Exercices de base avec Kubernetes

1) Les pods dans Kubernetes

Un **pod** est un **ensemble de conteneurs** partageant un réseau et un espace de noms (**namespace**). Il constitue l'unité de base du déploiement dans Kubernetes. Tous les conteneurs d'un pod sont planifiés sur le même nœud.

Pour lancer un pod en utilisant le conteneur image **mhausenblas/simpleservice**: 0.5.0 et en exposant une API HTTP sur le **port 9876,** exécutez:

sudo kubectl run sise --image=mhausenblas/simpleservice:0.5.0 --port=9876

Nous pouvons maintenant voir que le pod est en cours d'exécution :

sudo kubectl get pods
```
NAME                  READY     STATUS    RESTARTS   AGE
sise-3210265840-k705b     1/1       Running   0          1m
```

sudo kubectl describe pod sise-3210265840-k705b | grep IP
```
IP:                 172.17.0.3
```

Depuis le cluster (par exemple via minishift ssh), ce pod est accessible via l'IP 172.17.0.3 du pod, que nous avons apprise à l'aide de la commande **kubectl describe** ci-dessus :

sudo curl 172.17.0.3:9876/info
```
{"host": "172.17.0.3:9876", "version": "0.5.0", "from": "172.17.0.1"}
```

Notez que kubectl run crée un déploiement. Pour le supprimer, vous devez exécuter **kubectl delete deployment sise.**

Utiliser le fichier de configuration

Vous pouvez également créer un pod à partir d'un fichier de configuration. Dans ce cas, le pod exécute l'image de **simpleservice** déjà connue et un conteneur CentOS générique :

sudo kubectl create -f pod.yaml

sudo kubectl get pods
```
NAME                  READY     STATUS    RESTARTS   AGE
twocontainers          2/2       Running   0          7s
```

Nous pouvons maintenant exécuter le conteneur CentOS et accéder à **simpleservice** sur localhost :

sudo kubectl exec twocontainers -c shell -i -t -- bash
[root@twocontainers /]# curl -s localhost:9876/info

{"host": "localhost:9876", "version": "0.5.0", "from": "127.0.0.1"}

Spécifiez le champ de ressources dans le pod pour influencer la quantité de CPU et/ou de RAM qu'un conteneur d'un pod peut utiliser (ici: 64 Mo de RAM et 0,5 CPU) :

sudo kubectl create -f constraint-pod.yaml

sudo kubectl describe pod constraintpod

```
...
Containers:
  sise:
    ...
    Limits:
      cpu:     500m
      memory:  64Mi
    Requests:
      cpu:     500m
      memory:  64Mi
...
```

Pour résumer, lancer un ou plusieurs conteneurs (ensemble) dans Kubernetes est simple. Cependant, le faire directement comme indiqué ci-dessus est assorti d'une grave limitation : vous devez veiller à ce qu'il soit maintenu manuellement en cas d'échec. Une meilleure façon de superviser les pods consiste à utiliser des **contrôleurs de réplication (replication controllers)**, voire des déploiements plus performants, pour un contrôle bien plus important.

2) Les labels dans Kubernetes

Les étiquettes (labels) sont le mécanisme que vous utilisez pour organiser les objets Kubernetes. Une étiquette est une paire clé-valeur avec certaines restrictions concernant la longueur et les valeurs autorisées, mais sans signification prédéfinie. Vous êtes donc libre de choisir les étiquettes qui vous conviennent, par exemple pour exprimer des environnements tels que "ce pod est en cours de production" ou que vous en êtes le propriétaire, comme "le département X est propriétaire de ce pod".

Créons un pod qui a initialement un label (**env = development**) :

sudo kubectl create -f pod.yaml

sudo kubectl get pods --show-labels

```
NAME     READY    STATUS   RESTARTS  AGE    LABELS
labelex  1/1      Running  0         10m    env=development
```

Dans la commande get pods ci-dessus, notez l'option **--show-labels** qui affiche les étiquettes d'un objet dans une colonne supplémentaire.

Vous pouvez ajouter une étiquette au pod en tant que:

sudo kubectl label pods labelex owner=michael

sudo kubectl get pods --show-labels

```
NAME      READY   STATUS   RESTARTS   AGE   LABELS
labelex   1/1     Running  0          16m   env=development,owner=michael
```

Pour utiliser une étiquette pour le filtrage, par exemple pour répertorier uniquement les pods dont le propriétaire est égal à michael, utilisez l'option **–selector** :

sudo kubectl get pods --selector owner=michael

```
NAME     READY   STATUS   RESTARTS   AGE
labelex  1/1     Running  0          27m
```

L'option **--selector** peut être abrégée en **-l**, aussi, pour sélectionner les pods étiquetés avec **env = development**, procédez comme suit :

sudo kubectl get pods -l env=development

```
NAME     READY   STATUS   RESTARTS   AGE
labelex  1/1     Running  0          27m
```

Souvent, les objets Kubernetes prennent également en charge les sélecteurs basés sur des ensembles (set-based selectors). Lançons un autre pod avec deux labels (env = production et propriétaire = michael) :

sudo kubectl create -f anotherpod.yaml

À présent, listons tous les pods étiquetés avec **env=development** ou **env=production** :

sudo kubectl get pods -l 'env in (production, development)'

```
NAME         READY   STATUS   RESTARTS   AGE
labelex      1/1     Running  0          43m
labelexother 1/1     Running  0          3m
```

Notez que les étiquettes ne sont pas limitées aux pods. En fait, vous pouvez les appliquer à toutes sortes d'objets, tels que des nœuds ou des services.

3) Les replication controllers (RC) dans Kubernetes

Un **contrôleur de réplication (replication controller - RC)** est un superviseur pour les pods de longue durée. Un RC lancera un nombre spécifié de pods appelés réplicas et s'assurera qu'ils continuent à s'exécuter, par exemple lorsqu'un nœud tombe en panne ou que quelque chose se trouve à l'intérieur d'un pod, c'est-à-dire que l'un de ses conteneurs ne fonctionne pas correctement.

Créons un RC qui supervise une seule réplique (replica) d’un pod:

sudo kubectl create -f rc.yaml

Vous pouvez voir le RC et le pod dont il s'agit :

sudo kubectl get rc
```
NAME          DESIRED   CURRENT   READY     AGE
rcex          1         1         1         3m
```

sudo kubectl get pods --show-labels
```
NAME          READY     STATUS    RESTARTS   AGE    LABELS
rcex-qrv8j    1/1       Running   0          4m     app=sise
```

Notez deux choses ici :

- Le pod supervisé a reçu un nom aléatoire (**rcex-qrv8j**)
- La manière dont la RC garde la trace de ses pods se fait via l'étiquette, ici **app=sise**

Pour augmenter le nombre de répliques, procédez comme suit :

sudo kubectl scale --replicas=3 rc/rcex

sudo kubectl get pods -l app=sise
```
NAME          READY     STATUS    RESTARTS   AGE
rcex-1rh9r    1/1       Running   0          54s
rcex-lv6xv    1/1       Running   0          54s
rcex-qrv8j    1/1       Running   0          10m
```

Enfin, pour vous débarrasser du RC et des pods qu'elle supervise, utilisez :

sudo kubectl delete rc rcex
```
replicationcontroller "rcex" deleted
```

Notez que, par la suite, les contrôleurs RC sont appelés **replica sets (RS)**, prenant en charge set-based selectors. Les RS sont déjà utilisés dans le cadre de déploiements.

4) Les déploiements (deployments) dans Kubernetes

Un déploiement est un superviseur pour les **pods** et les **replica sets,** vous permettant de contrôler avec précision comment et quand une nouvelle version de pod est déployée et restaurée à un état antérieur.

Créons un déploiement appelé **sise-deploy** qui supervise deux réplicas d'un pod ainsi qu'un replica set :

sudo kubectl create -f d09.yaml

Vous pouvez voir le déploiement, le replica set et les pods comme il se présente :

sudo kubectl get deploy

```
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
sise-deploy  2         2         2            2           10s
```

sudo kubectl get rs

```
NAME                    DESIRED   CURRENT   READY     AGE
sise-deploy-3513442901  2         2         2         19s
```

sudo kubectl get pods

```
NAME                          READY     STATUS    RESTARTS   AGE
sise-deploy-3513442901-cndsx  1/1       Running   0          25s
sise-deploy-3513442901-sn74v  1/1       Running   0          25s
```

Notez la dénomination des pods et des replica set, dérivés du nom du déploiement.

À ce stade, les conteneurs Sise s'exécutant dans les pods sont configurés pour renvoyer la version 0.9. Vérifions cela depuis le cluster (en utilisant **kubectl describe** d'abord pour obtenir l'adresse IP de l'un des pods) :

sudo curl 172.17.0.3:9876/info

```
{"host": "172.17.0.3:9876", "version": "0.9", "from": "172.17.0.1"}
```

Voyons maintenant ce qui se passe si nous changeons cette version en 1.0 dans un déploiement de mise à jour:

sudo kubectl apply -f d10.yaml

```
deployment "sise-deploy" configured
```

Notez que vous auriez pu utiliser **kubectl edit deploy/sise-deploy** alternativement pour atteindre le même objectif en modifiant manuellement le déploiement.

Nous voyons maintenant le lancement de deux nouveaux pods avec la version 1.0 mise à jour, ainsi que les deux anciens pods avec la version 0.9 en cours de terminaison :

sudo kubectl get pods

```
NAME                          READY     STATUS        RESTARTS   AGE
sise-deploy-2958877261-nfv28  1/1       Running       0          25s
sise-deploy-2958877261-w024b  1/1       Running       0          25s
sise-deploy-3513442901-cndsx  1/1       Terminating   0          16m
sise-deploy-3513442901-sn74v  1/1       Terminating   0          16m
```

De plus, un nouveau replica set a été créé par le déploiement :

sudo kubectl get rs

```
NAME                    DESIRED   CURRENT   READY     AGE
sise-deploy-2958877261  2         2         2         4s
sise-deploy-3513442901  0         0         0         24m
```

Notez que pendant le déploiement, vous pouvez vérifier la progression à l'aide de **kubectl rollout status deploy/sise-deploy.**

Pour vérifier que si la nouvelle version 1.0 est vraiment disponible, nous exécutons à partir du cluster (utilisez à nouveau **kubectl describe** afin d'obtenir l'adresse IP de l'un des pods):

sudo curl 172.17.0.5:9876/info
{"host": "172.17.0.5:9876", "version": "1.0", "from": "172.17.0.1"}

Un historique de tous les déploiements est disponible via :

sudo kubectl rollout history deploy/sise-deploy
deployments "sise-deploy"
REVISION CHANGE-CAUSE
1 <none>
2 <none>

S'il y a des problèmes dans le déploiement, Kubernetes rétablit automatiquement la version précédente, mais vous pouvez également explicitement revenir à une révision spécifique, comme dans notre cas à la révision 1 (la version du pod d'origine) :

sudo kubectl rollout undo deploy/sise-deploy --to-revision=1
deployment "sise-deploy" rolled back

sudo kubectl rollout history deploy/sise-deploy
deployments "sise-deploy"
REVISION CHANGE-CAUSE
2 <none>
3 <none>

sudo kubectl get pods
NAME READY STATUS RESTARTS AGE
sise-deploy-3513442901-ng8fz 1/1 Running 0 1m
sise-deploy-3513442901-s8q4s 1/1 Running 0 1m

À ce stade, nous sommes revenus à l'origine, avec deux nouveaux pods servant à nouveau la version 0.9.

Enfin, pour nettoyer, nous supprimons le déploiement et, avec lui, les replica sets et les pods qu'il supervise :

sudo kubectl delete deploy sise-deploy
deployment "sise-deploy" deleted

5) Les services dans Kubernetes

Un service est une abstraction pour les pods, fournissant une adresse IP virtuelle et stable (VIP). Même si les pods vont et viennent, les services permettent aux clients de se connecter de manière fiable aux conteneurs s'exécutant dans les pods, à l'aide du **VIP (Virtual Internet Protocol)**. Virtual dans VIP signifie que ce n'est pas une adresse IP réelle connectée à une interface réseau, mais que son but est uniquement de transférer le trafic vers un ou plusieurs pods. Garder la correspondance entre le VIP et les pods à jour est le travail de **kube-proxy**. Ce dernier représente un processus qui s'exécute sur chaque nœud et interroge le serveur d'API pour en savoir plus sur les nouveaux services du cluster.

Créons un pod supervisé par un RC et un service :

sudo kubectl create -f rc.yaml

sudo kubectl create -f svc.yaml

Maintenant, nous avons le pod supervisé en cours d'exécution :

sudo kubectl get pods -l app=sise
```
NAME          READY     STATUS    RESTARTS   AGE
rcsise-6nq3k  1/1       Running   0          57s
```

sudo kubectl describe pod rcsise-6nq3k
```
Name:               rcsise-6nq3k
Namespace:          default
Security Policy:    restricted
Node:               localhost/192.168.99.100
Start Time:         Tue, 25 Apr 2017 14:47:45 +0100
Labels:             app=sise
Status:             Running
IP:                 172.17.0.3
Controllers:        ReplicationController/rcsise
Containers:
...
```

Depuis le cluster, vous pouvez accéder directement au pod via l'IP 172.17.0.3 qui lui est attribuée :

sudo curl 172.17.0.3:9876/info
```
{"host": "172.17.0.3:9876", "version": "0.5.0", "from": "172.17.0.1"}
```

Comme indiqué ci-dessus, cela n'est toutefois pas conseillé, car les adresses IP attribuées aux pods peuvent changer. Par conséquent, entrez le **simpleservice** que nous avons créé :

sudo kubectl get svc
```
NAME          CLUSTER-IP      EXTERNAL-IP   PORT(S)             AGE
simpleservice 172.30.228.255  <none>        80/TCP              5m
```

sudo kubectl describe svc simpleservice

Name: simpleservice
Namespace: default
Labels: <none>
Selector: app=sise
Type: ClusterIP
IP: 172.30.228.255
Port: <unset> 80/TCP
Endpoints: 172.17.0.3:9876
Session Affinity: None
No events.
Le service garde la trace des pods auxquels il achemine le trafic via l'étiquette, dans notre cas, **app=sise.**

Depuis le cluster, vous pouvez maintenant accéder à **simpleservice** comme ceci :

sudo curl 172.30.228.255:80/info
{"host": "172.30.228.255", "version": "0.5.0", "from": "10.0.2.15"}

Qu'est-ce qui fait que le VIP 172.30.228.255 transfère le trafic vers le pod? La réponse est : **IPtables**, qui est essentiellement une longue liste de règles qui indique au noyau Linux quoi faire avec un certain paquet IP.

En regardant les règles qui concernent notre service (exécuté sur un nœud de cluster), vous obtenez :

[cluster] $ **sudo iptables-save | grep simpleservice**
-A KUBE-SEP-4SQFZS32ZVMTQEZV -s 172.17.0.3/32 -m comment --comment "default/simpleservice:" -j KUBE-MARK-MASQ

-A KUBE-SEP-4SQFZS32ZVMTQEZV -p tcp -m comment --comment "default/simpleservice:" -m tcp -j DNAT --to-destination 172.17.0.3:9876

-A KUBE-SERVICES -d 172.30.228.255/32 -p tcp -m comment --comment "default/simpleservice: cluster IP" -m tcp --dport 80 -j KUBE-SVC-EZC6WLOVQADP4IAW

-A KUBE-SVC-EZC6WLOVQADP4IAW -m comment --comment "default/simpleservice:" -j KUBE-SEP-4SQFZS32ZVMTQEZV

Ci-dessus, vous pouvez voir les quatre règles que **kube-proxy** a heureusement ajoutées à la table de routage, indiquant essentiellement que le trafic TCP vers **172.30.228.255:80** doit être transmis à **172.17.0.3:9876**, qui est notre pod.

Ajoutons maintenant un deuxième pod avec une mise à l'échelle (scaling up) du RC qui le supervise :

sudo kubectl scale --replicas=2 rc/rcsise

replicationcontroller "rcsise" scaled

sudo kubectl get pods -l app=sise

```
NAME          READY    STATUS    RESTARTS   AGE
rcsise-6nq3k  1/1      Running   0          15m
rcsise-nv8zm  1/1      Running   0          5s
```

Lorsque nous vérifions à nouveau les parties pertinentes de la table de routage, nous avons constaté l'ajout de nombreuses **règles Iptables** :

[cluster] $ **sudo iptables-save | grep simpleservice**

-A KUBE-SEP-4SQFZS32ZVMTQEZV -s 172.17.0.3/32 -m comment --comment "default/simpleservice:" -j KUBE-MARK-MASQ

-A KUBE-SEP-4SQFZS32ZVMTQEZV -p tcp -m comment --comment "default/simpleservice:" -m tcp -j DNAT --to-destination 172.17.0.3:9876

-A KUBE-SEP-PXYYII6AHMUWKLYX -s 172.17.0.4/32 -m comment --comment "default/simpleservice:" -j KUBE-MARK-MASQ

-A KUBE-SEP-PXYYII6AHMUWKLYX -p tcp -m comment --comment "default/simpleservice:" -m tcp -j DNAT --to-destination 172.17.0.4:9876

-A KUBE-SERVICES -d 172.30.228.255/32 -p tcp -m comment --comment "default/simpleservice: cluster IP" -m tcp --dport 80 -j KUBE-SVC-EZC6WLOVQADP4IAW

-A KUBE-SVC-EZC6WLOVQADP4IAW -m comment --comment "default/simpleservice:" -m statistic --mode random --probability 0.50000000000 -j KUBE-SEP-4SQFZS32ZVMTQEZV

-A KUBE-SVC-EZC6WLOVQADP4IAW -m comment --comment "default/simpleservice:" -j KUBE-SEP-PXYYII6AHMUWKLYX

Dans la liste de table de routage ci-dessus, nous voyons les règles pour le pod nouvellement créé servant à 172.17.0.4:9876, ainsi qu'une règle supplémentaire :

-A KUBE-SVC-EZC6WLOVQADP4IAW -m comment --comment "default/simpleservice:" -m statistic --mode random --probability 0.50000000000 -j KUBE-SEP-4SQFZS32ZVMTQEZV

Cela provoque la répartition égale du trafic vers le service entre nos deux pods en appelant le module de statistiques d'**Iptables**.

6) Le service discovery dans Kubernetes

La découverte de service (Service discovery) consiste à déterminer comment se connecter à un service. Bien qu'il existe une option de découverte de service basée sur les variables

d'environnement disponibles, la découverte de service basée sur le DNS est préférable. Notez que le DNS est un plugin de cluster, assurez-vous que votre distribution Kubernetes en fournit un ou installez-la vous-même.

Créons un service nommé **thesvc** et un **replication controller** supervisant certains pods :

sudo kubectl create -f rc.yaml

sudo kubectl create -f svc.yaml

Nous voulons maintenant nous connecter au **service thesvc** à partir du cluster, par exemple, à partir d'un autre service. Pour simuler cela, nous créons un jump pod dans le même espace de nom (par défaut - default, puisque nous n'avons rien spécifié d'autre) :

sudo kubectl create -f jumpod.yaml

Le module complémentaire DNS s'assurera que notre service **thesvc** est disponible via le nom de domaine complet (FQDN) **thesvc.default.svc.cluster.local** à partir d'autres pods du cluster. Essayons-le :

sudo kubectl exec jumpod -c shell -i -t -- ping thesvc.default.svc.cluster.local
PING thesvc.reshifter.svc.cluster.local (172.30.251.137) 56(84) bytes of data.
...

La réponse au ping nous indique que le service est disponible via le cluster IP 172.30.251.137. Nous pouvons directement nous connecter et utiliser le service (dans le même espace de noms) comme ceci :

sudo kubectl exec jumpod -c shell -i -t -- curl http://thesvc/info
{"host": "thesvc", "version": "0.5.0", "from": "172.17.0.5"}

Notez que l'adresse IP 172.17.0.5 ci-dessus est l'adresse IP interne au cluster du jump pod.

Pour accéder à un service déployé dans un espace de noms différent de celui auquel vous accédez, utilisez un nom de domaine complet au format **$ SVC. $ NAMESPACE.svc.cluster.local.**

Voyons comment cela fonctionne en créant :

1. un espace de noms **autre.**

2. un service **thesvc** dans l'espace de noms **autre.**

3. un RC supervisant les pods, également dans l'espace de noms **autre.**

sudo kubectl create -f other-ns.yaml

sudo kubectl create -f other-rc.yaml

sudo kubectl create -f other-svc.yaml

Nous sommes maintenant en mesure de consommer le service **thesvc** dans l'espace de nom **autre** de l'espace de nom default (à nouveau via le jump pod):

sudo kubectl exec jumpod -c shell -i -t -- curl http://thesvc.other/info
{"host": "thesvc.other", "version": "0.5.0", "from": "172.17.0.5"}

En résumé, la découverte de services basée sur le DNS fournit un moyen flexible et générique de se connecter aux services du cluster.

7) Les Heath checks dans Kubernetes

Afin de vérifier si un conteneur dans un pod est en bon état et prêt à gérer le trafic, Kubernetes propose une gamme de mécanismes de **health check**. Les Heath check, ou sondes (probes) dans Kubernetes, sont effectuées par le kubelet pour déterminer quand redémarrer un conteneur (pour **livenessProbe**) et par les services pour déterminer si un pod doit recevoir du trafic ou non (pour **readinessProbe**).

Nous allons nous concentrer sur les health check HTTP dans ce qui suit. Notez qu'il incombe au développeur de l'application de révéler (expose) une URL que le kubelet peut utiliser pour déterminer si le conteneur est sain (et potentiellement prêt).

Créons un pod qui expose un point de terminaison /health et répond avec un code d'état HTTP 200 :

sudo kubectl create -f pod.yaml

Dans la spécification du pod, nous avons défini ce qui suit:

```
livenessProbe:
  initialDelaySeconds: 2
  periodSeconds: 5
  httpGet:
    path: /health
    port: 9876
```

Cette spécification signifie que Kubernetes va commencer la vérification /le point de terminaison health toutes les 5 secondes après avoir attendu 2 secondes pour le premier contrôle.

Si nous regardons maintenant le pod, nous pouvons voir qu'il est considéré comme sain (**healthy**) :

sudo kubectl describe pod hc

```
Name:                   hc
Namespace:              default
Security Policy:        anyuid
Node:                   192.168.99.100/192.168.99.100
Start Time:             Tue, 25 Apr 2017 16:21:11 +0100
Labels:                 <none>
Status:                 Running
...
Events:
  FirstSeen     LastSeen        Count   From                    SubobjectPath           Type
Reason          Message
  ---------     --------        -----   ----                    -------------           --------        ------          ----
---
  3s            3s              1       {default-scheduler }                            Normal
Scheduled       Successfully assigned hc to 192.168.99.100
  3s            3s              1       {kubelet 192.168.99.100}        spec.containers{sise}   Normal
Pulled          Container image "mhausenblas/simpleservice:0.5.0"
already present on machine
  3s            3s              1       {kubelet 192.168.99.100}        spec.containers{sise}   Normal
Created         Created container with docker id 8a628578d6ad;
Security:[seccomp=unconfined]
  2s            2s              1       {kubelet 192.168.99.100}        spec.containers{sise}   Normal
Started         Started container with docker id 8a628578d6ad
```

Nous lançons maintenant un pod incorrect, c'est-à-dire un pod dont le conteneur ne retourne pas un code 200 (dans un temps de 1 à 4 secondes) :

sudo kubectl create -f badpod.yaml

Looking at the events of the bad pod, we can see that the failed:

En regardant les événements du mauvais pod, nous pouvons voir que le Heath check a échoué :

sudo kubectl describe pod badpod

```
...
Events:
  FirstSeen     LastSeen        Count   From                    SubobjectPath           Type
Reason          Message
  ---------     --------        -----   ----                    -------------           --------        ------          ----
---
  1m            1m              1       {default-scheduler }                            Normal
Scheduled       Successfully assigned badpod to 192.168.99.100
  1m            1m              1       {kubelet 192.168.99.100}        spec.containers{sise}   Normal
Created         Created container with docker id 7dd660f04945;
Security:[seccomp=unconfined]
```

```
  1m        1m        1       {kubelet 192.168.99.100}        spec.containers{sise}   Normal
Started         Started container with docker id 7dd660f04945
  1m        23s       2       {kubelet 192.168.99.100}        spec.containers{sise}   Normal
Pulled          Container image "mhausenblas/simpleservice:0.5.0" already present on machine
  23s       23s       1       {kubelet 192.168.99.100}        spec.containers{sise}   Normal
Killing         Killing container with docker id 7dd660f04945: pod "badpod_default(53e5c06a-
29cb-11e7-b44f-be3e8f4350ff)" container "sise" is unhealthy, it will be killed and re-created.
  23s       23s       1       {kubelet 192.168.99.100}        spec.containers{sise}   Normal
Created         Created container with docker id ec63dc3edfaa; Security:[seccomp=unconfined]
  23s       23s       1       {kubelet 192.168.99.100}        spec.containers{sise}   Normal
Started         Started container with docker id ec63dc3edfaa
  1m        18s       4       {kubelet 192.168.99.100}        spec.containers{sise}   Warning
Unhealthy       Liveness probe failed: Get http://172.17.0.4:9876/health: net/http: request
canceled (Client.Timeout exceeded while awaiting headers)
```

Ceci peut également être vérifié comme suit :

sudo kubectl get pods

```
NAME                READY     STATUS    RESTARTS   AGE
badpod              1/1       Running   4          2m
hc                  1/1       Running   0          6m
```

Avec l'affichage ci-dessus, vous pouvez voir que le **badpod** a déjà été relancé 4 fois, car le health check a échoué.

En plus de **livenessProbe**, vous pouvez également spécifier un **readinessProbe**, qui peut être configuré de la même manière mais avec un cas d'utilisation et une sémantique différents : il est utilisé pour vérifier la phase de démarrage d'un conteneur dans le pod. Imaginez un conteneur qui charge des données à partir d'un stockage externe, tel que S3, ou une base de données devant initialiser certaines tables. Dans ce cas, vous souhaitez signaler quand le conteneur est prêt à servir le trafic.

Créons un pod avec un ReadinessProbe qui démarre après 10 secondes :

sudo kubectl create -f ready.yaml

En regardant les événements du pod, nous pouvons voir que, finalement, le pod est prêt à servir le trafic :

sudo kubectl describe pod ready

```
...
Conditions:
[0/1888]
  Type          Status
  Initialized   True
  Ready         True
  PodScheduled  True
```

...

8) Les variables d'environnement dans Kubernetes

Vous pouvez définir des variables d'environnement pour les conteneurs s'exécutant dans un pod. En outre, Kubernetes expose automatiquement certaines informations d'exécution via des variables d'environnement.

Lançons un pod qui passe une variable d'environnement SIMPLE_SERVICE_VERSION avec la valeur 1.0 :

sudo kubectl create -f pod.yaml

sudo kubectl describe pod envs | grep IP
IP: 172.17.0.3

Maintenant, vérifions depuis le cluster si l'application qui s'exécute dans le pod a bien récupéré la variable d'environnement **SIMPLE_SERVICE_VERSION** :

sudo curl 172.17.0.3:9876/info
{"host": "172.17.0.3:9876", "version": "1.0", "from": "172.17.0.1"}

Et en effet, il a pris la variable d'environnement fournie par l'utilisateur (la réponse par défaut serait "version": "0.5.0").

Vous pouvez vérifier quelles variables d'environnement Kubernetes lui-même fournit automatiquement (à partir du cluster, à l'aide d'un nœud final dédié exposé par l'application) :

sudo curl 172.17.0.3:9876/env
{"version": "1.0", "env": "{'HOSTNAME': 'envs',
'DOCKER_REGISTRY_SERVICE_PORT': '5000',
'KUBERNETES_PORT_443_TCP_ADDR': '172.30.0.1',
'ROUTER_PORT_80_TCP_PROTO': 'tcp', 'KUBERNETES_PORT_53_UDP_PROTO':
'udp', 'ROUTER_SERVICE_HOST': '172.30.246.127',
'ROUTER_PORT_1936_TCP_PROTO': 'tcp', 'KUBERNETES_SERVICE_PORT_DNS':
'53', 'DOCKER_REGISTRY_PORT_5000_TCP_PORT': '5000', 'PATH':
'/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin',
'ROUTER_SERVICE_PORT_443_TCP': '443', 'KUBERNETES_PORT_53_TCP':
'tcp://172.30.0.1:53', 'KUBERNETES_SERVICE_PORT': '443',
'ROUTER_PORT_80_TCP_ADDR': '172.30.246.127', 'LANG': 'C.UTF-8',
'KUBERNETES_PORT_53_TCP_ADDR': '172.30.0.1', 'PYTHON_VERSION': '2.7.13',
'KUBERNETES_SERVICE_HOST': '172.30.0.1', 'PYTHON_PIP_VERSION': '9.0.1',
'DOCKER_REGISTRY_PORT_5000_TCP_PROTO': 'tcp', 'REFRESHED_AT': '2017-04-
24T13:50', 'ROUTER_PORT_1936_TCP': 'tcp://172.30.246.127:1936',
'KUBERNETES_PORT_53_TCP_PROTO': 'tcp', 'KUBERNETES_PORT_53_TCP_PORT':
'53', 'HOME': '/root', 'DOCKER_REGISTRY_SERVICE_HOST': '172.30.1.1', 'GPG_KEY':
'C01E1CAD5EA2C4F0B8E3571504C367C218ADD4FF',
'ROUTER_SERVICE_PORT_80_TCP': '80', 'ROUTER_PORT_443_TCP_ADDR':

'172.30.246.127', 'ROUTER_PORT_1936_TCP_ADDR': '172.30.246.127',
'ROUTER_SERVICE_PORT': '80', 'ROUTER_PORT_443_TCP_PORT': '443',
'KUBERNETES_SERVICE_PORT_DNS_TCP': '53',
'KUBERNETES_PORT_53_UDP_ADDR': '172.30.0.1', 'KUBERNETES_PORT_53_UDP':
'udp://172.30.0.1:53', 'KUBERNETES_PORT': 'tcp://172.30.0.1:443',
'ROUTER_PORT_1936_TCP_PORT': '1936', 'ROUTER_PORT_80_TCP':
'tcp://172.30.246.127:80', 'KUBERNETES_SERVICE_PORT_HTTPS': '443',
'KUBERNETES_PORT_53_UDP_PORT': '53', 'ROUTER_PORT_80_TCP_PORT': '80',
'ROUTER_PORT': 'tcp://172.30.246.127:80', 'ROUTER_PORT_443_TCP':
'tcp://172.30.246.127:443', 'SIMPLE_SERVICE_VERSION': '1.0',
'ROUTER_PORT_443_TCP_PROTO': 'tcp', 'KUBERNETES_PORT_443_TCP':
'tcp://172.30.0.1:443', 'DOCKER_REGISTRY_PORT_5000_TCP': 'tcp://172.30.1.1:5000',
'DOCKER_REGISTRY_PORT': 'tcp://172.30.1.1:5000',
'KUBERNETES_PORT_443_TCP_PORT': '443', 'ROUTER_SERVICE_PORT_1936_TCP':
'1936', 'DOCKER_REGISTRY_PORT_5000_TCP_ADDR': '172.30.1.1',
'DOCKER_REGISTRY_SERVICE_PORT_5000_TCP': '5000',
'KUBERNETES_PORT_443_TCP_PROTO': 'tcp'}"}

Vous pouvez également utiliser **kubectl exec** pour vous connecter au conteneur et lister directement les variables d'environnement, ici:

sudo kubectl exec envs -- printenv
PATH=/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=envs
SIMPLE_SERVICE_VERSION=1.0
KUBERNETES_PORT_53_UDP_ADDR=172.30.0.1
KUBERNETES_PORT_53_TCP_PORT=53
ROUTER_PORT_443_TCP_PROTO=tcp
DOCKER_REGISTRY_PORT_5000_TCP_ADDR=172.30.1.1
KUBERNETES_SERVICE_PORT_DNS_TCP=53
ROUTER_PORT=tcp://172.30.246.127:80
…

9) Les espaces de noms (namespaces) dans Kubernetes

Les **espaces de noms** fournissent une étendue d'objets Kubernetes. Vous pouvez le considérer comme un espace de travail que vous partagez avec d'autres utilisateurs. De nombreux objets tels que les pods et les services sont nommés, tandis que d'autres (comme les nœuds) ne le sont pas. En tant que développeur, vous utilisez généralement simplement un espace de noms attribué. Toutefois, les administrateurs peuvent souhaiter les gérer, par exemple pour configurer un contrôle d'accès ou des quotas de ressources.

Répertorions tous les espaces de noms (notez que la sortie dépend de l'environnement que vous utilisez, ici Minishift) :

sudo kubectl get ns
NAME STATUS AGE

```
default          Active    13d
kube-system      Active    13d
namingthings     Active    12d
openshift        Active    13d
openshift-infra  Active    13d
```

Vous pouvez en apprendre plus sur un espace de noms en utilisant le verbe **describe**, par exemple :

sudo kubectl describe ns default

```
Name:   default
Labels: <none>
Status: Active

No resource quota.

No resource limits.
```

Créons maintenant un nouvel espace de noms appelé **test** :

sudo kubectl create -f ns.yaml

```
namespace "test" created
```

sudo kubectl get ns

```
NAME             STATUS    AGE
default          Active    13d
kube-system      Active    13d
namingthings     Active    12d
openshift        Active    13d
openshift-infra  Active    13d
test             Active    3s
```

Pour lancer un pod dans l'espace de nom **test** nouvellement créé, procédez comme suit :

sudo kubectl create --namespace=test -f pod.yaml

Notez qu'en utilisant la méthode ci-dessus, l'espace de noms devient une propriété d'exécution, c'est-à-dire que vous pouvez facilement déployer le même pod ou service, ou le même replication contrôleur, etc. dans plusieurs espaces de noms (par exemple : **dev** et **prod**). Si vous préférez cependant coder en dur l'espace de noms, vous pouvez le définir directement dans les métadonnées comme suit :

```
apiVersion: v1
kind: Pod
metadata:
  name: podintest
  namespace: test
```

Pour répertorier les objets nommés tels que notre pod **podintest**, exécutez la commande suivante en tant que :

sudo kubectl get pods --namespace=test
```
NAME       READY    STATUS    RESTARTS   AGE
podintest  1/1      Running   0          16s
```

Si vous êtes un administrateur, vous pouvez consulter la documentation pour plus d'informations sur la gestion des espaces de noms.

10) Les volumes dans Kubernetes

Un volume Kubernetes est essentiellement un répertoire accessible à tous les conteneurs exécutés dans un pod. Contrairement au système de fichiers local au conteneur, les données des volumes sont conservées lors des redémarrages du conteneur. Le support de sauvegarde d'un volume et son contenu sont déterminés par le type de volume :

- Types de nœuds locaux tels que **emptyDir** ou **hostPath**
- Types de partage de fichiers tels que **nfs**
- Types spécifiques au fournisseur de cloud, tels que **awsElasticBlockStore, azureDisk** ou **gcePersistentDisk**
- Types de système de fichiers distribués, par exemple **glusterfs** ou **cephfs**
- Types spéciaux comme **secret**, **gitRepo**

Un type spécial de volume est **PersistentVolume**, que nous couvrirons ailleurs.

Créons un pod avec deux conteneurs qui utilisent un volume **emptyDir** pour échanger des données :

sudo kubectl create -f pod.yaml

sudo kubectl describe pod sharevol
```
Name:              sharevol
Namespace:         default
...
Volumes:
  xchange:
    Type:      EmptyDir (a temporary directory that shares a pod's lifetime)
    Medium:
```

Nous faisons d'abord l'exécution dans l'un des conteneurs du pod, **c1**, vérifions le montage du volume et générons des données :

sudo kubectl exec sharevol -c c1 -i -t -- bash
```
[root@sharevol /]# mount | grep xchange
/dev/sda1 on /tmp/xchange type ext4 (rw,relatime,data=ordered)
[root@sharevol /]# echo 'some data' > /tmp/xchange/data
```

Lorsque nous exécutons maintenant dans **C2**, le deuxième conteneur exécuté dans le pod, nous pouvons voir le volume monté dans **/tmp/data** et être en mesure de lire les données créées à l'étape précédente :

sudo kubectl exec sharevol -c c2 -i -t -- bash
[root@sharevol /]# **mount | grep /tmp/data**
/dev/sda1 on /tmp/data type ext4 (rw,relatime,data=ordered)
[root@sharevol /]# **cat /tmp/data/data**
some data

Notez que dans chaque conteneur, vous devez décider où monter le volume et que, pour **emptyDir**, vous ne pouvez pas spécifier de limite de consommation des ressources.

11) Les secrets dans Kubernetes

Vous ne souhaitez pas que des informations sensibles telles qu'un mot de passe de base de données ou une clé d'API soient conservées en texte clair. Les secrets vous fournissent un mécanisme pour utiliser ces informations de manière sûre et fiable avec les propriétés suivantes :

- Les secrets sont des objets d'espace de noms, c'est-à-dire qu'ils existent dans le contexte d'un espace de noms.
- Vous pouvez y accéder via un volume ou une variable d'environnement à partir d'un conteneur s'exécutant dans un pod.
- Les données secrètes sur les nœuds sont stockées dans des volumes tmpfs
- Il existe une limite de taille par secret de 1 Mo
- Le serveur d'API stocke les secrets en texte brut dans etcd

Créons un **apikey secret** contenant une clé d'API :

sudo echo -n "A19fh68B001j" > ./apikey.txt

sudo kubectl create secret generic apikey --from-file=./apikey.txt
secret "apikey" created

sudo kubectl describe secrets/apikey
Name: apikey
Namespace: default
Labels: <none>
Annotations: <none>

Type: Opaque

Data
====
apikey.txt: 12 bytes

Utilisons maintenant le secret dans un pod via un volume :

sudo kubectl create -f pod.yaml

Si nous exécutons maintenant dans le conteneur, nous voyons le secret monté dans **/tmp/apikey**:

sudo kubectl exec consumesec -c shell -i -t -- bash
[root@consumesec /]# **mount | grep apikey**
tmpfs on /tmp/apikey type tmpfs (ro,relatime)
[root@consumesec /]# **cat /tmp/apikey/apikey.txt**
A19fh68B001j

Notez que pour les comptes de service, Kubernetes crée automatiquement des secrets contenant des informations d'identification pour accéder à l'API et modifie vos pods pour utiliser ce type de secret.

12) La journalisation (logging) dans Kubernetes

La **journalisation (logging)** est une option pour comprendre ce qui se passe dans vos applications et dans le cluster en général. La journalisation de base dans Kubernetes rend disponible la sortie produite par un conteneur, ce qui constitue un cas d'utilisation judicieux pour le débogage. Les configurations plus avancées prennent en compte les journaux (logs) sur les nœuds et les stockent dans un emplacement central, au sein du cluster ou via un service dédié (basé sur le cloud).

Créons un pod appelé **logme** qui exécute un conteneur en écrivant sur **stdout** et **stderr** :

sudo kubectl create -f pod.yaml

Pour afficher les cinq dernières lignes de journal du conteneur gen dans le pod Logme, exécutez la procédure suivante :

sudo kubectl logs --tail=5 logme -c gen
Mon Oct 29 15:48:06 UTC 2018
Mon Oct 29 15:48:07 UTC 2018
Mon Oct 29 15:48:07 UTC 2018
Mon Oct 29 15:48:08 UTC 2018
Mon Oct 29 15:48:08 UTC 2018

Pour diffuser le journal (log) du conteneur **gen** dans le pod logme (comme **tail -f**), procédez comme suit:

sudo kubectl logs -f --since=10s logme -c gen
 15:47:28 UTC 2018
Mon Oct 29 15:47:28 UTC 2018
Mon Oct 29 15:47:29 UTC 2018

Mon Oct 29 15:47:29 UTC 2018
Mon Oct 29 15:47:30 UTC 2018
Mon Oct 29 15:47:30 UTC 2018
Mon Oct 29 15:47:31 UTC 2018
Mon Oct 29 15:47:31 UTC 2018
Mon Oct 29 15:47:32 UTC 2018
Mon Oct 29 15:47:32 UTC 2018
Mon Oct 29 15:47:33 UTC 2018
Mon Oct 29 15:47:33 UTC 2018
Mon Oct 29 15:47:34 UTC 2018
...

Notez que si vous n'aviez pas spécifié **--since = 10s** dans la commande ci-dessus, vous auriez obtenu toutes les lignes de journal à partir du début du conteneur.

Vous pouvez également afficher les journaux des pods ayant déjà terminé leur cycle de vie. Pour cela, nous créons un pod appelé **onehot** avec un compte à rebours de 9 à 1 puis se termine. À l'aide de l'option **-p,** vous pouvez imprimer les logs des instances précédentes du conteneur dans un pod :

sudo kubectl create -f oneshotpod.yaml
sudo kubectl logs -p oneshot -c gen
9
8
7
6
5
4
3
2
1

13) Les jobs dans Kubernetes

Un job est un superviseur pour les pods effectuant des processus par lots (batch), c'est-à-dire un processus qui s'exécute pendant un certain temps, par exemple un calcul ou une opération de sauvegarde.

Créons un job appelé **countdown** qui supervise un pod comptant de 9 à 1:

sudo kubectl create -f job.yaml

Vous pouvez voir le job et le pod dont il s'agit :

sudo kubectl get jobs

NAME	DESIRED	SUCCESSFUL	AGE
countdown	1	1	5s

sudo kubectl get pods --show-all

```
NAME                    READY     STATUS      RESTARTS   AGE
countdown-lc80g         0/1       Completed   0          16s
```

Pour en savoir plus sur le statut du job, procédez comme suit :

sudo kubectl describe jobs/countdown

```
Name:           countdown
Namespace:      default
Image(s):       centos:7
Selector:       controller-uid=ff585b92-2b43-11e7-b44f-be3e8f4350ff
Parallelism:    1
Completions:    1
Start Time:     Thu, 27 Apr 2017 13:21:10 +0100
Labels:         controller-uid=ff585b92-2b43-11e7-b44f-be3e8f4350ff
                job-name=countdown
Pods Statuses:  0 Running / 1 Succeeded / 0 Failed
No volumes.
Events:
  FirstSeen     LastSeen        Count   From                SubobjectPath   Type            Reason
Message
  ---------     --------        -----   ----                -------------   --------        ------                  -------
  2m            2m              1       {job-controller }                   Normal          SuccessfulCreate
Created pod:countdown-qfqw2
```

Et pour voir la sortie du job via le pod qu'il a supervisé, exécutez :

sudo kubectl logs countdown-qfqw2

```
9
8
7
6
5
4
3
2
1
```

Pour nettoyer, utilisez le verbe **delete** sur l'objet job qui supprimera tous les pods supervisés :

sudo kubectl delete job countdown

```
job "countdown" deleted
```

Notez qu'il existe également des moyens plus avancés d'utiliser les jobs, par exemple, en utilisant une file d'attente de travail ou en planifiant l'exécution à un moment donné via des cron jobs.

14) Les nœuds (nodes) dans Kubernetes

Dans Kubernetes, les nœuds sont les workers sur lesquels vos pods sont exécutés.

En tant que développeur, vous n'utilisez généralement pas de nœuds. Toutefois, en tant qu'administrateur, vous souhaiterez peut-être vous familiariser avec les opérations des nœuds (nodes).

Pour répertorier les nœuds disponibles dans votre cluster (notez que la sortie dépend de l'environnement que vous utilisez, ici Minishift) :

sudo kubectl get nodes

```
NAME      STATUS   ROLES    AGE   VERSION
minikube  Ready    master   6h    v1.10.0
```

Une tâche intéressante, du point de vue du développeur, consiste à obliger Kubernetes à planifier un pod sur un certain nœud. Pour cela, nous devons d'abord étiqueter (label) le noeud que nous voulons cibler :

sudo kubectl label nodes minikube shouldrun=here

```
node/minikube labeled
```

Nous pouvons maintenant créer un pod planifié sur le nœud avec l'étiquette **shouldrun = here** :

sudo kubectl create -f pod.yaml

sudo kubectl get pods --output=wide

```
NAME              READY    STATUS    RESTARTS   AGE       IP            NODE
onspecificnode    1/1      Running   0          8s        172.17.0.3    192.168.99.100
```

Pour en savoir plus sur un nœud spécifique, 192.168.99.100, procédez comme suit :

sudo kubectl describe node minikube

```
Name:               minikube
Roles:              master
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    kubernetes.io/hostname=minikube
                    node-role.kubernetes.io/master=
                    shouldrun=here
Annotations:        node.alpha.kubernetes.io/ttl: 0
                    volumes.kubernetes.io/controller-managed-attach-detach: true
CreationTimestamp:  Mon, 29 Oct 2018 10:28:35 +0100
Taints:             <none>
Unschedulable:      false
```

```
Conditions :
  Type             Status  LastHeartbeatTime                 LastTransitionTime                Reason
Message
  ----             ------  -----------------                 ------------------                ------
  OutOfDisk        False   Mon, 29 Oct 2018 16:59:17 +0100   Mon, 29 Oct 2018 10:28:31
+0100   KubeletHasSufficientDisk     kubelet has sufficient disk space available
  MemoryPressure   False   Mon, 29 Oct 2018 16:59:17 +0100   Mon, 29 Oct 2018 10:28:31
+0100   KubeletHasSufficientMemory   kubelet has sufficient memory available
  DiskPressure     False   Mon, 29 Oct 2018 16:59:17 +0100   Mon, 29 Oct 2018 10:28:31
+0100   KubeletHasNoDiskPressure     kubelet has no disk pressure
  PIDPressure      False   Mon, 29 Oct 2018 16:59:17 +0100   Mon, 29 Oct 2018 10:28:31
+0100   KubeletHasSufficientPID      kubelet has sufficient PID available
  Ready            True    Mon, 29 Oct 2018 16:59:17 +0100   Mon, 29 Oct 2018 10:28:31 +0100
KubeletReady                 kubelet is posting ready status
Addresses:
  InternalIP:  192.168.122.151
  Hostname:    minikube
Capacity:
 cpu:                2
 ephemeral-storage:  16058792Ki
 hugepages-2Mi:      0
 memory:             1942928Ki
 pods:               110
Allocatable:
 cpu:                2
 ephemeral-storage:  14799782683
 hugepages-2Mi:      0
 memory:             1840528Ki
 pods:               110
System Info:
 Machine ID:                 8a350ebb7c0541a3837b704f7094a0b3
 System UUID:                8A350EBB-7C05-41A3-837B-704F7094A0B3
 Boot ID:                    dedae1c9-056f-4d21-a087-1e49c309192a
 Kernel Version:             4.16.14
 OS Image:                   Buildroot 2018.05
 Operating System:           linux
 Architecture:               amd64
 Container Runtime Version:  docker://17.12.1-ce
 Kubelet Version:            v1.10.0
 Kube-Proxy Version:         v1.10.0
Non-terminated Pods:         (32 in total)
  Namespace                  Name                          CPU Requests  CPU Limits  Memory
Requests  Memory Limits
  ---------                  ----                          ------------  ----------  ---------------  -------------
  default                    badpod                        0 (0%)        0 (0%)      0 (0%)           0 (0%)
  default                    constraintpod                 500m (25%)    500m (25%)  64Mi (3%)
64Mi (3%)
```

```
  default                envs                                    0 (0%)       0 (0%)     0 (0%)         0 (0%)
  default                first-deployment-59f6bb4956-p575s       0 (0%)       0 (0%)     0 (0%)
0 (0%)
  default                hc                                  0 (0%)       0 (0%)     0 (0%)         0 (0%)
  default                http-7b77c4cd66-ln4pn               0 (0%)       0 (0%)     0 (0%)         0
(0%)
  default                http-7b77c4cd66-sh55h               0 (0%)       0 (0%)     0 (0%)         0
(0%)
  default                http-7b77c4cd66-whdds               0 (0%)       0 (0%)     0 (0%)
0 (0%)
  default                httpexposed-64b57f7766-tw9c7        0 (0%)       0 (0%)     0 (0%)
0 (0%)
  default                jumpod                              0 (0%)       0 (0%)     0 (0%)         0 (0%)
  default                labelex                             0 (0%)       0 (0%)     0 (0%)         0 (0%)
  default                labelexother                        0 (0%)       0 (0%)     0 (0%)         0 (0%)
  default                logme                               0 (0%)       0 (0%)     0 (0%)         0 (0%)
  default                oneshot                             0 (0%)       0 (0%)     0 (0%)         0 (0%)
  default                onspecificnode                      0 (0%)       0 (0%)     0 (0%)         0
(0%)
  default                rcsise-gbx65                        0 (0%)       0 (0%)     0 (0%)         0
(0%)
  default                rcsise-zl8zv                        0 (0%)       0 (0%)     0 (0%)         0 (0%)
  default                ready                               0 (0%)       0 (0%)     0 (0%)         0 (0%)
  default                sharevol                            0 (0%)       0 (0%)     0 (0%)         0 (0%)
  default                twocontainers                       0 (0%)       0 (0%)     0 (0%)         0
(0%)
  kube-system            etcd-minikube                       0 (0%)       0 (0%)     0 (0%)         0
(0%)
  kube-system            kube-addon-manager-minikube         5m (0%)      0 (0%)     50Mi
(2%)       0 (0%)
  kube-system            kube-apiserver-minikube             250m (12%)   0 (0%)     0 (0%)
0 (0%)
  kube-system            kube-controller-manager-minikube    200m (10%)   0 (0%)     0
(0%)       0 (0%)
  kube-system            kube-dns-86f4d74b45-q5w5d           260m (13%)   0 (0%)
110Mi (6%)     170Mi (9%)
  kube-system            kube-proxy-krpt4                    0 (0%)       0 (0%)     0 (0%)
0 (0%)
  kube-system            kube-scheduler-minikube             100m (5%)    0 (0%)     0 (0%)
0 (0%)
  kube-system            kubernetes-dashboard-5498ccf677-bmwf5  0 (0%)    0 (0%)     0
(0%)       0 (0%)
  kube-system            storage-provisioner                 0 (0%)       0 (0%)     0 (0%)
0 (0%)
  other                  rcsise-mkjb5                        0 (0%)       0 (0%)     0 (0%)         0 (0%)
  other                  rcsise-w8s6j                        0 (0%)       0 (0%)     0 (0%)         0 (0%)
  test                   podintest                           0 (0%)       0 (0%)     0 (0%)         0 (0%)
```

```
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource  Requests     Limits
  --------  --------     ------
  cpu       1315m (65%)  500m (25%)
  memory    224Mi (12%)  234Mi (13%)
Events:     <none>
```