, peut-être que les images de conteneurs de votre application sont construites sur busybox mais que vous avez besoin d'utilitaires de débogage qui ne sont pas inclus dans busybox. Vous pouvez simuler ce scénario en utilisant kubectl run :

```
ludo@kubernetes:$ kubectl run myapp --image=busybox:1.28 --restart=Never --
sleep 1d
```

Exécutez cette commande pour créer une copie de myapp nommée **myapp-debug** qui ajoute un nouveau conteneur Ubuntu pour le débogage :

```
ludo@kubernetes:$ kubectl debug myapp -it --image=ubuntu --share-processes --
copy-to=myapp-debug
```

Le nom du conteneur de débogage est par défaut debugger-w7xmf.
Si vous ne voyez pas d'invite de commande, essayez d'appuyer sur Entrée.

```
root@myapp-debug:/#
```

Note : kubectl debug génère automatiquement le nom du conteneur de débogage :

kubectl debug génère automatiquement un nom de conteneur si vous n'en choisissez pas un avec l'option **--container**.
L'option **-i** permet à kubectl debug de s'attacher au nouveau conteneur par défaut. Vous pouvez empêcher cela en spécifiant **--attach=false**. Si votre session est déconnectée, vous pouvez vous rattacher en utilisant **kubectl attach**.
L'option **--share-processes** permet aux conteneurs de ce Pod de voir les processus des autres conteneurs du Pod.

N'oubliez pas de nettoyer le Pod de débogage lorsque vous avez terminé :

```
ludo@kubernetes:$ kubectl delete pod myapp myapp-debug
```

Débogage via un shell sur le nœud
Si aucune de ces approches ne fonctionne, vous pouvez trouver le nœud sur lequel le pod s'exécute et créer un pod s'exécutant sur le nœud. Pour créer un shell interactif sur un nœud à l'aide de kubectl debug, exécutez :

```
ludo@kubernetes:$ kubectl debug node/mynode -it --image=ubuntu
```

Création du pod de débogage node-debugger-mynode-xxxx avec le débogueur de conteneur sur le nœud mynode. Si vous ne voyez pas d'invite de commande, essayez d'appuyer sur Entrée.

```
root@ek8s-node:/#
```

Lors de la création d'une session de débogage sur un nœud, gardez à l'esprit que :

kubectl debug génère automatiquement le nom du nouveau pod en fonction du nom du nœud. Le système de fichiers racine du nœud sera monté sur /host.

Le conteneur fonctionne dans les espaces de noms IPC, Network et PID de l'hôte, bien que le pod ne soit pas privilégié, de sorte que la lecture de certaines informations de processus peut échouer et que le chroot /host peut échouer.
Si vous avez besoin d'un pod privilégié, créez-le manuellement.

N'oubliez pas de nettoyer le pod de débogage lorsque vous avez terminé :

```
ludo@kubernetes:$ kubectl delete pod node-debugger-mynode-pdx84
```

HorizontalPodAutoscaler

Introduction

Un HorizontalPodAutoscaler (HPA en abrégé) met automatiquement à jour une ressource de charge de travail (telle qu'un Deployment ou un StatefulSet), dans le but d'adapter automatiquement la charge de travail à la demande.

La mise à l'échelle horizontale signifie une réponse à l'augmentation de la charge qui consiste à déployer davantage de Pods. Cela diffère de la mise à l'échelle verticale qui, pour Kubernetes, consisterait à affecter davantage de ressources (par exemple : mémoire ou CPU) aux Pods déjà en cours d'exécution pour la charge de travail.

Si la charge diminue et que le nombre de Pods est supérieur au minimum configuré, le HorizontalPodAutoscaler demande à la ressource (Deployment, StatefulSet ou autre ressource similaire) de réduire ses instances.

Exemple d'HPA

Nous allons créer un HorizontalPodAutoscaler pour gérer automatiquement la mise à l'échelle d'un exemple d'application web. Cet exemple Workload est Apache httpd exécutant du code PHP.

Metrics Server doit être déployé dans le cluster, a été déployé et configuré. Le serveur de métriques Kubernetes collecte des métriques de ressources auprès des kubelets de votre cluster et expose ces métriques via l'API Kubernetes, en utilisant un APIService pour ajouter de nouveaux types de ressources qui représentent des lectures de métriques.

Exécuter et exposer le serveur php-apache

Vous allez d'abord démarrer un Déploiement qui exécute un conteneur en utilisant l'image hpa-example, et l'exposer en tant que Service en utilisant le manifeste suivant :

```
ludo@kubernetes:$ kubectl apply -f php-apache.yaml
```

Puis vous allez créer l'hpa. Service en utilisant le manifeste suivant :

```
ludo@kubernetes:$ kubectl apply -f hpa.yaml
```

Augmenter la charge

Ensuite, nous allons voir comment l'autoscaler réagit à une augmentation de la charge. Pour ce faire, vous allez démarrer un autre Pod qui servira de client. Le conteneur du Pod client tourne en boucle infinie, envoyant des requêtes au service php-apache :

```
ludo@kubernetes:$  kubectl run -i --tty load-generator --rm --
image=busybox:1.28 --restart=Never -- /bin/sh -c "while sleep 0.01; do wget -q
-O- http://php-apache; done"
```

Exécutez maintenant :

tapez Ctrl+C pour mettre fin à la surveillance quand vous êtes prêt

```
ludo@kubernetes:$  kubectl get hpa php-apache --watch
```

Au bout d'une minute environ, vous devriez constater une augmentation de la charge du processeur.