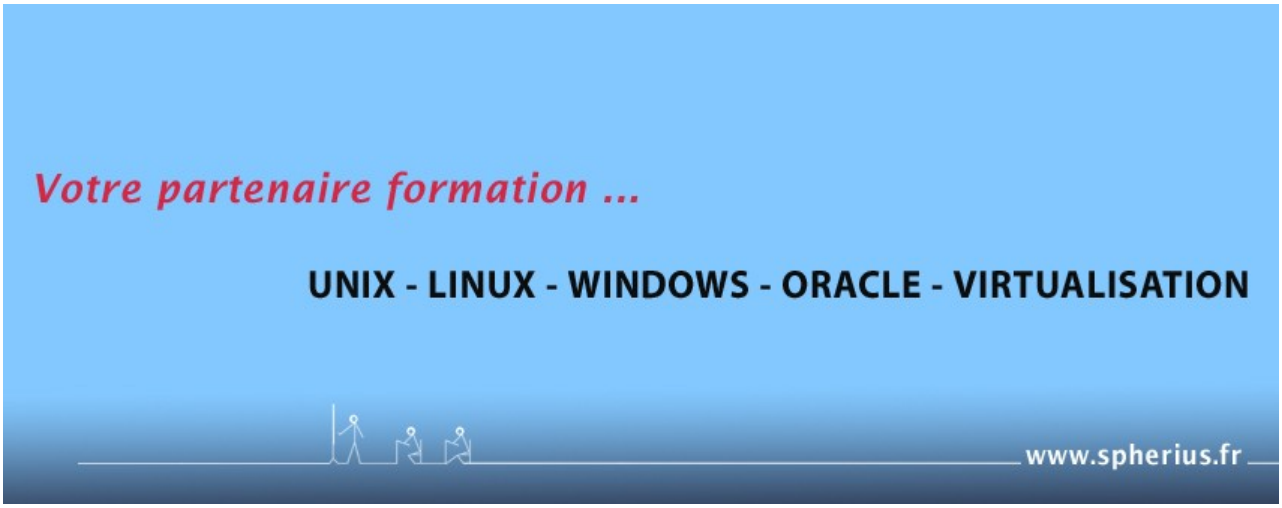
KUBERNETES



SOMMAIRE

Les auteurs de ce document sont :

- Steeven Herlant,
- Jean-Marc Baranger,
- Theo Schomaker.

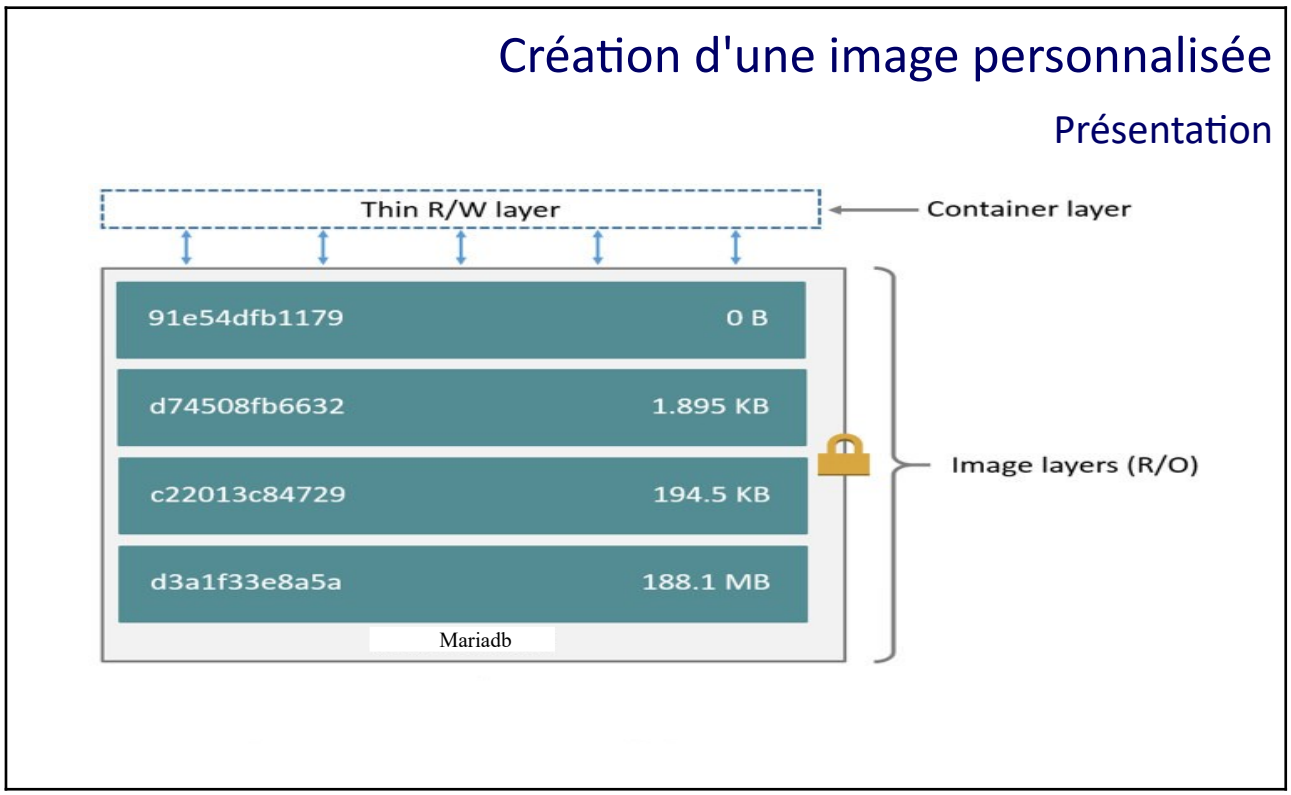
La version de Linux utilisée pour les commandes de ce support de cours est :

Ubuntu 20.04

Les références sont : les documents disponibles sur le site web de Kubernetes.

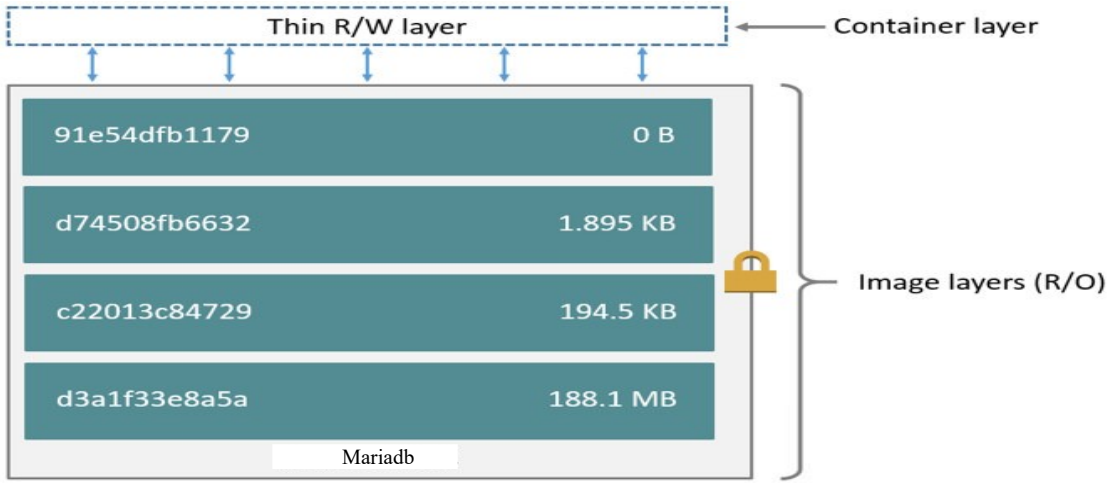
Rappels - Création d'une image personnalisée

Dans ce chapitre nous allons étudier la création d'images personnalisées.



Présentation

Une image se représente sous forme de couches (layers) correspondant à différents répertoires sous l'arborescence du répertoire où sont localisées les images.



Création d'une image personnalisée

Le fichier Dockerfile

```
$ tree  projet1
projet1
├── Dockerfile
└── src
    ├── database.sql
    └── index.html
```

```
$ cat Dockerfile
FROM centos:latest

RUN yum  install -y  wget  gcc  make
RUN wget  https://ftp.gnu.org/gnu/hello/hello-2.10.tar.gz
RUN tar  zxvf  hello-2.10.tar.gz
RUN cd  hello-2.10 && ./configure && make && make  install
```

```
$ docker  build  -t bonjour  .
```

Le fichier Dockerfile

Le fichier Dockerfile contient toutes les instructions pour créer une image personnalisée.

La bonne pratique est de créer un répertoire pour le projet de création d'une image. Il contiendra le fichier Dockerfile, ainsi que tous les fichiers nécessaires.

Exemple :

```
$ tree  projet1
projet1
├── Dockerfile
└── src
    ├── database.sql
    └── index.html
```

```
$ cat Dockerfile
FROM centos:latest

RUN yum  install -y  wget  gcc  make
RUN wget  https://ftp.gnu.org/gnu/hello/hello-2.10.tar.gz
RUN tar  zxvf  hello-2.10.tar.gz
RUN cd  hello-2.10 && ./configure && make && make  install
```

FROM — indique l'image source à utiliser.
Bonne pratique: spécifier la version (tag).

RUN — pour exécuter une commande.

La sous-commande build permet de créer une image en utilisant le fichier Dockerfile. Ce dernier est spécifié en argument :

docker build -t nom_de_l_image repertoire_du_fichier_Dockerfile

```
$ docker  build  -t bonjour  .
Step 1/5 : FROM centos:latest
latest: Pulling from library/centos
aeb7866da422: Pull complete
Digest: sha256:67dad89757a55bfdfabec8abd0e22f8c7c12a18565147264702280 63ed86593b
Status: Downloaded newer image for centos:latest
 ---> 75835a67d134
Step 2/5 : RUN yum  install -y  wget  gcc  make
 ---> Running in e697ef1a5edd
. . .
Removing intermediate container e697ef1a5edd
 ---> 51e1f0bd34fa
Step 3/5 : RUN wget  https://ftp.gnu.org/gnu/hello/hello-2.10.tar.gz
 ---> Running in f5e89bc3b6ab
. . .
Removing intermediate container f5e89bc3b6ab
 ---> c010a8c15c7a
Step 4/5 : RUN tar  zxvf  hello-2.10.tar.gz
 ---> Running in bc5cb7525b1a
. . .
Removing intermediate container bc5cb7525b1a
 ---> 2fdbfc5d6549
Step 5/5 : RUN cd  hello-2.10 && ./configure && make && make  install
 ---> Running in 16b3673c7c85
. . .
Removing intermediate container 16b3673c7c85
 ---> e6078bfa581b
Successfully built e6078bfa581b
Successfully tagged bonjour:latest
$

$ docker  images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
bonjour             latest              e6078bfa581b        4 minutes ago       354MB
centos              latest              75835a67d134        4 weeks ago         199MB

$ docker history bonjour
IMAGE          CREATED         CREATED BY                                      SIZE
e6078bfa581b   3 minutes ago   /bin/sh -c cd  hello-2.10 && ./configure && …  1.49MB
2fdbfc5d6549   4 minutes ago   /bin/sh -c tar  zxvf  hello-2.10.tar.gz        3.09MB
c010a8c15c7a   4 minutes ago   /bin/sh -c wget  https://ftp.gnu.org/gnu/hel…  726kB
51e1f0bd34fa   4 minutes ago   /bin/sh -c yum  install -y  wget  gcc  make    147MB
75835a67d134   3 weeks ago     /bin/sh -c #(nop)  CMD ["/bin/bash"]            0B
<missing>      3 weeks ago     /bin/sh -c #(nop)  LABEL org.label-schema.sc…  0B
<missing>      3 weeks ago     /bin/sh -c #(nop) ADD file:fbe9badfd2790f074…  200MB
```

Pour minimiser la taille de l'image, il faut minimiser le nombre de couches :

```
$ cat  Dockerfile
FROM centos:latest

RUN yum  install -y  wget  gcc  make \
    &&  wget  https://ftp.gnu.org/gnu/hello/hello-2.10.tar.gz \
    &&  tar  zxvf  hello-2.10.tar.gz \
    &&  cd  hello-2.10 && ./configure && make && make  install \
    &&  yum remove -y   make gcc wget \
    &&  cd / && rm -rf hello-2.10.tar.gz hello-2.10
```

```
$ docker  build  -t bonjour3  .
Sending build context to Docker daemon  2.048kB
Step 1/2 : FROM centos:latest
 ---> e934aafc2206
Step 2/2 : RUN yum  install -y  wget  gcc  make     &&  wget
https://ftp.gnu.org/gnu/hello/hello-2.10.tar.gz     &&  tar  zxvf  hello-2.10.tar.gz
&&  cd  hello-2.10 && ./configure && make && make  install     &&  yum remove -y   make
gcc wget     &&  cd / && rm -rf hello-2.10.tar.gz hello-2.10
 ---> Running in 7c9dbbe4676e
. . .
Removing intermediate container 7c9dbbe4676e
 ---> bf99120c78ff
Successfully built bf99120c78ff
Successfully tagged bonjour3:latest
$

$ docker  images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
bonjour3            latest              bf99120c78ff        31 seconds ago      324MB
bonjour             latest              5a40877c50fd        20 minutes ago      354MB
```

Pour conserver que la dernière image :

```
$ docker  rmi  bonjour
Untagged: bonjour:latest
Deleted: sha256:5a40877c50fd3c7d1072fd64777347814fcb22b3a55f9c54161ac3cfaa9cd442
Deleted: sha256:7926c1dbc65668d85c7433f91d8cc25efffebef9bd26e001962b90fba615c3c3
Deleted: sha256:bb62f56c5e8a2cf3a064ee6d0e7afdc29cbff3ba0e4985c9b12b90e39a955afe
Deleted: sha256:f6e4ed59f989febbcf5db6e694ebcd2654224df7306237b5b5ec6b2d4573bb3b
Deleted: sha256:8d026888fd51ab6c9c8e1488842f6cfa326f11cc8e2705f33c6c6134e6fbd7c6
Deleted: sha256:0f34d6969d72d17e54f1ed4834e2be2fcb649feed5e97fd26d69074c75bb5b29
Deleted: sha256:d47e74858532cb56ba39549748353b84d3b8541d3917dce6732fbcc36c4557c3
Deleted: sha256:c2fe39a1aeb2d8c01ef5ded8c1f436b52972cc65e227b68f742c4842bdf099d5


$ docker  tag  bonjour3  bonjour
$ docker  rmi  bonjour3
Untagged: bonjour3:latest
$ docker  images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
bonjour             latest              bf99120c78ff        5 minutes ago       324MB
```

Création d'une image personnalisée

Le Dockerfile – les mots clefs

ENTRYPOINT	CMD
LABEL	
EXPOSE	VOLUME
COPY	ADD
USER	
ENV	ARG
WORKDIR	STOPSIGNAL

Le Dockerfile – les mots clefs

Les lignes commençant par dièse sont des commentaires.

ENTRYPOINT impose la commande à exécuter lors du démarrage du conteneur.

ENTRYPOINT ["executable", "param1", "param2"]

```
$ cat  Dockerfile
FROM centos:latest

RUN yum  install -y  wget  gcc  make \
    &&  wget  https://ftp.gnu.org/gnu/hello/hello-2.10.tar.gz \
    &&  tar  zxvf  hello-2.10.tar.gz \
    &&  cd  hello-2.10 && ./configure && make && make  install \
    &&  yum remove -y   make gcc wget \
    &&  cd / && rm -rf hello-2.10.tar.gz hello-2.10

ENTRYPOINT ["hello"]
```

```
$ docker  build  -t  bonjour4  .

$ docker  run  --rm  bonjour4
Hello, world!

$ docker  run  --rm  bonjour4  -g "Bonne journee"
Bonne journee
```

CMD définit la commande par défaut à exécuter lors du démarrage du conteneur.

CMD ["cat","/etc/passwd","/etc/group"]
Commande exécutée : cat /etc/passwd /etc/group

LABEL **maintener="baranger@spherius.fr"**
LABEL sert à définir des méta données.
Le mot clé maintener définit l'auteur de l'image.

EXPOSE pour exposer un port réseau.

VOLUME pour créer un volume pour le stockage.

COPY pour copier un fichier ou répertoire de l'hôte vers l'image.

ADD pour copier un fichier (ou répertoire) de l'hôte ou depuis une URL vers l'image, il sert également à décompresser automatiquement une archive (tar, zip, etc).

USER définit l'utilisateur qui lance la commande du ENTRYPOINT ou du CMD.

ENV pour définir des variables d'environnement pour l'image.
on peut surcharger la valeur lors de l'exécution (run) par l'option "-e".

ARG similaire à ENV, mais juste le temps de la construction de l'image.

WORKDIR définit le répertoire de travail.
Il correspond au répertoire de travail lorsque l'on se connecte au conteneur.
Il sert de répertoire de base pour les chemins relatifs du Dockerfile pour les instructions qui sont après WORKDIR : ADD, COPY, RUN, CMD, ENTRYPOINT.

STOPSIGNAL définit le signal qui sera envoyé au conteneur lorsqu'il sera stoppé par docker stop.

Exemple :

```
$ tree .
.
├── Dockerfile
└── src
    ├── site.conf
    └── index.modele
```

```
$ cat  Dockerfile
FROM ubuntu

LABEL description="Test de creation d image" \
      maintainer="Baranger Jean-Marc" \
      version="1.0"

ARG reppage=/var/www/html repconf=/etc/apache2
ENV ville Paris
ENV APACHE_RUN_USER www-data
```

```
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/web/log/apache2
ENV APACHE_PID_FILE /var/run/apache2.pid
ENV APACHE_RUN_DIR /var/run/apache2
ENV APACHE_LOCK_DIR /var/lock/apache2

RUN export DEBIAN_FRONTEND=noninteractive && apt-get update && apt-get -y -q upgrade &&
apt-get -y -q install apache2

COPY src/index.modele ${reppage}/index.html
COPY src/site.conf  ${repconf}/apache2.conf

EXPOSE 80 443
VOLUME /var/www/html

WORKDIR /var/www

CMD ["apache2ctl","-D","FOREGROUND"]
```

```
$ docker  build  -t  jmb/apache  .

$ docker  images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
jmb/apache          latest              0476afa3d9e6        7 seconds ago       215MB

$ docker  run  -d  --name site  jmb/apache
de785fe243a19b5f6b60e2f468f80a1a0239faeb9b43e693215bb349cea467a0

$ docker  inspect  site  |  grep  -i  ipaddress
                    "IPAddress": "172.17.0.2",
```

Le site web fonctionne avec http://172.17.0.2.

Notes

Présentation de Kubernetes

Dans ce chapitre nous allons présenter les concepts de Kubernetes.

Présentation de Kubernetes

- Origine du projet
- De la virtualisation à la conteneurisation
- Problèmes soulevés par la conteneurisation
- Les solutions apportées par Kubernetes
- Conteneurs supportés, plates-formes utilisant Kubernetes
- Composants de Kubernetes
- Définitions: pods, labels, controllers, services

Présentation de Kubernetes

Origine du projet

- Le projet BORG
- Google et la CNCF

kubernetes

Origine du projet

Kubernetes est une plate-forme de gestion de conteneurs proposée par Google. A l'origine, il provient du système BORG, outil interne à Google, lui permettant de gérer ses quelques milliards de conteneurs.

En 2015, Google s'associe à la Fondation Linux pour créer la CNCF (Cloud Native Computing Foundation). Cette organisation à caractère non lucratif a pour but de se concentrer sur l'élaboration de softs open-source liés à la zone de croissance rapide des applications cloud native.

Cette Organisation compte comme contributeurs des grands noms de l'informatique tels que AWS, Microsoft Azure, Oracle, Red Hat et bien d'autres.

Lors de leur association, Google offre à la CNCF Kubernetes, un fork open-source de leur solution interne. Depuis, Kubernetes n'a cessé de grandir et est aujourd'hui devenu quasiment incontournable pour qui souhaite se lancer dans l'ère des conteneurs en production.

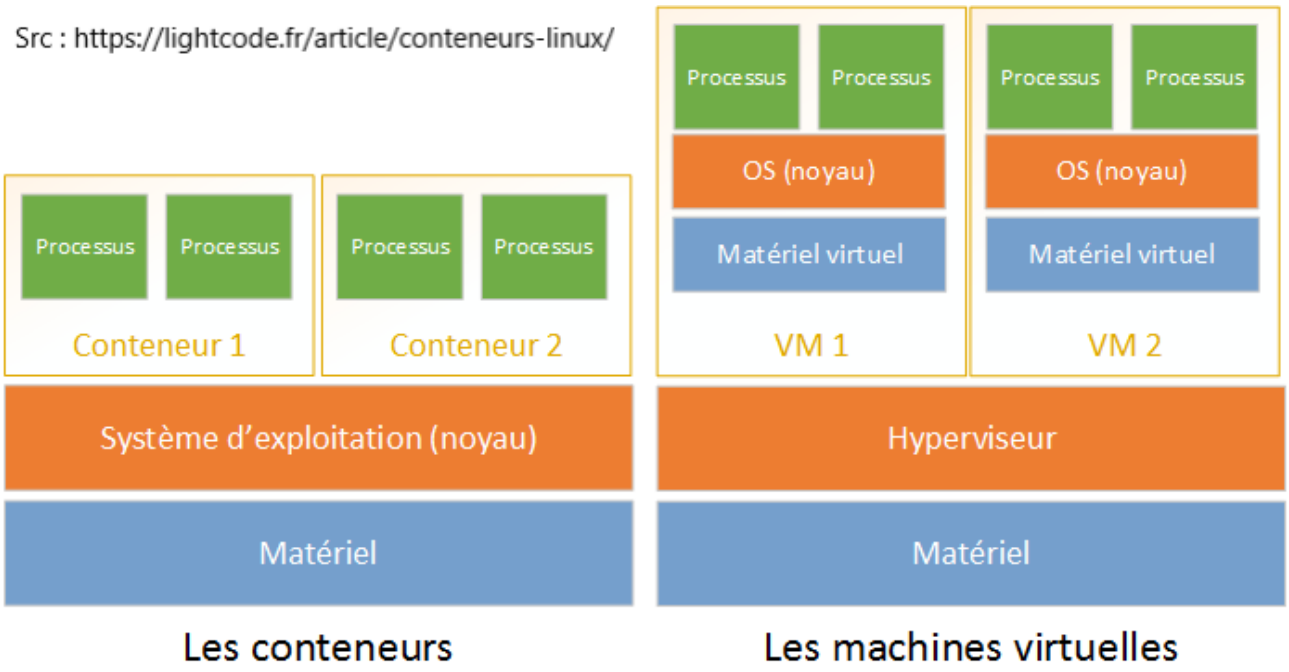
Présentation de Kubernetes

De la virtualisation à la conteneurisation

- Principes généraux de virtualisation
- Principes généraux de la conteneurisation
- Avantages de la conteneurisation

De la virtualisation à la conteneurisation

Src : https://lightcode.fr/article/conteneurs-linux/



La virtualisation consiste à exécuter sur un système hôte appelé un hyperviseur, une instance virtuelle d'une infrastructure matérielle. Cette instance possédera son propre système d'exploitation, ses librairies, binaires et fichiers de configuration.

La conteneurisation, quand à elle, consiste à créer des instances virtuelles qui vont partager le même système d'exploitation, et contenir uniquement l'application, avec ses binaires et les bibliothèques associées.

Les avantages de la conteneurisation :

- La rapidité de déploiement : Un conteneur ne contient que ce qui lui est strictement nécessaire pour s'exécuter.
- La migration : Il est très facile de réaliser un test de développement d'application dans un conteneur en local, puis de déployer ce conteneur sur le cloud par exemple.
- Architecture micro-services : les conteneurs amènent avec eux cette nouvelle notion qui consiste à déployer une application en plusieurs services plutôt qu'en un service monolithique. Cette architecture apporte elle aussi son lot d'avantages : maintenabilité du code, élasticité de l'application, optimisation par briques…

Présentation de Kubernetes

Problèmes soulevés par la conteneurisation

- Mise à jour
- Supervision
- Gestion à grande échelle

Problèmes soulevés par la conteneurisation

Avec l'arrivée des conteneurs, sont arrivés de nouveaux problèmes :

- Un conteneur repose sur une image, généralement récupérée sur le Docker HUB ou dans un registre privé.

  Comment dans ce cas, pouvons nous gérer les mises à jour de nos conteneurs, sachant que les images, elles sont mises à jour très régulièrement ?

  De plus, dans un environnement de production, comment pouvons-nous mettre à jour nos conteneurs sans interruption de services ?

- Un conteneur est par définition un environnement éphémère. En effet, en cas de dysfonctionnement d'une machine virtuelle, la conduite à tenir est généralement de résoudre manuellement le problème rencontré. Un conteneur pouvant être déployé très rapidement, on préfère généralement le supprimer pour le reconstruire plutôt que de perdre du temps à essayer de comprendre d'où vient le problème, sauf bien sur s'il s'agit d'un problème récurrent, dont l'origine est en général une image défectueuse.

  Cela implique un grand nombre de destructions et de constructions de conteneurs. Dans un environnement tel que celui-ci, comment gérer la supervision et la métrologie de nos conteneurs ?

- A la destruction de nos conteneurs, tout ce qui compose celui-ci est détruit, y compris les données qui auraient pu y être stockées.

  Comment gérer dans ce cas la persistance de nos données, notamment pour le cas de services ou celle-ci est primordiale, comme par exemple pour des bases de données ?

- Comment pouvons-nous également gérer un problème tel que la montée en charge de nos applications, au vu du nombre de conteneurs que nous avons à gérer ?

- Une application doit pouvoir être exposée aux utilisateurs. Nous devons donc pouvoir également gérer facilement cette exposition, ce qui encore une fois est rendu difficile de part la multiplication de nos conteneurs.

Tous ces problèmes peuvent bien sur être traités soit par l'élaboration de script d'administration ou la mise en place d'outils, mais cette démarche réclame de connaître énormément de produits différents.

Présentation de Kubernetes

Les solutions apportées par Kubernetes

- Mises à jour
- Montée en charge
- Persistance

Les solutions apportées par Kubernetes

Kubernetes se propose de nous apporter des solutions aux problèmes évoqués.

C'est cet outil qui va nous permettre de gérer automatiquement le cycle de vie de nos conteneurs (mises à jour).

Il va s'occuper de la gestion de la montée en charge, qu'elle soit applicative ou système.

Il pourra ainsi démarrer de nouveaux conteneurs lorsqu'il détectera une montée en charge, afin que celle-ci puisse être absorbée.

C'est lui qui vérifiera l'état de santé de nos conteneurs, et il pourra en cas de défaillance de l'un d'eux, en démarrer un nouveau et arrêter l'ancien.

Kubernetes s'occupera de la gestion de nos volumes persistants, action rendue difficile notamment par la gestion multi-nodes.

Il nous permettra de gérer également l'environnement de nos conteneurs (variables, configuration…) ainsi que l'exposition de nos applications aux utilisateurs.

Présentation de Kubernetes

Containers supportés, plates-formes utilisant Kubernetes

- Types de conteneurs
- Types de plate-formes

Containers supportés, plates-formes utilisant Kubernetes

Kubernetes permet de gérer divers conteneurs :

- Docker : Sans doute le plus connu.
- Rkt : Container Engine délivré dans les distributions type Archlinux ou Fedora.
- Containerd : Container Engine entré dans la CNCF depuis début 2019.
- Cri-containerd : Implémentation de containerd prévue pour Kubernetes.
- CRI-O : Container Engine développé pour Kubernetes, se veut plus léger que Docker.
- Katacontainers : Container Engine tourné sécurité.

Même si l'on retrouve généralement Kubernetes sur des plate-formes Linux, il est également possible de l'installer sur Windows et MacOS.

Nous pouvons également, surtout à des fins de tests, installer une plate-forme sur des architectures ARM, comme par exemple sur des Raspberry PI

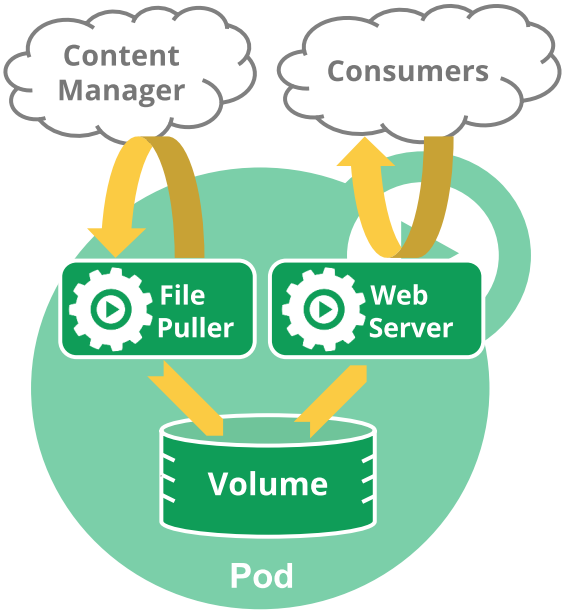
Présentation de Kubernetes

Définitions: pods, labels, controllers, services

- Pods
- Labels
- Controllers
- Services

Définitions: pods, labels, controllers, services

Issue de la documentation officielle de Kubernetes



- Pods : Il s'agit de la plus petite entité présente dans Kubernetes. Un pod est composé d'un ou plusieurs conteneurs. Ces conteneurs vont avoir un stockage et réseau partagé.

Ils vont également profiter d'un contexte partagé, c'est à dire qu'ils vont reposer sur les mêmes namespaces Linux et les mêmes cgroups.

Ils partagent une même adresse IP et un espace de ports, et peuvent communiquer via localhost.

Pour faire une analogie avec docker utilisé en standalone, Si l'on part sur une simple application web avec le code de l'application d'un côté et une base de données de l'autre, nous aurions nos 2 conteneurs docker avec un réseau mis en place sur chacun de ces conteneurs pour permettre la communication, et parfois, un autre réseau pour gérer l'exposition aux utilisateurs.

Avec les pods, toutes ces entités peuvent être gérées au sein d'un seul et unique pod. Nous aurions donc juste à gérer notre pod, avec à l'intérieur nos différents conteneurs.

- Labels : Ce sont des ensembles de clés/valeurs que nous allons pouvoir définir sur les objets Kubernetes, tels que les pods. Grâce à ses labels, nous allons pouvoir travailler de manière plus efficiente, en sélectionnant par exemple tous nos pods marqués « production », ou encore toutes les pods dont le code est du python pour une montée de version…

- Controllers : Il s'agit de boucles de contrôle permettant de surveiller l'état du cluster.

  Les controllers s'exécutent dans le Kube-controller-manager. Kubernetes embarque par défaut de nombreux controllers, chacun gérant un type de ressource.

  Il est également possible d'ajouter des controllers, voir même d'en créer, afin d'étendre les capacités de son cluster.

- Services : Un pod est une entité éphémère. Cela signifie qu'à chaque nouvelle création, un nouveau nom sera créé pour le pod. Cela pose des difficultés, notamment pour l'accès à nos applications depuis l'extérieur. C'est la que la notion de service entre en jeu. Il va nous permettre de donner un nom DNS à nos applications, nom qui sera réutilisé même si le pod contenant l'application est détruit et recréé.

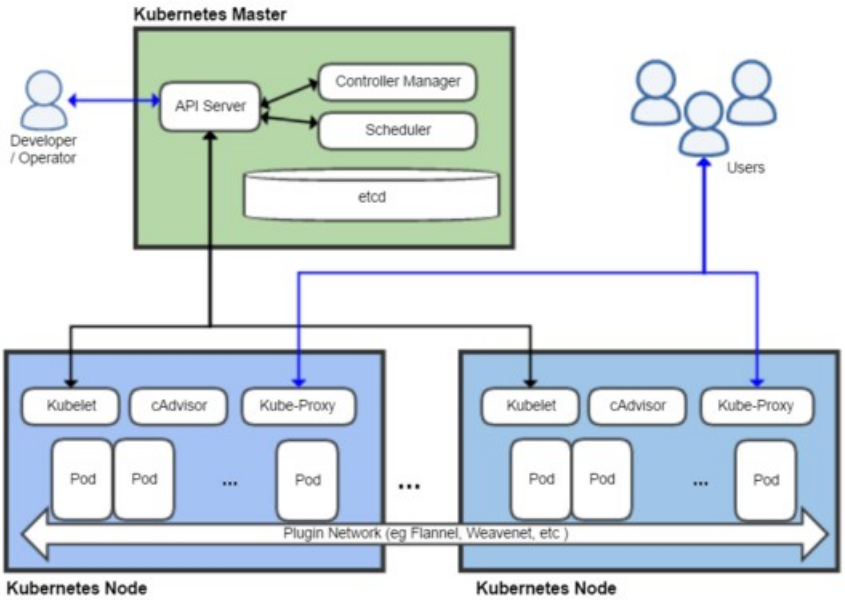
Notes

Architecture

Dans ce chapitre nous allons découvrir l'architecture d'un cluster Kubernetes.

Architecture

- Kubernetes Master : etcd, API server, Controller manager, Scheduler
- Kubernetes Node : Kubelet, pods
- Le réseau dans Kubernetes



Architecture

Kubernetes Master : etcd, API server, Controller manager, Scheduler

- Etcd
- API server
- Controller manager
- Scheduler

Kubernetes Master : etcd, API server, Controller manager, Scheduler

Issue de la documentation officielle de Kubernetes (kubernetes.io)

Le Kube Master est l'élément de contrôle du cluster. C'est lui qui va s'occuper de prendre les décisions. Le Kube Master est composé des éléments suivants :

- kube-apiserver:Composant qui va s'occuper d'exposer l'API de Kubernetes. C'est par le biais de cette API que passe les ordres donnés par le client à Kubernetes.
- Etcd : Base de données du cluster. Elle recense tous les objets composant le cluster ainsi que leur état. Lorsque l'on donne un ordre via l'API, c'est cette base qui est modifiée.
- Kube-scheduler : S'occupe de répartir les pods sur les différents nœuds du cluster. Il s'occupe également de la partie planification du cluster.
- Kube-controller-manager : Composant qui s'occupe de la gestion des controllers du cluster.
- Cloud-controller-manager : Permet de faire la liaison entre les commandes d'un cluster Kubernetes et celles des cloud providers.

Architecture

Kubernetes Node : Kubelet, pods

- Kubelet
- Pods

Kubernetes Node : Kubelet, pods

Les nodes vont fournir l'environnement d'exécution du cluster. Ils sont composés des éléments suivants :

- Kubelet : Permet de superviser les conteneurs au sein des pods. C'est lui qui va prendre en charge le démarrage, l'arrêt et la maintenance des conteneurs.

- Container Runtime : Responsable de l'exécution des conteneurs. Comme vu précédemment, gère un bon nombre de conteneurs différents.

Les nodes (aussi appelés workers) sont en charge d'exécuter les pods. Via le kubelet, le master reçoit régulièrement le statut des workers. Si l'un d'entre eux n'est pas en bonne santé, le cluster relancera les pods qui tournaient dessus sur les autres workers du cluster, par le biais du Replication Controller.

Architecture

Le réseau dans Kubernetes

- Solution k8s par défaut
- CNI et un plugin réseau

Le réseau dans Kubernetes

Source : https://kubernetes.io/docs/concepts/cluster-administration/addons/#networking-and-network-policy

La partie souvent la plus complexe lorsque l'on gère un cluster, et Kubernetes ne fait pas exception, est bien souvent la gestion du réseau.

En effet, il faut qu'un mécanisme soit mis en place pour que lorsque l'on cherche à accéder à une application, le routage soit correctement effectué vers le pod sur le worker du cluster en question.

Sur Kubernetes, il existe une première solution native : Il s'agit en fait de configurer manuellement des bridges sur l'ensemble des nœuds, avec une plage d' adresses ip, puis à ajouter les routes nécessaires entre les différents nœuds. Bien que possible, cette solution est longue et fastidieuse à mettre en place.

Une deuxième solution consiste à utiliser le CNI (Container Network Interface). Ce CNI est un ensemble de spécifications et librairies ayant pour but de faciliter l'intégration de plugins réseaux.

Les 3 solutions basées sur le CNI les plus utilisées dans Kubernetes sont :

- Calico : cette solution se veut simple, scalable et sécurisée. Elle repose sur l'utilisation de Kube-proxy, un composant natif à Kubernetes. Kube-proxy utilise les iptables Linux pour créer des règles de filtrage sur les réseaux, ce qui permet l'isolation des conteneurs.

  Calico se sert de différents composants :

  - Felix : un agent, présent sur chaque hôte, qui fournit des interfaces vers l'extérieur. Il partage les tables d'adressage ip et les routes entre les nœuds.
  - BIRD : un client qui fait office de routeur.
  - Confd : sert à monitorer etcd et à générer les configurations BIRD.

- Flannel : Cette solution repose sur le VxLAN. Flannel créé un réseau de ce type et met en place un sous-réseau par hôte grâce à un agent flanneld. Flannel utilise également etcd pour stocker ses configurations.

- WeaveNet : Contrairement aux autres solutions, WeaveNet n'utilise pas etcd pour stocker les données, mais les mets dans un fichier /weavedb/weave-netdata.db sur chaque pod créé par le DaemonSet. Ces pods sont créés sur chaque worker (pas le master) et contiennent 2 conteneurs. Un conteneur weave qui gère tout le fonctionnement de Weave sur le nœud et un conteneur weave-npc qui s'occupe de la partie filtrage. Les pods possèdent l'adresse ip du nœud sur lequel ils sont placés. Comme Flannel, il se base sur l'utilisation de VxLAN.

Notes

Installation et Configuration

Dans ce chapitre nous allons installer et configurer un cluster Kubernetes.

Installation et Configuration

- Présentation des différentes solutions d'installation
- Pré-requis à la solution retenue pour ce cours
- Installation des outils
- Mise en place du cluster

Installation et Configuration

Présentation des différentes solutions d'installation

- Minikube
- Kubeadm
- Cloud Provider

Présentation des différentes solutions d'installation

Kubernetes est une solution complexe à installer from scratch. Il faut de très solides connaissances système pour réussir l'exercice. Il existe toutefois des solutions pour installer de manière plus ou moins automatisé un cluster Kubernetes et pouvoir ainsi commencer à travailler dessus.

- Minikube : Cette solution permet d'installer un cluster Kubernetes en local sur sa machine. Il va en fait créer une Machine Virtuelle et créer à l'intérieur un cluster à un nœud.

  Cette méthode peut être utile pour tester certaines fonctionnalités de Kubernetes, mais elle ne respecte malheureusement pas l'architecture qu'un cluster devrait avoir, ce qui nous fait passer à côté de fonctionnalités.

- Kops : Cet utilitaire s'appuie sur le principe de kubeadm, mais propose cette fois-ci une installation entièrement automatisée. Kops permet de provisionner un cluster Kubernetes sur Linux, via des outils tel que Terraform, ou encore sur des plate-forme Cloud telle que AWS.

- Kubespray : Cet outil permet de provisionner un cluster Kubernetes en utilisant Kubeadm et Ansible. Comme Kops, l'installation est entièrement automatisée.
  Il peut être déployé sur des conteneurs Linux, ou sur les principales distributions telles Ubuntu, Debian ou CentOS.

- Installation sur Cloud Provider : La plupart des Cloud Provider proposent des solutions utilisant Kubernetes. Bien évidemment ces solutions sont payantes. Chaque Cloud Provider propose des outils internes permettant de vous assister lors de l'installation de votre cluster.

- Kubeadm : Kubeadm est un autre utilitaire permettant d'installer un cluster Kubernetes. Il peut s'agir cette fois ci d'une installation locale sur une machine, ou d'une installation plus classique sur plusieurs nœuds. Ces nœuds peuvent être des ordinateurs, des machines virtuelles, des raspberry pi ou bien encore des serveurs.
  C'est cette méthode que nous utiliserons dans cette formation. Elle nous permettra de mieux appréhender les différents concepts de Kubernetes.

Installation et Configuration

Pré-Requis à la solution retenue pour ce cours

- Infrastructure de base
- Configuration initiale des machines virtuelles

Pré-Requis à la solution retenue pour ce cours

Pour déployer notre cluster Kubernetes, nous allons donc utiliser l'outil Kubeadm.
Nous utiliserons 3 machines virtuelles : 1 master et 2 workers.
Cela nous permettra d'avoir matière pour utiliser les fonctionnalités d'un cluster.
Une quatrième machine virtuelle, nommé admin, nous servira à installer et administrer notre cluster.

Selon la documentation officielle, nos machines virtuelles doivent comporter au minimum 2Go de Ram et 2 CPUs. Deux interfaces réseaux seront utilisées par machine. La première nous permettra d'avoir accès à internet, tandis que la seconde servira à la communication entre les nœuds. Il est évident que dans le cas d'une installation dans un environnement de production, il nous faudrait plusieurs interfaces afin de séparer les flux (admin, métier…).
Il est à noter qu'il ne faut pas configurer de swap sur les nœuds d'un cluster Kubernetes.

Nos machines seront donc installées en Ubuntu 20.04. Nous détaillons ci-dessous les différentes opérations à effectuer à l'issue d'une installation système de base de cette distribution.

Les 3 paquets que nous devrons installer sur notre cluster sont les suivants :

- kubeadm : Cette commande nous permettra d'initialiser le cluster.
- kubelet : Ce composant s'exécute sur l'ensemble des nœuds du cluster et sert à effectuer des actions tel que le démarrage des pods.
- kubectl : Cette commande nous permet de communiquer avec le cluster.

Installation et Configuration

Installation des Outils

- Configuration des dépôts
- Installation des paquets
- Tuning OS

Installation des Outils

Il nous faut d'abord installer un certain nombre de paquets dont nous allons avoir besoin :

```
# apt install -y apt-transport-https gnupg2 software-properties-common
ca-certificates curl wget
```

Nous pouvons ensuite configurer l'accès aux dépôts Kubernetes :

```
# mkdir -p /etc/apt/keyrings
# curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.28/deb/Release.key \
    | sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg
# cat <<EOF > /etc/apt/sources.list.d/kubernetes.list
deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg]
  https://pkgs.k8s.io/core:/stable:/v1.28/deb/ /
EOF
```

Il faut également mettre en place les règles sysctl afin de permettre aux paquets « bridgés » de traverser les règles iptables :

```
# cat <<EOF > /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
net.ipv4.ip_forward = 1
EOF
# sysctl --system
```

Nous devons maintenant activer les modules br_netfilter et overlay :

```
# modprobe br_netfilter
# modprobe overlay
```

Il nous faut également désactiver la swap. Cela fait partie des pré-requis que l'on peut trouver sur le site de Kubernetes :

```
# swapoff -a
```

Il faut penser à commenter la ligne correspondant à la swap dans le fichier /etc/fstab. Dans le cas contraire, notre modification ne sera pas persistante au reboot.

Nous allons maintenant installer un moteur de conteneurs sur chacun de nos nœuds. Dans le cadre de ce cours, nous avons choisi d'installer containerd. Mais il est également possible d'en installer d'autres, comme CRI-O par exemple. Cette manipulation est à réaliser sur les 3 nœuds :

Mise en place du dépôt Docker afin de pouvoir installer containerd :

```
# curl -s https://download.docker.com/linux/ubuntu/gpg | apt-key add -

# add-apt-repository "deb [arch=amd64] \
"https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
```

Installation de containerd :

```
# apt update && apt install -y containerd.io
```

Mise en place de la configuration de containerd :

```
# mkdir -p /etc/containerd
# containerd config default | tee /etc/containerd/config.toml
```

Modification de la configuration containerd afin d'utiliser systemd-cgroup :

```
# sed -i 's/SystemdCgroup = false/SystemdCgroup = true/' \
/etc/containerd/config.toml
```

Redémarrage et activation du service containerd :

```
# systemctl restart containerd
# systemctl enable containerd
```

Nous n'avons pas de pare-feu d'activé par défaut sur Ubuntu. Dans le cas contraire, il faut penser à ouvrir les ports suivants pour le bon fonctionnement de Kubernetes :

Sur le Master : 6443/tcp, 2379-2380/tcp, 10251/tcp, 10252/tcp, 10255/tcp.

Sur les Workers (les ports 30000 à 32767 serviront lorsque nous utiliserons les NodePorts, plus tard au cours de la formation) : 10250/tcp, 30000-32767/tcp.

Ces informations peuvent être trouvées sur le site de Kubernetes.

Nous pouvons maintenant installer les binaires nécessaires au cluster

```
# apt install -y kubelet kubeadm kubectl
```

Puis nous activons le service kubelet :

```
# systemctl enable --now kubelet
```

Sur un environnement de production, il convient également de bloquer les versions de ces paquets, afin de contrôler les mises à jour :

```
# apt-mark hold kubelet kubeadm kubectl
```

Nos 3 machines virtuelles sont maintenant prêtes pour l'installation du cluster Kubernetes.

Installation et Configuration

Mise en place du Cluster

- Initialisation du Master
- Configuration du Réseau
- Intégration (join) des Workers

Mise en place du Cluster

Pour configurer notre master, nous allons utiliser kubeadm. Il va falloir lui donner l'ip à utiliser pour que nous puissions communiquer avec le cluster, à savoir l'ip de notre VM master, ainsi que le hostname du master et le réseau que nous voulons donner à Kubernetes pour les ips internes des pods.

```
# kubeadm init --apiserver-advertise-address=192.168.56.31 --node-name
$HOSTNAME --pod-network-cidr=10.244.0.0/16
[init] Using Kubernetes version: v1.28.2
[preflight] Running pre-flight checks
[preflight] Pulling images required for setting up a Kubernetes cluster
[preflight] This might take a minute or two, depending on the speed of your internet
connection
[preflight] You can also perform this action in beforehand using 'kubeadm config images
pull'
[certs] Using certificateDir folder "/etc/kubernetes/pki"
[certs] Generating "ca" certificate and key
[certs] Generating "apiserver" certificate and key
[certs] apiserver serving cert is signed for DNS names [kubernetes kubernetes.default
kubernetes.default.svc kubernetes.default.svc.cluster.local master] and IPs [10.96.0.1
192.168.56.31]
[certs] Generating "apiserver-kubelet-client" certificate and key
[certs] Generating "front-proxy-ca" certificate and key
[certs] Generating "front-proxy-client" certificate and key
[certs] Generating "etcd/ca" certificate and key
[certs] Generating "etcd/server" certificate and key
[certs] etcd/server serving cert is signed for DNS names [localhost master] and IPs
[192.168.56.31 127.0.0.1 ::1]
[certs] Generating "etcd/peer" certificate and key
[certs] etcd/peer serving cert is signed for DNS names [localhost master] and IPs
[192.168.56.31 127.0.0.1 ::1]
[certs] Generating "etcd/healthcheck-client" certificate and key
```

```
[certs] Generating "apiserver-etcd-client" certificate and key
[certs] Generating "sa" key and public key
[kubeconfig] Using kubeconfig folder "/etc/kubernetes"
[kubeconfig] Writing "admin.conf" kubeconfig file
[kubeconfig] Writing "kubelet.conf" kubeconfig file
[kubeconfig] Writing "controller-manager.conf" kubeconfig file
[kubeconfig] Writing "scheduler.conf" kubeconfig file
[etcd] Creating static Pod manifest for local etcd in "/etc/kubernetes/manifests"
[control-plane] Using manifest folder "/etc/kubernetes/manifests"
[control-plane] Creating static Pod manifest for "kube-apiserver"
[control-plane] Creating static Pod manifest for "kube-controller-manager"
[control-plane] Creating static Pod manifest for "kube-scheduler"
[kubelet-start] Writing kubelet environment file with flags to file
"/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Starting the kubelet
[wait-control-plane] Waiting for the kubelet to boot up the control plane as static Pods
from directory "/etc/kubernetes/manifests". This can take up to 4m0s
[apiclient] All control plane components are healthy after 5.502474 seconds
[upload-config] Storing the configuration used in ConfigMap "kubeadm-config" in the
"kube-system" Namespace
[kubelet] Creating a ConfigMap "kubelet-config" in namespace kube-system with the
configuration for the kubelets in the cluster
[upload-certs] Skipping phase. Please see --upload-certs
[mark-control-plane] Marking the node master as control-plane by adding the labels:
[node-role.kubernetes.io/control-plane node.kubernetes.io/exclude-from-external-load-
balancers]
[mark-control-plane] Marking the node master as control-plane by adding the taints [node-
role.kubernetes.io/control-plane:NoSchedule]
[bootstrap-token] Using token: wtm6mn.qny5ybabfqm9vooh
[bootstrap-token] Configuring bootstrap tokens, cluster-info ConfigMap, RBAC Roles
[bootstrap-token] Configured RBAC rules to allow Node Bootstrap tokens to get nodes
[bootstrap-token] Configured RBAC rules to allow Node Bootstrap tokens to post CSRs in
order for nodes to get long term certificate credentials
[bootstrap-token] Configured RBAC rules to allow the csrapprover controller automatically
approve CSRs from a Node Bootstrap Token
[bootstrap-token] Configured RBAC rules to allow certificate rotation for all node client
certificates in the cluster
[bootstrap-token] Creating the "cluster-info" ConfigMap in the "kube-public" namespace
[kubelet-finalize] Updating "/etc/kubernetes/kubelet.conf" to point to a rotatable
kubelet client certificate and key
[addons] Applied essential addon: CoreDNS
[addons] Applied essential addon: kube-proxy

Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

Alternatively, if you are the root user, you can run:

  export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 192.168.56.31:6443 --token wtm6mn.qny5ybabfqm9vooh \
      --discovery-token-ca-cert-hash
sha256:940e48637695ed60c123f3117709b591c85ede4f0bb9fdf96e1f6bb47ff88699
```

Comme nous pouvons le voir, le lancement de cette commande va nous permettre d'initialiser le

cluster. A la fin, elle nous indique quelle commande lancer sur les workers pour les faire rejoindre le cluster, et nous indique également que nous devons créer le répertoire et le fichier de configuration du cluster, et également créer la partie réseau interne pour qu'il puisse fonctionner.

Nous allons tout d'abord créer le fichier de configuration :

```
$ mkdir -p $HOME/.kube
$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
$ sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Ces commandes vont nous permettre de copier le fichier de configuration du cluster dans notre home, et ainsi d'avoir à la commande kubectl afin d'interagir avec le cluster. Pour plus de sécurité, il convient d'effectuer ces opérations en tant qu'utilisateur non privilégié.

Il est à noter que cette manipulation peut être faite sur un autre poste (une station d'administration par exemple), pour communiquer avec le cluster sur ce poste. Il conviendra à se moment d'installer kubectl (et également de mettre en place le dépôt Kubernetes si nécessaire).

A ce stade nous pouvons récupérer l'état de notre nœud :

```
$ kubectl get nodes
NAME          STATUS     ROLES    AGE   VERSION
master        NotReady   master   78s   v1.28.2
```

Nous constatons que notre master est en status NotReady, ce qui est normal puisque le cluster attend que nous ajoutions un CNI, le composant réseau, avant de démarrer les pods system coredns.

Pour la partie réseau, nous avons vu précédemment qu'il existait différentes implémentations possible. Pour ce cours, nous avons fait le choix de Calico.

Dans le cas ou nous aurions un pare-feu activé (ce qui n'est pas le cas ici), nous aurions besoin d'activer l'ip masquerading, et d'ouvrir les ports suivants sur l'ensemble des nœuds : 179/tcp, 4789/udp. Ces informations sont présentes sur la documentation de Calico.

Nous allons utiliser pour l'installation des éléments de Calico un fichier de configuration présent sur le site du projet Calico, appelé Manifest. Toutefois, il y a deux particularités à prendre en compte.
Nous travaillons sur VirtualBox, avec des machines virtuelles possédant 2 interfaces réseau.
Par défaut, Calico utilise la première interface qu'il trouve. Dans notre cas, il s'agit de l'interface NAT qui permet à nos Vms de communiquer avec Internet. Nous voulons que Calico utilise la deuxième interface, celle qui nous sert pour communiquer entre les nœuds du cluster, configuré en réseau privé hôte. Il va donc falloir récupérer les fichiers manifest nécessaires au déploiement de Calico, et modifier le manifest custom-resources.yml pour prendre en compte la bonne interface.

Dans un premier temps, récupérons les 2 fichiers manifests :

```
$ wget -O /tmp/tigera-operator.yml \
https://docs.projectcalico.org/manifests/tigera-operator.yaml
$ wget -O /tmp/custom-resources.yml \
https://docs.projectcalico.org/manifests/custom-resources.yaml
```

Il nous faut modifier le fichier /tmp/custom-resources.yml afin de prendre en compte la bonne interface, mais également de mettre le bon réseau pour nos pods (celui configuré lors du kubeadm init) :

```
$ cat /tmp/custom-resources.yml
...
spec:
  # Configures Calico networking.
  calicoNetwork:
    nodeAddressAutodetectionV4:
      interface: enp0s8
    # Note: The ipPools section cannot be modified post-install.
    ipPools:
    - blockSize: 26
      cidr: 10.244.0.0/16
      encapsulation: VXLANCrossSubnet
      natOutgoing: Enabled
      nodeSelector: all()
...
```

Nous pouvons maintenant appliquer la configuration à notre cluster :

```
$ kubectl create -f /tmp/tigera-operator.yml
$ kubectl create -f /tmp/custom-resources.yml
```

Nous pouvons vérifier l'état de notre cluster :

```
$ kubectl get nodes
NAME     STATUS   ROLES    AGE   VERSION
master   Ready    master   27m   v1.28.2
```

Ainsi que les pods déjà créés :

```
$ kubectl get pods --all-namespaces
NAMESPACE        NAME                                       READY   STATUS    RESTARTS   AGE
calico-apiserver calico-apiserver-74cd6b9d8b-z52rx          1/1     Running   0          28m
calico-system    calico-kube-controllers-6b7b9c649d-p5d6x   1/1     Running   0          28m
calico-system      calico-node-9bgm9                        1/1     Running   0          28m
kube-system   coredns-64897985d-dzfth                       1/1     Running   0          28m
kube-system   coredns-64897985d-flnlz                       1/1     Running   0          28m
kube-system   etcd-master                                   1/1     Running   0          28m
kube-system   kube-apiserver-master                         1/1     Running   0          28m
kube-system   kube-controller-manager-master                1/1     Running   0          28m
kube-system   kube-proxy-gxzdg                              1/1     Running   0          28m
kube-system   kube-scheduler-master                         1/1     Running   0          28m
```

Nous y retrouvons les pods system, nécessaires au fonctionnement de Kubernetes, ainsi que les pods faisant fonctionner Calico. Derrière ces pods, nous avons en réalité des conteneurs qui tournent :

```
$ sudo ctr --namespace k8s.io container list
CONTAINER
IMAGE                                                    RUNTIME
7bf217dfff946fad5b81138944d88018afea7e4186cb7fa3bd5e26116b6b3088
registry.k8s.io/kube-scheduler:v1.28.1                   io.containerd.runc.v2
16a11c205d0de1ca5681d455888cc59380b0c727e64235222267b4087d1c93a2
k8s.gcr.io/pause:3.6                                     io.containerd.runc.v2
9c5e75252445ce4235e2450b8c66bd89e5c71a33d30915fca82b4abdbae5192
docker.io/calico/pod2daemon-flexvol:v3.25.0              io.containerd.runc.v2
47156d7a49f9c778f1e8cecf9faf069b024be047b5934416c65e55ca348f851f
k8s.gcr.io/pause:3.6                                     io.containerd.runc.v2
49e203380f9a5cb4d0292a55b5c3c4c29c6b6d0b1d19b377794d54c68ad538a3
k8s.gcr.io/pause:3.6                                     io.containerd.runc.v2
2c725a05480a670da990c0eec8b0d7bf615d8102c653b16a0e651c5a4db6d216
registry.k8s.io/coredns/coredns:v1.9.3                   io.containerd.runc.v2
cb892fd1619f4c19edb48baaf8878b5cbc92104def75903cb55463d3b2adc9cb
registry.k8s.io/etcd:3.5.6-0                             io.containerd.runc.v2
66e3c07a9e4a4396d5c8554119a7c1ffb07e7dde57bf54fed7d5822878ea017c
k8s.gcr.io/pause:3.6                                     io.containerd.runc.v2
bcaeff90a2438a6a5cf5471a6e6ce05af3dc16c15af61db273f36a1903557937
registry.k8s.io/kube-controller-manager:v1.28.1          io.containerd.runc.v2
512797b465bd0da894e3f19f08ad7db1707dd87d92c80015779cefb86e63af14
registry.k8s.io/kube-apiserver:v1.28.1                   io.containerd.runc.v2
7383e79f2487dc7e9d261301efefc3c7db101fb4858b8376a4f67b4ebbc9f4eb
k8s.gcr.io/pause:3.6                                     io.containerd.runc.v2
74da4d06578f7a91289c420b404ffed9e7b85780430f5e98b4a048a4abe4f12c
k8s.gcr.io/pause:3.6                                     io.containerd.runc.v2
9ccb3a5a997d2df478b0d203040ba0f632cfcc9fffcd36fbe359dc854ab6c3bf
docker.io/calico/cni:v3.25.0                             io.containerd.runc.v2
8b4b0c0f236ebfc76d7a6f9f5fd5d67abd9034b202e1522813e45dd1a78f6959
k8s.gcr.io/pause:3.6                                     io.containerd.runc.v2
a4b6fbf93a61f80c457e27215ba6a089913ddf932a7d6dc45813f92803ea210d
registry.k8s.io/coredns/coredns:v1.9.3                   io.containerd.runc.v2
997e9bf309a8b14ee551743e06aa7b31da63f995d41dd94d6425c66831a86c2f
docker.io/calico/node:v3.25.0                            io.containerd.runc.v2
72f816992698961d36f79a4270d1ee5dc6ab2b34409e463350cc551cd2e8f5ac
registry.k8s.io/kube-proxy:v1.28.1                       io.containerd.runc.v2
. . .
```

Il ne nous reste plus qu'à faire joindre les workers au cluster (commande à lancer sur les workers) :

```
# kubeadm join 192.168.56.31:6443 --token wtm6mn.qny5ybabfqm9vooh \
      --discovery-token-ca-cert-hash
sha256:940e48637695ed60c123f3117709b591c85ede4f0bb9fdf96e1f6bb47ff88699
[preflight] Running pre-flight checks
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get cm
kubeadm-config -o yaml'
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file
"/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Starting the kubelet
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...

This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.
```

Et vérifier que nos workers ont bien rejoint le cluster :

```
$ kubectl get nodes
NAME      STATUS   ROLES           AGE   VERSION
master    Ready    control-plane   30m   v1.28.2
worker1   Ready    <none>          30m   v1.28.2
worker2   Ready    <none>          30m   v1.28.2
```

```
$ kubectl get pods -A -o wide
NAMESPACE          NAME                                       READY STATUS RESTARTS AGE NODE...
calico-apiserver   calico-apiserver-7c65b676f5-8mvvv          1/1   Running 0       13h worker2
calico-apiserver   calico-apiserver-7c65b676f5-x2mb2          1/1   Running 0       13h worker1
calico-system      calico-kube-controllers-b4dc46c6-7grw 1/1   Running 0       13h worker2
calico-system      calico-node-cgfmr                          1/1   Running 0       13h worker2
calico-system      calico-node-nlwx4                          1/1   Running 0       13h master
calico-system      calico-node-vcdcn                          1/1   Running 0       13h worker1
calico-system      calico-typha-5f9f645ccb-7lwfj              1/1   Running 0       13h worker1
calico-system      calico-typha-5f9f645ccb-wghwx              1/1   Running 0       13h worker2
calico-system      csi-node-driver-g8q99                      2/2   Running 0       13h worker2
calico-system      csi-node-driver-n2cq6                      2/2   Running 0       13h worker1
calico-system      csi-node-driver-x5dx5                      2/2   Running 0       13h master
kube-system        coredns-5dd5756b68-6vgml                   1/1   Running 0       13h worker2
kube-system        coredns-5dd5756b68-pn95w                   1/1   Running 0       13h worker2
kube-system        etcd-master                                1/1   Running 0       13h master
kube-system        kube-apiserver-master                      1/1   Running 0       13h master
kube-system        kube-controller-manager-master             1/1   Running 0       13h master
kube-system        kube-proxy-hm54f                           1/1   Running 0       13h master
kube-system        kube-proxy-twszj                           1/1   Running 0       13h worker2
kube-system        kube-proxy-xt6m2                           1/1   Running 0       13h worker1
kube-system        kube-scheduler-master                      1/1   Running 0       13h master
tigera-operator    tigera-operator-94d7f7696-4dxzh            1/1   Running 0       13h worker2
```

Nous avons maintenant un cluster Kubernetes fonctionnel.

Notes

Administration

Dans ce chapitre nous allons prendre en main les outils d'administration.

Administration

- Gestion du cluster en Cli : kubectl, lancement d'un pod, déploiement, autocomplétion
- L'utilisation des fichiers YAML
- Configuration de pods et containers : mémoire, stockage, processeurs, affinité, Namespaces, Scaling
- Les services dans Kubernetes
- Outils de supervision, analyse des logs, debugging

Administration

Gestion du cluster en Cli : kubectl, lancement d'un pod, déploiement, autocomplétion

- kubectl
- Lancement d'un pod
- Déploiement
- Autocomplétion

Gestion du cluster en Cli : kubectl, lancement d'un pod, déploiement, autocomplétion

Kubectl :

La commande kubectl est la commande qui va nous permettre d'interagir avec notre cluster.

Cette commande va en réalité donner des ordres à l'API du cluster, qui va ensuite réaliser les actions demandées.

Cette commande permet de gérer la partie système de notre cluster. Nous pouvons par exemple avoir des informations concernant les nœuds qui composent le cluster :

```
$ kubectl get nodes
NAME      STATUS   ROLES           AGE   VERSION
master    Ready    control-plane   13h   v1.28.2
worker1   Ready    <none>          13h   v1.28.2
worker2   Ready    <none>          13h   v1.28.2
```

Ou de manière plus détaillée :

```
$ kubectl get nodes -o wide
NAME      STATUS   ROLES           AGE   VERSION   INTERNAL-IP     EXTERNAL-IP
OS-IMAGE             KERNEL-VERSION      CONTAINER-RUNTIME
master    Ready    control-plane   13h   v1.28.2   192.168.56.31   <none>
Ubuntu 20.04.4 LTS   5.4.0-107-generic   containerd://1.6.22
worker1   Ready    <none>          13h   v1.28.2   192.168.56.32   <none>
Ubuntu 20.04.4 LTS   5.4.0-107-generic   containerd://1.6.22
worker2   Ready    <none>          13h   v1.28.2   192.168.56.33   <none>
Ubuntu 20.04.4 LTS   5.4.0-107-generic   containerd://1.6.22
```

Nous pouvons également obtenir des informations concernant les pods présents au sein de notre cluster :

```
$ kubectl get pods --all-namespaces
NAMESPACE         NAME                                       READY   STATUS    RESTARTS   AGE
calico-apiserver  calico-apiserver-74cd6b9d8b-4wztv          1/1     Running   2          23h
calico-apiserver  calico-apiserver-74cd6b9d8b-z52rx          1/1     Running   2          23h
calico-system     calico-kube-controllers-6b7b9c649d-p5d6x   1/1     Running   2          23h
calico-system       calico-node-9bgm9                        1/1     Running   2          23h
calico-system       calico-node-lkwm6                        1/1     Running   2          23h
calico-system       calico-node-rgvk6                        1/1     Running   2          23h
calico-system       calico-typha-599d589f47-8r479            1/1     Running   2          23h
calico-system       calico-typha-599d589f47-ksp5c            1/1     Running   2          23h
calico-system       csi-node-driver-j4tbs                    2/2     Running   2          23h
calico-system       csi-node-driver-jpsl5                    2/2     Running   2          23h
calico-system       csi-node-driver-wtdqb                    2/2     Running   2          23h
kube-system         coredns-787d4945fb-c8nd9                 1/1     Running   2          23h
kube-system         coredns-787d4945fb-hljkf                 1/1     Running   2          23h
kube-system         etcd-master                              1/1     Running   2          23h
kube-system         kube-apiserver-master                    1/1     Running   2          23h
kube-system         kube-controller-manager-master           1/1     Running   2          23h
kube-system         kube-proxy-5pkng                         1/1     Running   2          23h
kube-system         kube-proxy-8nlf5                         1/1     Running   2          23h
kube-system         kube-proxy-dpfrd                         1/1     Running   2          23h
kube-system         kube-scheduler-master                    1/1     Running   2          23h
tigera-operator     tigera-operator-54b47459dd-7tlpl         1/1     Running   2          23h
```

L'option --all-namespaces nous permet de voir les pods déployés dans le namespace kube-system. Nous verrons plus tard cette notion mais retenez juste pour l'instant que tant que nous n'avons pas créé nos premiers pods, les seuls présents sont ceux de ce namespace, qui servent au fonctionnement du cluster.

Nous utiliserons cette commande tout au long de ce cours afin d'interagir avec notre cluster.

Lancement d'un pod :

Nous allons maintenant voir comment lancer un pod. Pour rappel, un pod est la plus petite entité existant au sein d'un cluster Kubernetes. Il peut être composé d'un ou plusieurs conteneurs. Nous verrons un peu plus loin que cette méthode n'est pas conseillée pour créer des pods.

Voici comment lancer un pod :

```
$ kubectl run -ti alpine1 --image alpine:latest
If you don't see a command prompt, try pressing enter.
/ # hostname
alpine1
/ #
Session ended, resume using 'kubectl attach alpine1 -c alpine1 -i -t' command when the
pod is running
```

Cette commande a créé un pod nommé alpine1, composé d'un conteneur alpine en version latest, auquel on a attaché un TTY et que l'on a lancé de manière interactive.

Un <Ctrl>+d nous permet de sortir du conteneur. Celui-ci est cependant toujours en cours d'exécution.

Une fois la commande lancée, le pod est créé et lancé sur un des workers, et nous sommes directement attaché au conteneur. La commande hostname nous permet de constater que nous sommes bien à l'intérieur du conteneur.

L'image alpine utilisée a été récupérée depuis le docker hub, comme si nous avions utilisé docker directement.

Comme nous sommes sur un cluster, notre pod a été lancé sur l'un de nos workers. Voici comment savoir sur lequel :

```
$ kubectl get pods -o wide
NAME      READY   STATUS    RESTARTS       AGE   IP             NODE      NOMINATED NODE
READINESS GATES
alpine1   1/1     Running   1(3m54s ago)   57s   10.244.189.75  worker2   <none>
<none>
```

Nous voyons ici que le conteneur a démarré sur worker2. C'est Kubernetes qui décide en interne du nœud sur lequel il va lancer le conteneur. Nous verrons par la suite que nous pouvons interagir sur ce choix.

De la même manière, nous pouvons utiliser -o yaml ou -o json pour obtenir l'ensemble des informations du pod sur ces formats.

Une autre façon intéressante d'afficher les informations et d'utiliser -o custom-columns. Par exemple si nous souhaitons afficher uniquement le nom, le status et le nœud sur lequel tourne un pod, nous pouvons le faire de la manière suivante :

```
$ kubectl get pods -o custom-columns=\
NAME:.metadata.name,STATUS:.status.phase,NODE:.spec.nodeName
NAME      STATUS    NODE
alpine1   Running   worker2
```

L'ensemble des informations récupérables étant disponibles en faisant un -o json ou -o yaml.

Nous pouvons également obtenir de plus amples informations concernant notre pod grâce à la commande :

```
$ kubectl describe pods alpine1
Name:             alpine1
Namespace:        default
Priority:         0
Service Account:  default
Node:             worker1/192.168.56.32
Start Time:       Fri, 17 Feb 2023 15:22:16 +0100
Labels:           run=alpine1
Annotations:      cni.projectcalico.org/containerID:
2e5d5457299888c3f6c67e76fd8fca42a07e70d336cb88539282438932e35551
                  cni.projectcalico.org/podIP: 10.244.235.136/32
                  cni.projectcalico.org/podIPs: 10.244.235.136/32
Status:           Running
IP:               10.244.235.136
IPs:
  IP:  10.244.235.136
Containers:
  alpine1:
    Container ID:   containerd://1bb2cc9e490709b969363936851514db89e7908094421406fd0b9d
    Image:          alpine:latest
    Image ID:       docker.io/library/alpine@sha256:69665d02cb32192e52e07644d76bc6f25ab
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Fri, 17 Feb 2023 15:22:27 +0100
    Last State:     Terminated
      Reason:       Completed
      Exit Code:    0
      Started:      Fri, 17 Feb 2023 15:22:21 +0100
      Finished:     Fri, 17 Feb 2023 15:22:25 +0100
    Ready:          True
    Restart Count:  1
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-z7x85 (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  kube-api-access-z7x85:
    Type:                    Projected (a volume that contains injected data from
multiple sources)
    TokenExpirationSeconds:  3607
    ConfigMapName:           kube-root-ca.crt
    ConfigMapOptional:       <nil>
    DownwardAPI:             true
QoS Class:                   BestEffort
Node-Selectors:              <none>
Tolerations:                 node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                             node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type    Reason     Age                From               Message
  ----    ------     ----               ----               -------
  Normal  Scheduled  <unknown>          default-scheduler  Successfully assigned
default/alpine1 to worker1
  Normal  Pulling    28s (x2 over 35s)  kubelet, worker1   Pulling image "alpine:latest"
  Normal  Pulled     27s (x2 over 34s)  kubelet, worker1   Successfully pulled image
"alpine:latest"
  Normal  Created    27s (x2 over 34s)  kubelet, worker1   Created container alpine1
  Normal  Started    27s (x2 over 34s)  kubelet, worker1   Started container alpine1
```

Cette commande nous permet notamment de voir l'ip affectée par Kubernetes à notre pod, le nœud sur lequel il tourne ou encore la section Events qui retrace les actions effectuées sur le pod. Nous pouvons voir ici qu'il a été programmé, puis l'image a été téléchargée et enfin il a été démarré.

Une autre fonctionnalité intéressante, c'est l'affichage des logs. Il suffit de lancer la commande suivante :

```
$ kubectl logs pod1
```

l'option -f|--follow permet de suivre les logs en direct.

Pour arrêter le pod et le supprimer :

```
$ kubectl get pods
NAME      READY   STATUS    RESTARTS        AGE
alpine1   1/1     Running   1 (4m44s ago)   4m39s

$ kubectl delete pods alpine1
pod "alpine1" deleted

$ kubectl get pods
No resources found in default namespace.
```

A noter que nous n'avons plus besoin de spécifier l'option --all-namespaces pour voir nos pods. En effet, sans autre spécification de notre part, les pods sont créés dans le namespace default, et c'est dans ce namespace que nous interagissons lorsque nous n'en spécifions pas un autre explicitement.

Déploiement :

Nous allons maintenant voir ce qu'est un déploiement :

```
$ kubectl create deploy alpine1 --image alpine:latest
deployment.apps/alpine1 created
```

Nous avons ici créé un déploiement. Nous pouvons nous en rendre compte de la manière suivante :

```
$ kubectl get deployments
AME       READY   UP-TO-DATE   AVAILABLE   AGE
alpine1   0/1     1            0           19s

$ kubectl get pods
NAME                       READY   STATUS             RESTARTS          AGE
alpine1-5ccd998c64-kkmt5   0/1     CrashLoopBackOff   3 (4m55s ago)   94s
```

Ici, ce n'est plus notre pod qui porte le nom alpine1 que nous avions spécifié, mais le déploiement associé. Nous pouvons remarquer que nous avons un pod à l'état CrashLoopBackOff.

Cela est dû au fait que nous n'avons pas spécifié de commande à lancer dans notre conteneur, et que donc celui-ci est lancé et s'arrête instantanément. Nous verrons après plus en détail ce phénomène mais nous allons pour l'instant juste demander à notre déploiement qu'il lance une commande à la création du conteneur afin que celui-ci ne crashe pas.

Pour cela nous allons utiliser la commande suivante :

```
$ kubectl edit deployments.apps alpine1
```

Nous entrons dans la configuration par défaut de notre déploiement, qui stipule un grand nombre de paramètres que nous verrons au fur et à mesure de la formation. Nous allons pour l'instant uniquement modifier à la fin de la configuration dans la partie container, et ajouter le paramètre 'command' à notre conteneur :

```
spec:
      containers:
      - image: alpine:latest
        imagePullPolicy: Always
        name: alpine
        command: ["sleep","600"]
        resources: {}
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
      dnsPolicy: ClusterFirst
      restartPolicy: Always
      schedulerName: default-scheduler
      securityContext: {}
      terminationGracePeriodSeconds: 30
```

Nous demandons donc à ce que le conteneur exécute la commande 'sleep 600', ce qui aura pour effet de laisser le conteneur avec un processus tournant pendant 10 minutes.

Une fois notre configuration sauvegardée, regardons ce qui se passe :

```
$ kubectl get deploy
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
alpine1   1/1     1            1           3m55s

$ kubectl get pods
NAME                       READY   STATUS    RESTARTS   AGE
alpine1-586b468694-b9xm7   1/1     Running   0          26s
```

Cette fois-ci, notre déploiement est READY 1/1, et nous avons bien un pod en status RUNNING.

Nous avons la possibilité de nous connecter au conteneur à l'intérieur du pod :

```
$ kubectl exec -ti alpine1-586b468694-b9xm7 -c alpine -- sh

      # hostname
      alpine1-586b468694-b9xm7
```

Si nous essayons maintenant de supprimer notre pod :

```
$ kubectl delete pods alpine1-586b468694-b9xm7
pod "alpine1-586b468694-b9xm7" deleted

$ kubectl get pods
NAME                       READY   STATUS    RESTARTS   AGE
alpine1-586b468694-jmv7d   1/1     Running   0          36s
```

Nous constatons que Kubernetes a automatiquement recréé un pod, avec un nouvel ID. Cela est dû au déploiement que nous avons créé, plutôt qu'un pod unique. Nous pouvons obtenir plus d'informations concernant ce déploiement :

```
$ kubectl describe deploy alpine1
Name:                   alpine1
Namespace:              default
CreationTimestamp:      Fri, 17 Feb 2023 15:28:32 +0100
Labels:                 app=alpine1
Annotations:            deployment.kubernetes.io/revision: 2
Selector:               app=alpine1
Replicas:               1 desired | 1 updated | 1 total | 1 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=alpine1
  Containers:
   alpine:
    Image:      alpine:latest
    Port:       <none>
    Host Port:  <none>
    Command:
      sleep
      600
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Progressing    True    NewReplicaSetAvailable
  Available      True    MinimumReplicasAvailable
```

```
OldReplicaSets:  <none>
NewReplicaSet:   alpine1-586b468694 (1/1 replicas created)
Events:
  Type    Reason             Age    From                   Message
  ----    ------             ----   ----                   -------
  Normal  ScalingReplicaSet  6m49s  deployment-controller  Scaled up replica set alpine1-
5ccd998c64 to 1
```

Nous retrouvons des informations déjà présentes dans le describe du pod de tout à l'heure, ainsi que de nouvelles informations. Nous avons notamment la partie Replicas, qui spécifie que nous avons demandé au cluster 1 réplica de notre déploiement. Il s'agit en réalité de la valeur par défaut lorsque nous créons un déploiement.

Cela signifie que le cluster essaiera de maintenir cette condition dans la mesure du possible, et donc, qu'il recréera un pod si nous essayons de le supprimer manuellement, afin de satisfaire la condition.

Pour pouvoir supprimer notre pod, il va donc falloir que l'on supprime d'abord le déploiement :

```
$ kubectl get deploy
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
alpine1   1/1     1            1           8m28s
$ kubectl delete deploy alpine1
deployment.apps "alpine1" deleted
$ kubectl get deploy
No resources found in default namespace.
```

Le pod est automatiquement supprimé :

```
$ kubectl get pods
NAME                       READY   STATUS        RESTARTS   AGE
alpine1-586b468694-jmv7d   1/1     Terminating   0          3m30s

$ kubectl get pods
No resources found in default namespace.
```

Autocomplétion :

Nous avons vu que pour pouvoir manipuler notre cluster, il nous faut passer par la commande kubectl, et que les commandes sont souvent très longues. Nous allons nous faciliter la vie en installant la complétion, afin de gagner du temps. Pour cela, il faut vérifier dans un premier temps que le paquet bash-completion est bien installé sur notre cluster :

```
$ dpkg -l|grep bash-completion
ii  bash-completion   1:2.10-1ubuntu1    all   programmable completion for the bash shell
```

Ou l'installer le cas échéant :

```
$ sudo apt install -y bash-completion
```

Il est nécessaire de se délogguer/relogguer pour la prise en compte.

Une fois ceci fait, il va nous falloir installer la complétion de kubectl. Il existe une commande pour cela :

```
$ kubectl completion bash
```

Cette commande va nous permettre de générer les fonctions pour la complétion. Il nous suffit ensuite de l'incorporer dans notre fichier .bashrc, dans notre home directory :

```
$ echo "source <(kubectl completion bash)" >> ~/.bashrc

$ source ~/.bashrc
```

Nous avons maintenant accès à la complétion de kubectl, avec par exemple un kubectl <TAB><TAB> :

```
$ kubectl
alpha          (Commands for features in alpha)
annotate       (Mettre à jour les annotations d'une ressource)
api-resources  (Print the supported API resources on the server)
api-versions   (Print the supported API versions on the server, in the form of
"group/version")
apply          (Apply a configuration to a resource by file name or stdin)
. . .
```

Administration

L'utilisation des fichiers YAML

- Exporter un fichier manifest
- Lancer la création à partir d'un manifest

L'utilisation des fichiers YAML

Nous avons vu comment déployer un pod ou un déploiement à l'aide de la ligne de commande. Toutefois, il existe un très grand nombre d'options possible pour la création de notre pod et surtout de notre déploiement et cela rendra très rapidement la ligne de commande compliquée, notamment à cause de la longueur que celle-ci va rapidement faire. Pour pallier à cela, nous allons maintenant nous intéresser aux fichiers manifests.

Pour faire le parallèle avec docker, ces fichiers ressemblent à des fichiers docker-compose, et vont permettre de définir la configuration que nous souhaitons atteindre et de l'appliquer ensuite à notre cluster.

Exporter un fichier manifest :

Nous allons voir dans un premier temps comment générer un fichier manifest à partir d'une ressource existante :

```
$ kubectl get pods alpine1-c47dbf4fc-tkspz -o yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    cni.projectcalico.org/containerID: 4ac274e02c6fbd07f2ae5e8b44ba5cd18ea5b2e97d9954a2
    cni.projectcalico.org/podIP: 10.244.235.139/32
    cni.projectcalico.org/podIPs: 10.244.235.139/32
  creationTimestamp: "2023-02-17T14:46:46Z"
  generateName: alpine1-586b468694-
```

```
  labels:
    app: alpine1
    pod-template-hash: 586b468694
  name: alpine1-586b468694-vxj5d
  namespace: default
  ownerReferences:
  - apiVersion: apps/v1
    blockOwnerDeletion: true
    controller: true
    kind: ReplicaSet
    name: alpine1-586b468694
    uid: d86c019e-dd17-496b-8525-5e04da194dac
  resourceVersion: "41930"
  uid: 73cecce2-aa2c-4bd3-afe4-557d779ea00c
spec:
  containers:
  - command:
    - sleep
    - "600"
    image: alpine:latest
    imagePullPolicy: Always
    name: alpine
    resources: {}
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: kube-api-access-qq42w
      readOnly: true
  dnsPolicy: ClusterFirst
  enableServiceLinks: true
  nodeName: worker1
  preemptionPolicy: PreemptLowerPriority
  priority: 0
  restartPolicy: Always
  schedulerName: default-scheduler
  securityContext: {}
  serviceAccount: default
  serviceAccountName: default
  terminationGracePeriodSeconds: 30
  tolerations:
  - effect: NoExecute
    key: node.kubernetes.io/not-ready
    operator: Exists
    tolerationSeconds: 300
  - effect: NoExecute
    key: node.kubernetes.io/unreachable
    operator: Exists
    tolerationSeconds: 300
  volumes:
  - name: kube-api-access-qq42w
    projected:
      defaultMode: 420
      sources:
      - serviceAccountToken:
          expirationSeconds: 3607
          path: token
      - configMap:
          items:
          - key: ca.crt
            path: ca.crt
          name: kube-root-ca.crt
      - downwardAPI:
          items:
          - fieldRef:
              apiVersion: v1
              fieldPath: metadata.namespace
            path: namespace
status:
```

```
  conditions:
  - lastProbeTime: null
    lastTransitionTime: "2023-02-17T14:46:42Z"
    status: "True"
    type: Initialized
  - lastProbeTime: null
    lastTransitionTime: "2023-02-17T14:46:45Z"
    status: "True"
    type: Ready
  - lastProbeTime: null
    lastTransitionTime: "2023-02-17T14:46:45Z"
    status: "True"
    type: ContainersReady
  - lastProbeTime: null
    lastTransitionTime: "2023-02-17T14:46:46Z"
    status: "True"
    type: PodScheduled
  containerStatuses:
  - containerID: containerd://369d74c4c53e93098830f7bf8810bf1ac0d828af96ce4c2eb6b
    image: docker.io/library/alpine:latest
    imageID: docker.io/library/alpine@sha256:69665d02cb32192e52e07644d76bc6f25abe
    lastState: {}
    name: alpine
    ready: true
    restartCount: 0
    started: true
    state:
      running:
        startedAt: "2023-02-17T14:46:44Z"
  hostIP: 192.168.56.32
  phase: Running
  podIP: 10.244.235.139
  podIPs:
  - ip: 10.244.235.139
  qosClass: BestEffort
  startTime: "2023-02-17T14:46:42Z"
```

Comme nous pouvons le constater, il y a énormément d'informations, dont bon nombre ont été générées automatiquement. Nous n'allons heureusement pas avoir besoin de renseigner avec autant de précision les fichiers manifests que nous allons créer.

Ce fichier est issu d'une ressource de type pod, mais nous pouvons bien entendu exporter un manifest depuis un déploiement, ou d'autres types de ressources présentes dans Kubernetes que nous verrons plus tard.

Lancer la création à partir d'un manifest :

Nous allons maintenant créer notre propre fichier manifest pour créer un pod nginx1 :

```
$ cat nginx_pod.yml
---
apiVersion: v1
kind: Pod
metadata:
  name: nginx1
spec:
  containers:
  - name: nginx
    image: nginx

$ kubectl apply -f nginx_pod.yml
```

Comme nous pouvons le constater, ce manifest est beaucoup plus petit que le précédent, et nous a néanmoins permis de créer notre pod nginx1, composé d'un conteneur nginx basé sur l'image du même nom.

Nous n'allons pas encore essayer d'accéder au serveur web ainsi créé, il nous manque encore quelques notions que nous verrons plus loin.

Nous pouvons bien entendu créer un manifest composé par exemple de 3 conteneurs au sein du pod :

```
$ cat multi-container.yml
---
apiVersion: v1
kind: Pod
metadata:
  name: multi-containers
spec:
  containers:
  - name: nginx
    image: nginx
  - name: debian
    image: debian
    command: ["sleep","600"]
  - name: alpine
    image: alpine
    command: ["sleep","600"]

$ kubectl apply -f multi-container.yml
pod/multi-containers created
```

Notez la présence d'une nouvelle option, « command », qui va nous permettre de lancer la commande passée au conteneur. Ici, nous avons utilisé la commande sleep pour illustrer la démonstration et empêcher nos conteneurs alpine et debian de s'arrêter immédiatement après leur création, faute de processus à exécuter. Cet usage est similaire à ce que l'on peut retrouver dans les DockerFile.

Nous pouvons ensuite nous connecter aux différents conteneurs de la façon suivante :

```
$ kubectl exec -ti -c nginx multi-containers -- sh

      # hostname
      multi-containers

$ kubectl exec -ti -c debian multi-containers -- sh

      # hostname
      multi-containers
```

Notez que l'ensemble des conteneurs partagent le même hostname.

Nous pouvons bien entendu utiliser les manifest pour créer des déploiements :

```
$ cat deploy_nginx.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mydeploy
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mydeploy
  template:
    metadata:
      labels:
        app: mydeploy
    spec:
      containers:
      - name: nginx
        image: nginx

$ kubectl apply -f deploy_nginx.yml
deployment.apps/mydeploy created

$ kubectl get deployments.apps
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
mydeploy   1/1     1            1           31s

$ kubectl get pods
NAME                         READY   STATUS    RESTARTS   AGE
mydeploy-6b648bdc98-k7srq    1/1     Running   0          3m6s
```

Le fichier manifest est ici un petit peu plus compliqué à appréhender. Nous retrouvons d'abord la partie concernant le déploiement puis le template des conteneurs à créer pour ce déploiement.

Il est obligatoire de définir la partie selector dans le déploiement, ainsi que le label du template, afin que Kubernetes sache quel template utiliser pour le déploiement.

Nous avons également ici spécifié le nombre de réplicas à 1, ce qui signifie que nous voulons une instance de ce déploiement au sein de notre cluster. Nous verrons par la suite la notion de scaling horizontal et l'intérêt de jouer avec cette valeur.

Administration

Configuration de pods et containers : mémoire, stockage, processeurs, affinité, Namespaces,Contextes, Labels, Annotations et Scaling

- Ressources
- NodeSelector et nodeName
- Affinités
- Namespaces, Contextes
- Labels, Annotations
- Scaling

Configuration de pods et containers : mémoire, stockage, processeurs, affinité, Namespaces, Contextes, Labels, Annotations et Scaling

Les Ressources :

Dans Kubernetes, il est possible de contrôler les ressources consommées par nos pods.

Il y a deux concepts dans l'allocation de ressources : le requests et le limit.

- Requests : Cette manière d'allouer les ressources dont le conteneur va avoir besoin au minimum pour fonctionner. En jouant avec cette fonctionnalité, nous allons pouvoir répartir également nos pods sur les conteneurs.
  En effet, en ayant 2 workers avec chacun 2Go de RAM, si j'alloue 1.5Go de RAM à 2 pods différents, ils seront nécessairement démarrés par Kubernetes répartis sur les 2 nœuds.

- Limits : Cette fonctionnalité permet de définir le maximum de ressources qui pourront être consommées par le pod.

Si nous reprenons notre déploiement précédent, nous pouvons inspecter le conteneur créé de la manière suivante :

- Récupération de l'id du conteneur

```
$ kubectl describe pod mydeploy-6686cc75cc-ms2t8
…
Container ID:
containerd://cce3554d7d9f6fe0d7d0e49114c2a2e77d8fd5662a2fbc5d06b5aa29d191d92d
…
```

- Récupération des limites appliquées au conteneur

```
root@worker1:~# sudo ctr --namespace k8s.io container info \
cce3554d7d9f6fe0d7d0e49114c2a2e77d8fd5662a2fbc5d06b5aa29d191d92d | jq \
'.Spec.linux.resources'
{
  "devices": [
    {
      "allow": false,
      "access": "rwm"
    }
  ],
  "memory": {},
  "cpu": {
    "shares": 2,
    "period": 100000
  }
}
```

- Si nous modifions maintenant notre déploiement pour y ajouter une limite en RAM :

```
$ cat deploy_alpine.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mydeploy
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mydeploy
  template:
    metadata:
      labels:
        app: mydeploy
    spec:
      containers:
      - name: alpine
        image: alpine
        command: ["sleep","600"]
        resources:
          requests:
            memory: 50Mi
          limits:
            memory: 100Mi
```

- Et que nous inspectons à nouveau notre conteneur créé :

```
root@worker1:~# sudo ctr --namespace k8s.io container info \
7234d3238e61340c669117d1120ca273e952d1984539f2119cf858f402c94b2e | jq \
'.Spec.linux.resources'
{
  "devices": [
    {
      "allow": false,
      "access": "rwm"
    }
  ],
  "memory": {
    "limit": 104857600
  },
  "cpu": {
    "shares": 2,
    "period": 100000
  }
}
```

Nous constatons à présent que notre limitation en ressources a bien été prise en compte.

Il est également possible d'appliquer des requests et limits sur les ressources CPU. Pour cela, nous utiliserons soit l'unité du type « «0.5 » pour demander un demi-cpu ou encore « 500m », pour 500 milliCPU, ce qui revient au même.

nodeSelector et nodeName :

Dans Kubernetes, il est possible de sélectionner les nœuds sur lesquels nous voulons lancer nos pods.

Il est tout d'abord possible de le faire en utilisant le paramètre nodeName. Prenons le déploiement suivant :

```
$ cat deploy_alpine.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myalpine
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myalpine
  template:
    metadata:
      labels:
        app: myalpine
    spec:
      containers:
      - name: alpine
        image: alpine
        command: ["sleep","600"]
      nodeName: worker1

$ kubectl apply -f deploy_alpine.yml
deployment.apps/myalpine created

$ kubectl get pods -o custom-columns=\
NAME:.metadata.name,STATUS:.status.phase,NODE:.spec.nodeName
NAME                        STATUS    NODE
mydeploy-86cdbb57c9-dlkkq   Running   worker1
```

Nous constatons que le pod créé tourne bien sur worker1.

Nous pouvons également utiliser la directive nodeSelector dans nos fichiers yaml. NodeSelector se base sur des labels. Nous allons par exemple assigner un label sur notre worker1 : podType=alpine, pour spécifier que nous voulons faire tourner nos pods alpine sur ce nœud. Nous aurions pu mettre n'importe quel couple clé/valeur, le tout étant de le reporter correctement sur notre fichier de déploiement.

Partons d'un déploiement d'un pod alpine sans nodeSelector :

```
$ cat deploy_alpine.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myalpine
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myalpine
  template:
```

```
    metadata:
      labels:
        app: myalpine
    spec:
      containers:
      - name: alpine
        image: alpine
        command: ["sleep","600"]

$ kubectl apply -f deploy_alpine.yml
deployment.apps/myalpine created

$ kubectl get pods -o wide
NAME                         READY   STATUS    RESTARTS   AGE   IP          NODE
NOMINATED NODE   READINESS GATES
myalpine-555ccbfff8-xlbfh    1/1     Running   0          69s   10.0.2.4    worker2
<none>           <none>
```

Nous constatons que notre pod tourne sur worker2.

Mettons en place un label sur notre nœud :

```
$ kubectl label nodes worker1 podType=alpine
```

Nous pouvons ensuite vérifier la bonne prise en compte de la manière suivante :

```
$  kubectl get nodes -o custom-columns=\
NAME:.metadata.name,LABELS:.metadata.labels.podType
NAME             LABELS
master           <none>
worker1          alpine
worker2          <none>
```

On constate que nous avons bien défini podType=alpine sur worker1.

A noter que la commande kubectl get nodes --show-labels nous permet également d'accéder à cette information.

Maintenant que notre nœud a le label, nous pouvons relancer un déploiement avec le ficher de configuration suivant :

```
$ cat deploy_alpine.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myalpine
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myalpine
  template:
    metadata:
      labels:
        app: myalpine
    spec:
      containers:
      - name: alpine
        image: alpine
        command: ["sleep","600"]
      nodeSelector:
        podType: alpine

$ kubectl apply -f deploy_alpine.yml
deployment.apps/myalpine configured

$ kubectl get pods -o custom-columns=\
NAME:.metadata.name,STATUS:.status.phase,NODE:.spec.nodeName
NAME                        STATUS    NODE
myalpine-7477d9f565-vlr6l   Running   worker1
```

Nous constatons qu'un nouveau pod a été créé sur worker1, et que le pod sur worker2 a été détruit.

Il est à noter que si nous supprimons le label sur le nœud worker1 et que nous l'appliquons à worker2, cela n'aura pas pour effet de créer un nouveau pod sur worker2 et supprimer l'ancien.

Namespaces, Contextes :

Depuis le début de cette formation, nous avons travaillé sur 2 namespaces. Le namespace kube-system et le namespace default. Kubernetes en crée par défaut un peu plus mais ces deux là sont les plus importants.

Le namespace kube-system est un namespace particulier. Il s'agit du namespace qui va contenir les pods nécessaires au bon fonctionnement de Kubernetes, comme le coredns ou encore Calico.

Le namespace default est, quant à lui, le namespace créé et utilisé de base. C'est ce namespace que nous avons utilisé pour créer nos pods et déploiements.

L'intérêt des namespaces est de pouvoir « isoler » nos pods dans des environnements de travail différents. Nous allons voir comment les créer, les utiliser et les supprimer.

Vérification des namespaces existants :

```
$ kubectl get namespaces
NAME              STATUS   AGE
calico-apiserver   Active   28h
calico-system      Active   28h
default            Active   28h
kube-node-lease    Active   28h
kube-public        Active   28h
kube-system        Active   28h
tigera-operator    Active   28h
```

Création d'un nouveau namespace :

```
$ kubectl create namespace ns1
namespace/ns1 created
```

Une fois le namespace créé, nous pouvons l'utiliser pour effectuer un déploiement :

```
$ cat deploy_alpine.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myalpine
  namespace: ns1
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myalpine
  template:
    metadata:
      labels:
        app: myalpine
    spec:
      containers:
      - name: alpine
        image: alpine
        command: ["sleep","600"]

$ kubectl get deploy
No resources found in default namespace.
```

Comme nous le disions tout à l'heure, le namespace utilisé par défaut est le namespace default.

Kubernetes nous le dit lorsque nous tentons d'afficher les déploiements.

Pour afficher les déploiements d'un namespace particulier, il faut le spécifier :

```
$ kubectl get deploy -n ns1
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
myalpine   1/1     1            1           3m20s
```

Toutes les commandes que nous avons vu précédemment sont applicables dans le namespace en utilisant l'option -n.

Pour supprimer un namespace :

```
$ kubectl delete namespaces ns1
namespace "ns1" deleted
```

Attention, la suppression d'un namespace entraînera la suppression des l'ensemble des pods et des déploiements qu'il contient.

Nous allons maintenant parler des **Contextes**. Les contextes vont nous permettre de définir le profil que nous souhaitons utiliser.

Nous pouvons lister les contextes existants :

```
$ kubectl config get-contexts
CURRENT   NAME                          CLUSTER      AUTHINFO           NAMESPACE
*         kubernetes-admin@kubernetes   kubernetes   kubernetes-admin
```

Voici le profil que nous utilisons par défaut. l'étoile dans le champs CURRENT nous indique le contexte courant. Il est également possible d'obtenir cette information de la façon suivante :

```
$ kubectl config current-context
kubernetes-admin@kubernetes
```

Nous allons maintenant créer notre propre contexte :

```
$ kubectl create ns ns-user
namespace/ns-user created

$ kubectl config set-context contexte1 --namespace ns-user \
                  --user kubernetes-admin --cluster kubernetes
Context "contexte1" created.

$ kubectl config get-contexts
CURRENT   NAME                          CLUSTER      AUTHINFO           NAMESPACE
*         kubernetes-admin@kubernetes   kubernetes   kubernetes-admin
          contexte1                     kubernetes   kubernetes-admin   ns-user
```

Nous avons créé dans un premier temps le namespace ns-user, puis un contexte contexte1 en

spécifiant l'utilisateur kubernetes-admin, le cluster kubernetes et le namespace ns-user.

Nous pouvons maintenant activer le contexte de la manière suivante :

```
$ kubectl config use-context  contexte1
Switched to context "contexte1".

$ kubectl config get-contexts
CURRENT   NAME                          CLUSTER      AUTHINFO           NAMESPACE
          kubernetes-admin@kubernetes   kubernetes   kubernetes-admin
*         contexte1                     kubernetes   kubernetes-admin   ns-user
```

Nous avons désormais basculé sur le contexte contexte1, ce qui nous permet d'exécuter nos commandes kubectl directement dans le namespace associé. Nous n'aurons plus besoin de spécifier l'option -n à chaque fois, ce qui peut s'avérer pratique.

Notez qu'il reste possible de spécifier un namespace différent avec l'option -n.

Labels, Annotations:

Nous allons maintenant nous intéresser aux labels et annotations. Nous avons déjà utilisé les labels lors de la partie sur les nodeSelector et nodeName. Nous allons approfondir un peu le sujet.

Les labels et les annotations peuvent s'appliquer à toutes les ressources.

Les labels vont avoir une utilité particulière dans Kubernetes. Elle va nous servir à marquer nos ressources, et nous utiliserons ensuite les selectors (comme le NodeSelector) pour sélectionner nos ressources labellisées.

Les annotations vont surtout nous permettre de mettre des étiquettes que nous pourrons ensuite utiliser en dehors de Kubernetes.

Nous allons repartir sur de la ligne de commande pour nos exemples. Nous sommes toujours dans le contexte user1 :

```
$ kubectl run myalpine1 --image alpine --labels "env=prod,group=back"  --
sleep 600
pod/myalpine1 created

$ kubectl run myalpine2 --image alpine --labels "env=dev,group=back"  --
sleep 600
pod/myalpine2 created

$ kubectl get pods --show-labels
NAME        READY   STATUS    RESTARTS   AGE   LABELS
myalpine1   1/1     Running   0          65s   env=prod,group=back
myalpine2   1/1     Running   0          58s   env=dev,group=back
```

Nous avons lancé 2 pods composés d'un conteneur alpine, avec des labels.

- Pour le premier env=prod et group=back.
- Pour le deuxième env=dev et group=back.

Nous pouvons ensuite afficher nos pods en mettant les labels avec l'option --show-labels.

Nous pouvons maintenant effectuer des actions sur nos labels :

- Les supprimer :

```
$ kubectl label pod myalpine2 --overwrite group-
pod/myalpine2 labeled

$ kubectl get pods --show-labels
NAME        READY   STATUS    RESTARTS   AGE     LABELS
myalpine1   1/1     Running   0          5m25s   env=prod,group=back
myalpine2   1/1     Running   0          5m18s   env=dev
```

- Les modifier :

```
$ kubectl label pod myalpine2 --overwrite group=front
pod/myalpine2 labeled

$ kubectl get pods --show-labels
NAME        READY   STATUS    RESTARTS   AGE    LABELS
myalpine1   1/1     Running   0          107s   env=prod,group=back
myalpine2   1/1     Running   0          100s   env=dev,group=front
```

- Ou encore en ajouter de nouveaux :

```
$ kubectl get pods --show-labels
NAME        READY   STATUS    RESTARTS   AGE     LABELS
myalpine1   1/1     Running   0          7m51s   app=test,env=prod,group=back
myalpine2   1/1     Running   0          7m44s   env=dev,group=front
```

Notez que l'utilisation de --overwrite n'est nécessaire que si le label existe déjà.

Il est alors possible de sélectionner nos pods à partir de selector :

```
$ kubectl get pod --selector env=prod --show-labels
NAME        READY   STATUS    RESTARTS   AGE   LABELS
myalpine1   1/1     Running   1          16m   app=test,env=prod,group=back

$ kubectl get pod --selector group=back --show-labels
NAME        READY   STATUS    RESTARTS   AGE   LABELS
myalpine1   1/1     Running   1          17m   app=test,env=prod,group=back
myalpine2   1/1     Running   1          17m   env=dev,group=back
```

Concernant les annotations, le fonctionnement est similaire. Il faudra pour cela utiliser la sous-commande annotate au lieu de label. Elle n'est toutefois pas utilisée pour les selectors.

Scaling :

Le scaling va nous permettre de déployer plusieurs instances de nos pods, afin par exemple d'absorber une montée en charge. Il est très simple à mettre en œuvre.

Prenons tout d'abord un déploiement classique d'un pod alpine (nous allons commencer à utiliser les bonnes pratiques et définir un namespace) :

```
$ kubectl create namespace test-scaling
namespace/test-scaling created

$ cat deploy_alpine_scaling.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myalpine
  namespace: test-scaling
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myalpine
  template:
    metadata:
      labels:
        app: myalpine
    spec:
      containers:
      - name: alpine
        image: alpine
        command: ["sleep","600"]

$ kubectl apply -f deploy_alpine_scaling.yml
deployment.apps/myalpine created

$ kubectl get pods -o custom-columns=\
NAME:.metadata.name,STATUS:.status.phase,NODE:.spec.nodeName \
-n test-scaling
NAME                        STATUS    NODE
myalpine-555ccbfff8-k5dzt   Running   worker2
```

Nous constatons que nous avons correctement déployé notre pod, et qu'il tourne actuellement sur worker2.

Nous allons maintenant demander à Kubernetes un scale de 2 sur notre déploiement :

```
$ kubectl scale deployment -n test-scaling myalpine --replicas=2
deployment.apps/myalpine scaled

$ kubectl get pods -o custom-columns=\
NAME:.metadata.name,STATUS:.status.phase,NODE:.spec.nodeName \
-n test-scaling
NAME                        STATUS    NODE
myalpine-555ccbfff8-7bgzb   Running   worker1
myalpine-555ccbfff8-k5dzt   Running   worker2
```

Nous constatons que Kubernetes à bien créé un deuxième pod sur worker1.

Il est bien sur possible de revenir en arrière en remettant le nombre de réplicas à 1, ou encore de remplir directement le nombre de replicas dans notre fichier de configuration yaml avant de l'appliquer.

Lors du scaling, Kubernetes va tenter de faire du load balancing et de répartir équitablement les pods sur les différents workers.

Un autre type de ressource existe dans Kubernetes, le ReplicaSet. Cette ressource permet également de faire du scaling. Toutefois, la ressource deployement englobe ces fonctionnalités et en propose d'autres, que nous avons vu.

Pour reprendre notre exemple précédent de scaling de pods alpine, il aurait fallu d'abord créer notre pod alpine, puis mettre en place la ressource ReplicaSet pour demander à Kubernetes de scaler les pods.

C'est donc pour cette raison que nous avons utilisé la ressource deployement.

Administration

Les services dans Kubernetes

- Qu'est ce qu'un service
- Les différents types de services
- Créer, administrer et supprimer un service

Les services dans Kubernetes

Qu'est ce qu'un service :

Dans docker, pour pouvoir accéder aux applications tournant à l'intérieur de nos conteneurs, il nous faut exposer les ports de ces applications.

Dans Kubernetes, cela est un peu plus compliqué. En effet, les pods peuvent tourner sur différents nœuds, et changer également de nœud au fil du temps. A chaque fois qu'un pod est créé, ou recréé, il va avoir une adresse ip, défini par le CNI, ici Calico.

Il faut donc une solution pour pouvoir accéder à ces applications, et cela va passer par la mise en place de services.

Selon le site Kubernetes.io, un service est une abstraction qui définit un ensemble logique de pods et une politique permettant d'y accéder (parfois ce modèle est appelé un micro-service).

Les différents types de services :

Il existe différents types de services, selon les besoins :

- Nodeport : Il s'agit de faire de l'exposition de port. Cela va nous permettre de rendre public notre pod via un port (par défaut compris entre 30000 et 32767). Le port défini sera exposé sur chacun des nœuds.
- Clusterip : Il s'agit du type par défaut. Il va nous permettre d'exposer le service sur une ip interne du cluster. Elle ne sera donc pas accessible depuis l'extérieur.
- Loadbalancer : Il va nous permettre d'exposer notre service via un contrôleur ingress ou dans le cloud.
- Externalname : Il va nous permettre d'exposer notre service au travers d'une url externe.

Nous allons voir par la suite comment déployer un service clusterip et un nodeport, les 2 autres types de services nécessitant l'utilisation d'outils comme un proxy ou un contrôleur ingress dont nous ne disposons pas.

Créer, administrer et supprimer un service :

Nous allons partir d'un déploiement d'un pod nginx :

```
$ kubectl create namespace test-svc
namespace/test-svc created

$ cat deploy_nginx.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mynginx
  namespace: test-svc
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mynginx
  template:
    metadata:
      labels:
        app: mynginx
    spec:
      containers:
      - name: nginx
        image: nginx
      nodeName: worker2

$ kubectl apply -f deploy_nginx.yml
deployment.apps/mynginx created

$ kubectl get pods -o custom-columns=\
NAME:.metadata.name,STATUS:.status.phase,NODE:.spec.nodeName,\
IP:.status.podIP -n test-svc
NAME                       STATUS    NODE      IP
mynginx-5f77684895-tfq8g   Running   worker2   10.244.189.79
```

Pour rappel, nginx est un serveur web écoutant sur le port 80.

Pour le moment, le seul moyen d'accéder à ce serveur est de récupérer l'ip qui a été affectée au pod :

```
$ curl 10.244.189.79
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
```

```
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Il est possible de lancer le curl depuis l'ensemble des nœuds, workers ou master. Toutefois, si pour une quelconque raison le pod est détruit et reconstruit par Kubernetes, il risque fortement de changer d'ip. Il faudra donc récupérer la nouvelle ip.

Nous allons maintenant créer un service de type clusterIp :

```
$ cat svc_nginx.yml
---
kind: Service
apiVersion: v1
metadata:
  name: nginx-svc
  namespace: test-svc
spec:
  type: ClusterIP
  ports:
  - port: 8080
    targetPort: 80
  selector:
    app: mynginx

$ kubectl apply -f svc_nginx.yml
service/nginx-svc created

$ kubectl get svc -n test-svc
NAME        TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
nginx-svc   ClusterIP   10.101.232.40   <none>        8080/TCP   10s
```

La définition du service n'est pas très compliquée. Nous spécifions le type ClusterIP, nous lui demandons d'exposer le port 8080 sur le port 80 du pod, port d'écoute de nginx.

Nous mettons également en place un selector, qui correspond à la partie matchLabels de notre déploiement, afin que le service s'applique sur celui-ci.

Nous pouvons à présent accéder à notre serveur web de la manière suivante :

```
$ curl 10.101.232.40:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
```

```
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Nous avons maintenant une ip qui est définie au niveau du service, et même si le pod est reconstruit, nous aurons toujours accès à notre serveur web de la même manière.

Cette méthode reste toutefois interne au cluster.

Nous allons maintenant reprendre notre définition du service et le modifier comme suit :

```
$ cat svc_nginx.yml
---
kind: Service
apiVersion: v1
metadata:
  name: nginx-svc
  namespace: test-svc
spec:
  type: NodePort
  ports:
  - nodePort: 30001
    port: 8080
    targetPort: 80
  selector:
    app: mynginx

$ kubectl apply -f svc_nginx.yml
service/nginx-svc configured

$ kubectl get svc -n test-svc
NAME        TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
nginx-svc   NodePort   10.101.232.40   <none>        8080:30001/TCP   12m
```

Notez maintenant le mapping de port entre le port 8080 et le port 30001. Nous avons changé dans la définition du service le type de service, et nous avons ajouté la directive nodePort: 30001.

L'ip interne du service n'a pas changé, et nous avons toujours accès à notre serveur web via cette ip sur le port 8080.

Nous avons par contre maintenant la possibilité d'accéder à notre serveur web depuis l'extérieur du cluster de la manière suivante :

```
$ curl master:30001
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
```

```
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Le serveur web est dorénavant accessible depuis l'ip externe du master, sur le port 30001 que nous avons spécifié. Mais ce n'est pas tout, nous avons aussi la possibilité d'y accéder depuis n'importe quel worker, sur le même port.

Nous pouvons également supprimer un service de la manière suivante :

```
$ kubectl delete svc -n test-svc nginx-svc
service "nginx-svc" deleted
```

Ou encore en supprimant le namespace associé comme vu précédemment, ce qui aura pour effet de supprimer également les déploiements.

Administration

Les DaemonSets, une ressource particulière

- Qu'est ce qu'un DaemonSet
- Déploiement d'un DaemonSet

Les DaemonSets, une ressource particulière

Qu'est ce qu'un DaemonSet :

Nous allons maintenant voir un type de ressource un peu particulier, le DaemonSet. Ce type de ressource permet également de faire du scaling, mais d'une manière un peu particulière.

Nous avons vu précédemment que nous pouvons augmenter/diminuer facilement le nombre de réplicas de nos pods à partir de deployments. Mais avec ce que nous avons vu, nous sommes uniquement capable de donner un nombre de réplicas en fonction de ce que nous souhaitions. Si nous voulions par exemple avoir un pod qui tourne en permanence sur l'ensemble de nos nœuds ?

Nous pourrions fixer le nombre de réplicas à 2. Ainsi, comme nous l'avons vu, Kubernetes va automatiquement nous créer un pod sur chaque worker. Mais si demain, nous ajoutons un worker, il nous faudrait alors modifier notre fichier de configuration et l'appliquer à nouveau pour que la modification soit prise en compte. Et si nous avons un certain nombre de services à scaler de cette manière, cela peut vite devenir fastidieux.

C'est la qu'entre en jeu le DaemonSet. Grâce à cette ressource, nous allons pouvoir demander au cluster d'avoir exactement un pod par nœud.

Déploiement d'un DaemonSet :

Nous allons partir du déploiement d'un daemonSet à partir du fichier de configuration suivant, dans le namespace test-daemon :

```
$ kubectl create ns test-daemon
namespace/test-daemon created

$ cat daemon_set_alpine.yml
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: mydaemonset
  namespace: test-daemon
spec:
  selector:
    matchLabels:
      app: myalpine
  template:
    metadata:
      labels:
        app: myalpine
    spec:
      nodeSelector:
        myDaemonSelector: runDaemonSet
      containers:
      - name: alpine
        image: alpine
        command: ["sleep","600"]


$ kubectl apply -f daemon_set_alpine.yml
daemonset.apps/mydaemonset created

$ kubectl get pods -n test-daemon
No resources found in test-daemon namespace.
```

Veuillez noter l'utilisation de la directive nodeSelector. Elle va nous permettre de définir sur quels nœuds nous voulons que nos pods tournent. Pour l'instant, il n'y a aucun pod de créé.

Mettons maintenant en place le label sur nos nœud :

```
$ kubectl label nodes worker1 myDaemonSelector=runDaemonSet
node/worker1 labeled

$ kubectl get pods -o custom-columns=\
NAME:.metadata.name,STATUS:.status.phase,\
NODE:.spec.nodeName -n test-daemon
NAME                STATUS    NODE
mydaemonset-96zpv   Running   worker1

$ kubectl label nodes worker2 myDaemonSelector=runDaemonSet
node/worker2 labeled

$ kubectl get pods -o custom-columns=\
NAME:.metadata.name,STATUS:.status.phase,\
NODE:.spec.nodeName -n test-daemon
NAME                STATUS    NODE
mydaemonset-hhg2g   Running   worker2
mydaemonset-96zpv   Running   worker1

$ kubectl label nodes worker1 myDaemonSelector-
node/worker1 unlabeled
```

```
$ kubectl get pods -o custom-columns=\
NAME:.metadata.name,STATUS:.status.phase,\
NODE:.spec.nodeName -n test-daemon
NAME                STATUS    NODE
mydaemonset-hhg2g   Running   worker2
```

On remarque bien qu'au fur et à mesure qu'on met les labels sur les nœuds, un pod se crée. Et si l'on retire le label du nœud, le DaemonSet supprime le pod.

A noter que si l'on retire la directive NodeSelector présente dans le fichier de configuration du DaemonSet, il créera automatiquement un pod sur chaque nœud disponible.

Administration

Les Volumes

- HostPath
- EmptyDir
- Persistent volume claim
- Externe
- Les configMaps et les Secrets

Les Volumes

Nous allons voir dans cette partie la gestion des volumes dans Kubernetes. C'est une partie très importante de Kubernetes, elle va nous permettre de faire de la persistance de données.

Il y a 4 types de volumes :

- hostPath
- emptyDir
- persistent volume claim
- externe

Nous allons découvrir comment fonctionne ces différents volumes.

HostPath :

Le hostPath est un volume du host que l'on va monter dans le pod. L'utilisation de ce type de volume est toutefois à prendre avec précaution. En effet, il n'est pas partagé entre les différents nœuds. Il faudra donc s'assurer que le volume que l'on cherche à monter est bien présent sur le nœud sur lequel tourne le pod.

Tout d'abord nous allons créer le répertoire qui nous servira de volume sur worker1 :

```
$ sudo mkdir /srv/data_hostpath
$ sudo chown $(id -u):$(id -g) /srv/data_hostpath/
$ ll -a /srv/data_hostpath/
total 0
drwxr-xr-x. 2 user1 user1  6 May 15 14:57 .
drwxr-xr-x. 3 root  root  27 May 15 14:57 ..
```

Nous pouvons maintenant créer notre déploiement, qui utilisera le volume :

```
$ cat deploy_alpine_hostpath.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myalpine
  namespace: test-hostpath
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myalpine
  template:
    metadata:
      labels:
        app: myalpine
    spec:
      containers:
      - name: myalpine
        image: alpine
        command: ["sleep","600"]
        volumeMounts:
        - mountPath: /root/data
          name: monvolume
      nodeName: worker1
      volumes:
      - name: monvolume
        hostPath:
          path: /srv/data_hostpath
          type: Directory

$ kubectl apply -f deploy_alpine_hostpath.yml
deployment.apps/myalpine created
```

Nous avons ajouté la section volumeMounts à notre conteneur myalpine, en lui spécifiant le point de montage que nous souhaitons à l'intérieur du conteneur et le nom du volume.

Nous avons ensuite déclaré notre volume monvolume, en lui spécifiant le chemin sur le nœud worker1 et le type, ici répertoire.

Il y a également d'autres type de hostPath que nous pouvons utiliser :

- DirectoryOrCreate : Comme Directory, mais le répertoire sera créé s'il n'existe pas.
- File : Permet de monter un fichier.
- FileOrCreate : Comme File, mais le fichier sera créé s'il n'existe pas.
- Socket : Monter un socket Unix.
- CharDevice : Monter un device de type char.
- BlockDevice : Monter un device de type block.

Connectons nous maintenant à notre pod :

```
$ kubectl exec -ti -n test-hostpath myalpine-6d5b6f4897-vsgwl -- sh
      # ls /root/
      data

      # touch /root/data/toto
```

Nous avons bien un répertoire data présent dans /root. Nous allons maintenant vérifier si ce répertoire correspond bien à /srv/data_hostpath sur worker1 :

```
$ ls /srv/data_hostpath/
toto
```

Nous avons correctement monté le répertoire. Veuillez noter que dans la configuration du déploiement, nous avons spécifier la directive nodeName pour être sur que notre pod tournerait sur worker1. Une autre possibilité aurait été de créer le répertoire /srv/data_hostpath également sur worker2.

Par contre les répertoires des 2 nœuds ne seraient pas synchronisés.

EmptyDir :

Nous allons maintenant voir les volumes de type emptyDir. Il va nous permettre de partager un volume éphémère entre conteneurs du même pod.

Nous allons pour cela utiliser le fichier de configuration suivant :

```
$ kubectl create ns test-emptydir
namespace/test-emptydir created

$ cat deploy_emptydir.yml
---
apiVersion: v1
kind: Pod
metadata:
  name: multi-containers
  namespace: test-emptydir
spec:
  containers:
  - name: debian
    image: debian
    command: ["sleep","600"]
    volumeMounts:
    - mountPath: /root/deb
      name: monvolume
  - name: alpine
    image: alpine
    command: ["sleep","600"]
    volumeMounts:
    - mountPath: /root/alp
      name: monvolume

  volumes:
  - name: monvolume
    emptyDir: {}

$ kubectl apply -f deploy_emptydir.yml
pod/multi-containers created
```

Comme pour le hostPath, nous définissons les volumes et les points de montage pour nos conteneurs.

Nous définissons ensuite le volume monvolume, de type emptyDir. Il n'y a pas besoin de créer un répertoire sur nos nœuds cette fois-ci, c'est Kubernetes qui va gérer le volume.

Vérifions maintenant ce qui se passe dans nos conteneurs :

```
$ kubectl exec -n test-emptydir -ti multi-containers -c alpine -- sh

      # ls -la /root/alp/
      total 0
      drwxrwxrwx    2 root     root             6 May 15 13:34 .
      drwx------    1 root     root            37 May 15 13:34 ..

      # touch /root/alp/toto
```

Nous avons bien retrouvé notre volume monté dans /root/alp sur notre conteneur alpine, et avons pu y créer un fichier toto.

Vérifions maintenant sur le deuxième conteneur :

```
$ kubectl exec -n test-emptydir -ti multi-containers -c debian -- sh

      # ls -la /root/deb
      total 0
      drwxrwxrwx. 2 root root 18 May 15 13:35 .
      drwx------. 1 root root 17 May 15 13:33 ..
      -rw-r--r--. 1 root root  0 May 15 13:35 toto
```

Nous avons effectivement notre volume monté dans /root/deb et il contient bien notre fichier toto créé depuis notre conteneur alpine.

Il n'y a pas besoin de se soucier ici d'avoir les volumes créés sur chacun des nœuds, puisque d'une part ils sont gérés par Kubernetes et d'autre part, un pod étant la plus petite entité du cluster, il ne peut se trouver à 2 endroits différents en même temps.

Il est également possible d'utiliser de la RAM pour le emptyDir, Nous n'allons pas refaire l'exemple mais la déclaration du volume serait faite ainsi :

```
$ cat deploy_emptydir.yml
---
apiVersion: v1
kind: Pod
metadata:
  name: multi-containers
  namespace: test-emptydir
spec:
  containers:
  - name: debian
    image: debian
    command: ["sleep","600"]
    volumeMounts:
    - mountPath: /root/deb
      name: monvolume
  - name: alpine
    image: alpine
    command: ["sleep","600"]
    volumeMounts:
    - mountPath: /root/alp
      name: monvolume

  volumes:
  - name: monvolume
    emptyDir:
      medium: Memory
```

Externe :

Voyons maintenant les volumes externes. Kubernetes offre la possibilité d'utiliser des volumes provenant d'autres outils. Il en existe un très grand nombre mais voici les principaux :

- awsElasticBlockStore : Permet d'utiliser un volume EBS Amazon Web Services.
- azureDisk : Permet d'utiliser un disque de données Microsoft Azure.
- cephfs : Permet d'utiliser un volume CephFS.
- fc : Permet d'utiliser une ressource de stockage Fiber Channel.
- iscsi : Permet d'utiliser une ressource de stockage iSCSI.
- nfs : Permet d'utiliser une ressource partagée par un serveur NFS.
- vsphereVolume : Permet d'utiliser un volume vSphere VMDK.
- Et bien d'autres.

Comme vous pouvez le constater, la plupart des providers sont supportés pour l'utilisation des volumes externes.

Persistent Volume Claim :

Les Persistent Volume Claim sont en réalité séparés en 2 parties : Les persistentVolumes et les persistentVolumesClaim. Cette outil que Kubernetes nous met à disposition va nous permettre de faire du provisionning et ensuite consommer ce provisionning.

Comme pour les hostPaths, il faut que le volume existe sur l'ensemble des nœuds. Nous allons donc créer un répertoire dont nous pourrons nous servir sur tous les nœuds :

```
$ sudo mkdir /srv/data_pv
$ sudo chown $(id -u):$(id -g) /srv/data_pv/
```

Nous allons maintenant créer un persistentVolume :

```
$ cat pv_local.yml
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: monpv
  namespace: test-pv
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 1G
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /srv/data_pv

$ kubectl apply -f pv_local.yml
persistentvolume/monpv created
```

Nous déclarons donc un PV dans le namespace test-pv. Il est de type local, ce qui signifie que nous utilisons un stockage en local (le répertoire /srv/data_pv dans le cas présent).

Nous définissons également un storageClassName, qui va nous servir après lorsque nous créerons le persistentVolumeClaim. Nous ne pouvons pas mettre n'importe quelle valeur pour le storageClassName, nous utilisons manual dans ce cas car nous utilisons un répertoire, mais cela dépend du type de volume que nous souhaitons utiliser.

Enfin, nous spécifions la taille du volume, son chemin et le type d'accès. Il existe 3 types :

- ReadWriteOnce : Lecture/Écriture par un seul pod
- ReadOnlyMany : Lecture seule par plusieurs pods
- ReadWriteMany : Lecture/Écriture par plusieurs pods.

Attention pour le dernier type, il faut être sur que le système de fichiers du volume gère l'écriture concurrentielle.

Il est possible d'obtenir des informations sur le PV créé :

```
$ kubectl -n test-pv get pv
NAME   CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS     CLAIM  STORAGECLASS  REASON AGE
monpv  1G        RWO           Retain          Available         manual               16s

$ kubectl -n test-pv describe pv
Name:            monpv
Labels:          type=local
Annotations:     <none>
Finalizers:      [kubernetes.io/pv-protection]
StorageClass:    manual
Status:          Available
Claim:
Reclaim Policy:  Retain
Access Modes:    RWO
VolumeMode:      Filesystem
Capacity:        1G
Node Affinity:   <none>
Message:
Source:
    Type:          HostPath (bare host directory volume)
    Path:          /srv/data_pv
    HostPathType:
Events:            <none>
```

Nous pouvons maintenant créer notre persistentVolumeClaim :

```
$ cat pvc_local.yml
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: monpvc
  namespace: test-pv
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1G

$ kubectl apply -f pvc_local.yml
persistentvolumeclaim/monpvc created
```

Nous déclarons ici un pvc monpvc dans le namespace test-pv. Nous réutilisons le storageClassName défini dans le pv. Sans ça, nous n'aurions pas pu créer le PVC.

Nous avons redéfini également le « access Modes ». Dans ce cas ce n'est pas nécessaire, mais il est possible de surcharger cette directive au niveau du PVC, dans le cas ou l'on souhaiterai réduire les accès uniquement évidemment.

Nous pouvons obtenir des informations sur le PVC créé :

```
$ kubectl get pvc -n test-pv
NAME     STATUS   VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
monpvc   Bound    monpv    1G         RWO            manual         7s
```

```
$ kubectl get pv,pvc -n test-pv
NAME                     CAPACITY ACCESS MODES RECLAIM POLICY  STATUS  CLAIM
STORAGECLASS   REASON   AGE
persistentvolume/monpv  1G       RWO          Retain          Bound   test-pv/monpvc
manual                  7m37s

NAME                           STATUS  VOLUME CAPACITY   ACCESS MODES   STORAGECLASS   AGE
persistentvolumeclaim/monpvc  Bound   monpv  1G         RWO            manual         2m
```

```
$ kubectl describe pv -n test-pv
ame:            monpv
Labels:          type=local
Annotations:     pv.kubernetes.io/bound-by-controller: yes
Finalizers:      [kubernetes.io/pv-protection]
StorageClass:    manual
Status:          Bound
Claim:           test-pv/monpvc
Reclaim Policy:  Retain
Access Modes:    RWO
VolumeMode:      Filesystem
Capacity:        1G
Node Affinity:   <none>
Message:
Source:
    Type:          HostPath (bare host directory volume)
    Path:          /srv/data_pv
    HostPathType:
Events:            <none>
```

```
$ kubectl describe pvc -n test-pv
Name:          monpvc
Namespace:     test-pv
StorageClass:  manual
Status:        Bound
Volume:        monpv
Labels:        <none>
Annotations:   pv.kubernetes.io/bind-completed: yes
               pv.kubernetes.io/bound-by-controller: yes
Finalizers:    [kubernetes.io/pvc-protection]
Capacity:      1G
Access Modes:  RWO
VolumeMode:    Filesystem
Mounted By:    <none>
Events:        <none>
```

Nous pouvons maintenant créer un déploiement en utilisant ce volume :

```
$ cat deploy_alpine_pv.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myalpine
  namespace: test-pv
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myalpine
  template:
    metadata:
      labels:
        app: myalpine
    spec:
      containers:
      - name: myalpine
        image: alpine
        command: ["sleep","600"]
        volumeMounts:
        - mountPath: /root/pv
          name: monvolume
      volumes:
      - name: monvolume
        persistentVolumeClaim:
          claimName: monpvc

$  kubectl apply -f deploy_alpine_pv.yml
deployment.apps/myalpine created
```

Nous avons utilisé la même méthode que pour le hostPath, en spécifiant cette fois-ci un type de volume persistentVolumeClaim et en spécifiant le nom du PVC que nous avons créé.

Vérifions maintenant que notre volume est correctement monté :

```
$ kubectl get pods -o custom-columns=\
NAME:.metadata.name,STATUS:.status.phase,NODE:.spec.nodeName -n test-pv
NAME                        STATUS    NODE
myalpine-78f7c9568b-5p8ck   Running   worker2

$ kubectl -n test-pv exec -ti myalpine-78f7c9568b-5p8ck -- sh
      # ls -la /root/pv
      drwxr-xr-x    2 1000     1000             6 May 15 14:15 .
      drwx------    1 root     root            36 May 15 14:48 ..

      # touch /root/pv/toto
```

Le volume est bien monté dans /root/pv, et nous avons pu y créer un fichier toto.

Nous pouvons maintenant vérifier sur worker2, là où tourne notre pod :

```
$  ls -l /srv/data_pv/
-rw-r--r--. 1 root root 0 May 15 16:50 toto
```

Notre fichier se trouve bien dans /srv/data_pv, là où nous avons déclaré notre PV.

Attention une nouvelle fois, il est bien important de comprendre que le fichier n'est présent que sur le nœud où se trouve le pod, le volume n'étant pas partagé entre les différents nœuds du cluster.

Les ConfigMaps et les Secrets :

Kubernetes proposent une autre fonctionnalité de stockage, les ConfigsMaps et les Secrets.

Ces outils vont nous permettre de centraliser, améliorer et faciliter la gestion de nos configurations.

Les Secrets sont des ConfigMaps encodés en base64.

Nous allons voir comment créer ses ConfigMaps en ligne de commande :

```
$ kubectl create ns config
namespace/config created

$ kubectl create configmap -n config langue --from-literal=LANGUAGE=Fr
configmap/langue created
```

Nous venons de créer un ConfigMap dans le namespace config, ce ConfigMap s'appelle langue et contient LANGUAGE=Fr

Nous pouvons le vérifier :

```
$ kubectl get configmaps -n config
NAME     DATA   AGE
langue   1      105s

$ kubectl describe configmaps -n config langue
Name:         langue
Namespace:    config
Labels:       <none>
Annotations:  <none>

Data
====
LANGUAGE:
----
Fr
Events:  <none>
```

Il est également possible de créer un secret de la même façon :

```
$ kubectl create secret -n config generic password \
                  --from-literal=PASSWORD=myStrongPassword
secret/password created

$ kubectl describe -n config secrets password
Name:         password
Namespace:    config
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
PASSWORD:  16 bytes
```

Il conviendra ensuite de supprimer l'historique des commandes afin d'éviter de laisser le mot de passe en clair à disposition, ou d'utiliser la commande « read -s ».

Comme on peut le voir, le secret n'est pas visible en clair comme le ConfigMap.

Nous pouvons également stocker plusieurs variables dans notre ConfigMap :

```
$ kubectl create -n config configmap langue \
          --from-literal=LANGUAGE=Fr --from-literal=ENCODING=UTF-8
configmap/langue created

$ kubectl describe configmaps -n config langue
Name:         langue
Namespace:    config
Labels:       <none>
Annotations:  <none>

Data
====
ENCODING:
----
UTF-8
LANGUAGE:
----
Fr
Events:  <none>
```

Et il est également possible de stocker des fichiers entiers :

```
$ cat monFic.env
XDG_SESSION_ID=2
HOSTNAME=master
SELINUX_ROLE_REQUESTED=
TERM=xterm-256color
SHELL=/bin/bash
HISTSIZE=1000
SSH_CLIENT=192.168.1.27 58251 22
SELINUX_USE_CURRENT_RANGE=
SSH_TTY=/dev/pts/1
USER=user1
[…]

$ kubectl create -n config configmap environment --from-file=monFic.env
configmap/environment created

$ kubectl describe -n config configmaps environment
Name:         environment
Namespace:    config
Labels:       <none>
Annotations:  <none>

Data
====
monFic.env:
----
XDG_SESSION_ID=2
HOSTNAME=master
SELINUX_ROLE_REQUESTED=
TERM=xterm-256color
SHELL=/bin/bash
HISTSIZE=1000
SSH_CLIENT=192.168.1.27 58251 22
SELINUX_USE_CURRENT_RANGE=
SSH_TTY=/dev/pts/1
USER=user1
[…]
```

Il est possible de modifier le ConfigMap grâce à la commande :

```
$ kubectl edit -n config configmaps langue
```

On entre alors en mode édition et il n'y a plus qu'à faire nos modifications et sauvegarder.

Il est également possible de créer un ConfigMap en partant d'un fichier de configuration :

```
$ cat configMap.yml
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: personne
  namespace: config
data:
  nom: user1
  passion: Kubernetes
  maVariableBlock: |
    monAge: 20
    monAnneeDeNaissance: 2000

$ kubectl apply -f configMap.yml
configmap/personne created

$ kubectl describe -n config configmaps personne
Name:         personne
Namespace:    config
Labels:       <none>
Annotations:
Data
====
nom:
----
user1
passion:
----
Kubernetes
maVariableBlock:
----
monAge: 20
monAnneeDeNaissance: 2000

Events:  <none>
```

Il est également possible de créer des Secrets de la même manière en utilisant le kind: Secret. Il faudra dans ce cas d'abord utiliser la commande base64 pour encoder les mots de passes avant de les stocker dans le fichier. Voici un exemple d'encodage :

```
$ echo -n "password1234" |base64
cGFzc3dvcmQxMjM0
```

Nous allons voir maintenant comment nous servir de ces ConfigMaps.

Il est tout d'abord possible de s'en servir sous forme de variables. Prenons le fichier de configuration suivant :

```
$ cat deploy_alpine.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myalpine
  namespace: config
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myalpine
  template:
    metadata:
      labels:
        app: myalpine
    spec:
      containers:
      - name: alpine
        image: alpine
        command: ["sleep","600"]
        env:
        - name: maLangue
          valueFrom:
            configMapKeyRef:
              name: langue
              key: LANGUAGE
        - name: monMotDePasse
          valueFrom:
            secretKeyRef:
              name: password
              key: PASSWORD
```

Nous passons dans la partie env du conteneur des variables d'environnement grâce aux ConfigMaps et Secrets. Notez qu'il est également possible de passer directement des variables dans la partie env, mais l'utilisation des ConfigMaps est plus élégante et plus appropriée.

Vérifions maintenant dans notre conteneur :

```
$ kubectl exec -n config -ti myalpine-55d5d6db64-c7n79 -- sh
      # env
      KUBERNETES_PORT=tcp://10.96.0.1:443
      KUBERNETES_SERVICE_PORT=443
      HOSTNAME=myalpine-55d5d6db64-c7n79
      SHLVL=1
      HOME=/root
      monMotDePasse=myStrongPassword
      TERM=xterm
      maLangue=Fr
      KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
      PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
      KUBERNETES_PORT_443_TCP_PORT=443
      KUBERNETES_PORT_443_TCP_PROTO=tcp
      KUBERNETES_SERVICE_PORT_HTTPS=443
      KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
      KUBERNETES_SERVICE_HOST=10.96.0.1
      PWD=/
```

Nous retrouvons bien nos variables définies en tant que variables d'environnement.

Il est également possible de passer l'intégralité de notre ConfigMap :

```
$ cat deploy_alpine.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myalpine
  namespace: config
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myalpine
  template:
    metadata:
      labels:
        app: myalpine
    spec:
      containers:
      - name: alpine
        image: alpine
        command: ["sleep","600"]
        envFrom:
        - configMapRef:
            name: environment

$ kubectl apply -f deploy_alpine.yml
deployment.apps/myalpine configured

$ kubectl exec -n config -ti myalpine-54bb9bdc78-l7z55 -- env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=myalpine-54bb9bdc78-l7z55
TERM=xterm
monFic.env=XDG_SESSION_ID=2
HOSTNAME=master
SELINUX_ROLE_REQUESTED=
TERM=xterm-256color
SHELL=/bin/bash
HISTSIZE=1000
SSH_CLIENT=192.168.1.27 58251 22
SELINUX_USE_CURRENT_RANGE=
SSH_TTY=/dev/pts/1
USER=user1
[…]
```

Nous allons maintenant utiliser nos ConfigMaps comme des volumes :

```
$ cat deploy_alpine.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myalpine
  namespace: config
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myalpine
  template:
    metadata:
      labels:
        app: myalpine
    spec:
```

```
    containers:
    - name: alpine
      image: alpine
      command: ["sleep","600"]
      volumeMounts:
      - mountPath: /root/langue
        name: vol-langue
      - mountPath: /root/env
        name: fic-env
      - mountPath: /root/secret
        name: secret-pass
    volumes:
    - name: vol-langue
      configMap:
        name: langue
    - name: fic-env
      configMap:
        name: environment
        items:
        - key: monFic.env
          path: fichier.env
    - name: secret-pass
      secret:
        secretName: password
        items:
        - key: PASSWORD
          path: secret.pass
```

Nous avons effectué plusieurs montages :

Le volume vol-langue : Nous demandons à Kubernetes de monter l'ensemble des données présentes dans le ConfigMap langue dans /root/langue.

Le volume fic-env : Nous demandons cette fois-ci que tout le contenu du fichier monFic.env soit monté dans /root/env, et que le fichier soit appelé fichier.env

Le volume secret-pass : Nous montons notre secret contenant notre mot de passe dans /root/secret, et nous spécifions que le fichier doit s'appeler secret.pass

Vérifions maintenant :

```
$ kubectl exec -n config -ti myalpine-8f5c467d6-jwhmx -- sh
    # apk add --update tree
    # tree /root
    /root
    ├── env
    │   └── fichier.env -> ..data/fichier.env
    ├── langue
    │   ├── ENCODING -> ..data/ENCODING
    │   └── LANGUAGE -> ..data/LANGUAGE
    └── secret
        └── secret.pass -> ..data/secret.pass
```

Nous retrouvons bien nos volumes montés tels que nous l'avons spécifié. Notez qu'il s'agit en réalité de lien symbolique pointant vers des répertoires ..data dans le même répertoire.

Il s'agit de la gestion interne de Kubernetes.

Administration

Outils de supervision, analyse des logs, débogage

- Les nœuds et leur états
- Pods, déploiements et services
- Logs

Outils de supervision, analyse des logs, débogage

Il est très important d'être capable de déboguer un cluster Kubernetes. En effet, au cours de vos tests, vous vous apercevrez que la technologie peut parfois être capricieuse et qu'à certains moments, la moindre erreur de configuration peut nuire au fonctionnement complet du cluster.

C'est ce que nous allons voir dans cette partie.

Les nœuds et leur états :

Il est important de pouvoir connaître l'état de nos nœuds dans Kubernetes, et également savoir chercher des informations concernant ces nœuds. Nous avons déjà vu la commande suivante :

```
$ kubectl get nodes
NAME      STATUS     ROLES           AGE   VERSION
master    Ready      control-plane   30h   v1.28.1
worker1   NotReady   <none>          30h   v1.28.1
worker2   NotReady   <none>          30h   v1.28.1
```

ou associé à l'option -o wide pour plus d'informations, qui nous donne un état général de nos nœuds.

Il est également possible d'obtenir des informations détaillées sur un nœud :

```
$ kubectl describe node master
Name:               master
Roles:              master
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    kubernetes.io/arch=amd64
                    kubernetes.io/hostname=master
                    kubernetes.io/os=linux
                    node-role.kubernetes.io/master=
Annotations:        kubeadm.alpha.kubernetes.io/cri-socket: /var/run/dockershim.sock
                    node.alpha.kubernetes.io/ttl: 0
                    projectcalico.org/IPv4Address: 192.168.1.40/24
                    projectcalico.org/IPv4IPIPTunnelAddr: 10.0.219.64
                    volumes.kubernetes.io/controller-managed-attach-detach: true
[...]
Events:
  Type    Reason                   Age                From                Message
  ----    ------                   ----               ----                -------
  Normal  Starting                 47m                kubelet, master     Starting
kubelet.
  Normal  NodeHasSufficientMemory  47m (x4 over 47m)  kubelet, master     Node master
status is now: NodeHasSufficientMemory
  Normal  NodeHasNoDiskPressure    47m (x4 over 47m)  kubelet, master     Node master
status is now: NodeHasNoDiskPressure
  Normal  NodeHasSufficientPID     47m (x4 over 47m)  kubelet, master     Node master
status is now: NodeHasSufficientPID
  Normal  NodeAllocatableEnforced  47m                kubelet, master     Updated Node
Allocatable limit across pods
  Normal  Starting                 47m                kubelet, master     Starting
kubelet.
  Normal  NodeHasSufficientMemory  47m                kubelet, master     Node master
status is now: NodeHasSufficientMemory
  Normal  NodeHasNoDiskPressure    47m                kubelet, master     Node master
status is now: NodeHasNoDiskPressure
  Normal  NodeHasSufficientPID     47m                kubelet, master     Node master
status is now: NodeHasSufficientPID
  Normal  NodeAllocatableEnforced  47m                kubelet, master     Updated Node
Allocatable limit across pods
  Normal  Starting                 47m                kube-proxy, master  Starting kube-
proxy.
  Normal  NodeReady                43m                kubelet, master     Node master
status is now: NodeReady
```

La dernière partie de cette commande est particulièrement intéressante puisqu'elle nous permet de connaître les différents évènements qui se sont produits sur le nœud.

Une autre manière d'obtenir des informations sur un nœud est d'utiliser la commande suivante :

```
$ kubectl get nodes --output json master
```

L'intérêt de cette dernière est que nous pouvons ensuite parser le résultat via un script par exemple. Il est également possible d'obtenir des informations sur l'état des composants « critiques » de notre cluster :

```
$ kubectl get componentstatuses
NAME                 STATUS    MESSAGE                         ERROR
controller-manager   Healthy   ok
scheduler            Healthy   ok
etcd-0               Healthy   {"health":"true","reason":""}
```

Pods, déploiements et services :

Nous allons maintenant voir comment obtenir des informations sur nos pods et nos services.

Dans un premier temps, nous avons accès à la commande suivante :

```
$ kubectl get pods --all-namespaces
NAMESPACE     NAME                                      READY   STATUS    RESTARTS   AGE
kube-system   calico-kube-controllers-789f6df884-rhhgn  1/1     Running   0          57m
kube-system   calico-node-4vc9z                         1/1     Running   0          57m
kube-system   calico-node-78xqq                         1/1     Running   0          57m
kube-system   calico-node-94j5z                         1/1     Running   0          57m
kube-system   coredns-66bff467f8-gtrcm                  1/1     Running   0          60m
kube-system   coredns-66bff467f8-mm5lz                  1/1     Running   0          60m
kube-system   etcd-master                               1/1     Running   0          60m
kube-system   kube-apiserver-master                     1/1     Running   0          60m
kube-system   kube-controller-manager-master            1/1     Running   1          60m
kube-system   kube-proxy-9vptj                          1/1     Running   0          59m
kube-system   kube-proxy-dh54l                          1/1     Running   0          60m
kube-system   kube-proxy-fddrg                          1/1     Running   0          59m
kube-system   kube-scheduler-master                     1/1     Running   1          60m
```

Nous pouvons également obtenir les pods d'un namespace particulier :

```
$ kubectl get pods -n test-svc
NAME                        READY   STATUS    RESTARTS   AGE
mynginx-5f77684895-cd9qm    1/1     Running   0          6s
```

Cette commande get va également nous permettre d'obtenir des informations sur nos déploiements, et services :

```
$ kubectl get deploy -n test-svc
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
mynginx   1/1     1            1           70s

$ kubectl get svc -n test-svc
NAME        TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
nginx-svc   NodePort   10.96.210.185   <none>        8080:30001/TCP   1s
```

Il est également possible d'obtenir les informations détaillées d'un pod, déploiement ou service :

```
$ kubectl describe svc -n test-svc nginx-svc
Name:                     nginx-svc
Namespace:                test-svc
Labels:                   <none>
Annotations:              Selector:  app=mynginx
Type:                     NodePort
```

```
IP:                       10.96.210.185
Port:                     <unset>  8080/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  30001/TCP
Endpoints:                10.0.189.67:80
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

Cette commande fonctionne de la même manière pour les déploiements et pods.

Logs :

Il est également possible d'afficher les logs de nos pods :

```
$ kubectl logs -n kube-system coredns-66bff467f8-mm5lz
.:53
[INFO] plugin/reload: Running configuration MD5 = 4e235fcc3696966e76816bcd9034ebc7
CoreDNS-1.6.7
linux/amd64, go1.13.6, da7f65b
```

Il est possible d'ajouter -f ou --follow pour suivre les logs. Les logs ne sont visibles que sur les pods. Il n'y a pas de principe de fusionnement des logs par défaut.

Il est également possible de consulter les évènements d'un pod :

```
$ kubectl get events -n test-svc mynginx-5f77684895-\
cd9qm.160f2994e4ee86c5
LAST SEEN   TYPE     REASON    OBJECT                          MESSAGE
14m         Normal   Created   pod/mynginx-5f77684895-cd9qm   Created container nginx
```

Metrics server :

Nous avons également la possibilité d'installer un composant supplémentaire dans Kubernetes, metrics-server. Ce composant, fourni par Kubernetes, va nous permettre deux choses :

- Utiliser la commande kubectl top, afin d'afficher la consommation instantanée des nœuds ou des pods de notre cluster.
- Mettre en place de l'autoscaling.

Voyons tout de suite comment le déployer au sein de notre cluster :

Tout d'abord, nous allons récupérer le manifest des différents objets nécessaires, présent sur le dépôt Github :

```
$ wget \
https://github.com/kubernetes-sigs/metrics-server/releases/latest/\
download/components.yaml
```

Il nous faut ensuite éditer le fichier manifest, afin de rajouter l'option « --kubelete-insecure-tls », dans le cas contraire, il faudrait configurer les certificats au sein de notre cluster :

```
$ vi components.yaml
…
containers:
  - args:
    - --cert-dir=/tmp
    - --secure-port=4443
    - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
    - --kubelet-use-node-status-port
    - --metric-resolution=15s
    - --kubelet-insecure-tls
…
```

Nous pouvons maintenant appliquer la configuration :

```
$ kubectl apply -f components.yaml
```

Une fois le déploiement réalisé, nous avons accès à la commande « kubectl top » :

```
$ kubectl top nodes
NAME      CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
master    86m          4%     1285Mi          68%
worker1   45m          2%     861Mi           45%
worker2   31m          1%     788Mi           41%
```

Nous avons maintenant la possibilité de mettre en place un nouveau type de ressources, les « HorizontalPodAutoscaler ». Cette nouvelle ressource va nous permettre de mettre en place de l'autoscaling.

Voici un exemple d'utilisation :

Création d'un déploiement :

```
$ cat deploy.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: monfront
  namespace: autoscaling
spec:
  replicas: 1
  selector:
    matchLabels:
      app: monfront
  template:
    metadata:
      labels:
        app: monfront
    spec:
      containers:
      - name: monpod
        image: nginx
        resources:
          requests:
            cpu: 10m

$ kubectl apply -f deploy.yml
```

```
$ kubectl get deploy -n autoscaling
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
monfront   1/1     1            1           7s

$ kubectl get pods -n autoscaling
NAME                        READY   STATUS    RESTARTS   AGE
monfront-8654588b9c-bzq99   1/1     Running   0          32s
```

Nous avons pour l'instant un pod issu de notre déploiement.

Création de la ressource hpa :

```
$ cat hpa.yml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: monhpa
  namespace: autoscaling
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: monfront
  minReplicas: 3
  maxReplicas: 10
  targetCPUUtilizationPercentage: 11

$ kubectl apply -f hpa.yml

$ kubectl get hpa -n autoscaling
NAME     REFERENCE             TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
monhpa   Deployment/monfront   0%/11%    3         10        3          42s

$ kubectl get deploy -n autoscaling
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
monfront   3/3     3            3           4m17s

$ kubectl get pods -n autoscaling
NAME                        READY   STATUS    RESTARTS   AGE
monfront-8654588b9c-2j626   1/1     Running   0          108s
monfront-8654588b9c-bzq99   1/1     Running   0          4m44s
monfront-8654588b9c-s76pq   1/1     Running   0          108s
```

Nous pouvons déjà constaté que le nombre de réplicas de notre déploiement à augmenter, pour atteindre le nombre minimum de 3.

Maintenant, essayons d'augmenter la consommation de ressources :

```
$ kubectl exec -ti -n autoscaling monfront-8654588b9c-2j626 -- bash
     # apt update && apt install -y stress
     # stress -c 2
```

Depuis un autre terminal :

```
$ kubectl get hpa -n autoscaling
NAME     REFERENCE             TARGETS     MINPODS   MAXPODS   REPLICAS   AGE
monhpa   Deployment/monfront   1949%/11%   3         10        10         5m7s

$ kubectl get deploy -n autoscaling
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
monfront   10/10   10           10          8m12s
```

```
$ kubectl get pods -n autoscaling
NAME                          READY   STATUS    RESTARTS   AGE
monfront-8654588b9c-2j626     1/1     Running   0          5m38s
monfront-8654588b9c-2xfp8     1/1     Running   0          113s
monfront-8654588b9c-468qv     1/1     Running   0          98s
monfront-8654588b9c-69xmr     1/1     Running   0          98s
monfront-8654588b9c-bzq99     1/1     Running   0          8m34s
monfront-8654588b9c-dhqvv     1/1     Running   0          113s
monfront-8654588b9c-m9sqt     1/1     Running   0          113s
monfront-8654588b9c-s76pq     1/1     Running   0          5m38s
monfront-8654588b9c-sk7z5     1/1     Running   0          98s
monfront-8654588b9c-tltzq     1/1     Running   0          98s
```

Nous pouvons constater que le HPA a automatiquement augmenté le nombre de réplicas, jusqu'à atteindre la limite de 10 replicas.

Une fois la commande stress « killée », et un laps de temps par défaut de 5 min passé (valeur customisable), le HPA va automatiquement scale down, pour repasser au minimum de 3 réplicas.

Notes

Sécurité

Dans ce chapitre nous allons voir la partie sécurité d'un cluster Kubernetes.

Sécurité

Rbac

- Le Role ou clusterRole
- Le serviceAccount
- Le RoleBinding ou ClusterRoleBinding

Rbac

Dans Kubernetes, il y a deux types d'utilisateurs :

- Les comptes système**s** : L'utilisateur système que nous utilisons pour lancer les commandes kubectl, qui nous permet~~tent~~ d'interagir avec l'API de Kubernetes.
- Les comptes applicatifs : Ils vont nous permettre d'interagir avec l'API de Kubernetes au travers de nos pods.

Les comptes applicatifs ont besoin de 3 composants de Kubernetes pour fonctionner :

- le Role ou clusterRole.
- le serviceAccount.
- le RoleBinding ou clusterRoleBinding

Nous allons voir comment mettre en place ces serviceAccount, puis nous verrons dans le chapitre suivant comment les utiliser avec nos pods.

Le Role ou clusterRole :

Le Role ou clusterRole est un profil que nous allons créer. Nous allons définir sur ce profil les accès/actions/ressources auxquels il aura accès.

La différence entre les 2 est que le Role permet de gérer l'ensemble des ressources à l'intérieur d'un namespace, là ou le clusterRole permet d'interagir avec les ressources de tous les namespaces, y compris les nœuds du cluster.

Prenons le fichier de configuration suivant :

```
$ cat role.yml
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: rbac
  name: role-rbac
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]

$ kubectl apply -f role.yml
role.rbac.authorization.k8s.io/role-rbac created

$ kubectl get role -n rbac
NAME        CREATED AT
role-rbac   2023-02-20T12:33:50Z

$ kubectl describe role -n rbac role-rbac
Name:         role-rbac
Labels:       <none>
Annotations:
PolicyRule:
  Resources   Non-Resource URLs  Resource Names  Verbs
  ---------   -----------------  --------------  -----
  pods        []                 []              [get watch list]
```

Nous définissons un rôle role-rbac dans le namespace rbac. Nous lui affectons une règle :

apiGroup : le groupe d'api que nous souhaitons utiliser. En mettant [""], nous voulons utiliser l'api group core, celle où se trouve les opérations qui nous intéresse concernant les pods. Nous pouvons voir les différents apiGroup avec la commande suivante :

```
$ kubectl api-resources
NAME                     SHORTNAMES   APIGROUP                NAMESPACED   KIND
bindings                                                      true         Binding
configmaps               cm                                   true         ConfigMap
endpoints                ep                                   true         Endpoints
events                   ev                                   true         Event
limitranges              limits                               true         LimitRange
namespaces               ns                                   false        Namespace
nodes                    no                                   false        Node
pods                     po                                   true         Pod
podtemplates                                                  true         PodTemplate
resourcequotas           quota                                true         ResourceQuota
secrets                                                       true         Secret
serviceaccounts          sa                                   true         ServiceAccount
services                 svc                                  true         Service
daemonsets               ds           apps                    true         DaemonSet
deployments              deploy       apps                    true         Deployment
replicasets              rs           apps                    true         ReplicaSet
```

```
statefulsets               sts          apps                     true         StatefulSet
tokenreviews                            authentication.k8s.io    false        TokenReview
clusterroles                            rbac.authorization...    false        ClusterRole
rolebindings                            rbac.authorization...    true         RoleBinding
roles                                   rbac.authorization...    true         Role
[…]
```

L'apiGroup core est représentée par un champs vide.

Nous avons donné ici les droits de read, watch et list. Nous verrons tout à l'heure comment les utiliser.

La documentation complète de l'api se trouve ici (adapter la fin de l'url avec votre version de Kubernetes) : https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.28

Elle peut être très pratique pour définir exactement les droits dont nous avons besoin.

Le serviceAccount :

Nous allons maintenant créer notre serviceAccount. C'est le compte applicatif que nous fournirons par la suite à notre pod.

Voici le fichier de configuration que nous allons utiliser :

```
$ cat serviceAccount.yml
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: user-rbac
  namespace: rbac

$ kubectl apply -f serviceAccount.yml
serviceaccount/user-rbac created

$ kubectl get serviceaccounts -n rbac
NAME        SECRETS   AGE
default     0         19m
user-rbac   0         14s
```

Comme vous pouvez le constater, la création de serviceAccount est très simple. Nous définissons un serviceAccount appelé user-rbac dans le namespace rbac.

Nous pouvons encore une fois accéder aux informations concernant cette nouvelle ressource via les sous-commandes get et describe.

Nous pouvons également remarquer qu'un serviceAccount default a été créé, à la création du namespace.

Bien entendu, vu la taille du fichier de configuration, il aurait été possible d'effectuer l'opération directement en ligne de commande :

```
$ kubectl create sa -n rbac user-rbac
serviceaccount/user-rbac created
```

Le RoleBinding ou ClusterRoleBinding :

Nous allons maintenant voir comment créer le dernier composant, le Role Binding ou ClusterRoleBinding.

C'est cette ressource qui va nous permettre de faire le lien entre les 2 autres entités :

```
$ cat bind.yml
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: user-rbac-bind
  namespace: rbac
subjects:
- kind: ServiceAccount
  name: user-rbac
  namespace: rbac
roleRef:
  kind: Role
  name: role-rbac
  apiGroup: rbac.authorization.k8s.io

$ kubectl apply -f bind.yml
rolebinding.rbac.authorization.k8s.io/user-rbac-bind created

$ kubectl get -n rbac rolebindings.rbac.authorization.k8s.io
NAME             ROLE             AGE
user-rbac-bind   Role/role-rbac   4m3s

$ kubectl describe -n rbac rolebindings.rbac.authorization.k8s.io user-rbac-bind
Name:         user-rbac-bind
Labels:       <none>
Annotations:  <none>
Role:
  Kind:       Role
  Name:       role-rbac
Subjects:
  Kind            Name       Namespace
  ----            ----       ---------
  ServiceAccount  user-rbac  rbac
```

Nous demandons donc à déployer un RoleBinding (on aurait utilisé le ClusterRoleBinding si nous souhaitions mapper un ClusterRole à un ServiceAccount).

Nous souhaitons que ce RoleBinding fasse le lien entre notre role role-rbac et notre serviceAccount user-rbac.

Nous pouvons ensuite récupérer des informations sur notre RoleBinding.

Sécurité

Accès à l'API Kubernetes

- Authentification
- Utilisation du ServiceAccount dans un pod

Accès à l'API Kubernetes

Nous allons maintenant voir comment utiliser l'api de Kubernetes. Jusqu'à présent, nous ne nous en sommes servi que grâce à la commande kubectl.

Cette section va nous permettre d'utiliser l'api sans avoir à passer par la commande kubectl. Nous verrons dans un premier temps comment le faire depuis une station de travail.

Puis nous verrons comment utiliser les serviceAccounts que nous avons créé dans le chapitre précédent pour accéder à l'api directement depuis un pod.

Il est possible de se connecter à l'api de Kubernetes depuis le nœud maître, en utilisant la sous-commande proxy.

Cette sous-commande va faire office de reverse-proxy et permettre l'accès à l'api en fonction des options que nous lui donnons :

```
$ kubectl proxy
Starting to serve on 127.0.0.1:8001
```

Cette commande lance l'écoute du proxy sur l'adresse locale et le port 8001.

Nous pouvons ensuite accéder à l'api :

```
$ curl -X GET 127.0.0.1:8001/api/
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "192.168.56.31:6443"
    }
  ]
}
```

Cela ne nous permet toutefois qu'une connexion en local, et qui plus est en http sans authentification. Il est possible d'y accéder à distance et également changer le port d'écoute mais il convient d'être très prudent avec cette méthode et ne l'utiliser que pour des tests, car nous désactivons toutes les sécurités :

```
$ kubectl proxy --address='0.0.0.0' --port='8081' --disable-filter=true
W0519 09:20:47.009430    6062 proxy.go:167] Request filter disabled, your proxy is
vulnerable to XSRF attacks, please be cautious
Starting to serve on [::]:8081
```

Sur un environnement de production, cette méthode n'est pas à privilégier.

Utilisation du ServiceAccount dans un pod :

Nous allons maintenant voir comment utiliser le serviceAccount que nous avons utilisé au chapitre précédent afin d'accéder à l'api de Kubernetes directement depuis un pod.

Partons du fichier de configuration suivant :

```
$ cat deploy_alpine.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myalpine
  namespace: rbac
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myalpine
  template:
    metadata:
      labels:
        app: myalpine
    spec:
      serviceAccountName: user-rbac
      containers:
      - name: alpine
        image: alpine
        command: ["sleep","600"]
        volumeMounts:
        - mountPath: /etc/secrets/tokens
          name: myToken
      volumes:
      - name: myToken
        projected:
          sources:
            - serviceAccountToken:
                path: myToken
                expirationSeconds: 7200
                audience: https://kubernetes.default.svc.cluster.local
            - configMap:
                items:
                  - key: ca.crt
                    path: ca.crt
                name: kube-root-ca.crt

$ kubectl apply -f deploy_alpine.yml
deployment.apps/myalpine created
```

Lorsque nous créons un serviceAccount, un volume projeté contenant un token d'authentification est automatiquement créé. Ce token est automatiquement monté dans les pods utilisant le serviceAccount, dans le répertoire /var/run/secrets/kubernetes.io/serviceaccount, ainsi que le certificat de l'authorité de certification.

Nous déployons un pod avec un conteneur alpine, dans le namespace rbac. Nous avons demandé à monter le volume projeté de notre serviceAccount, non pas dans le répertoire par défaut, mais dans /etc/secrets/tokens. Nous allons retrouver dans ce répertoire à l'intérieur du pod, le token, ainsi que le certificat. Il est à noter que les parties volumes et volumeMounts ne sont pas obligatoires. Nous ne les avons mis que pour choisir ou monter notre token.

Voyons maintenant ce que l'on peut faire depuis le conteneur :

```
$ kubectl exec -ti -n rbac myalpine-6d6b78447f-4cc89 -- sh
# ls /etc/secrets/tokens/
ca.crt     myToken
```

Nous retrouvons 2 fichiers dans notre volume :

- ca.crt : le certificat
- myToken : le token pour l'authentification

Nous pouvons maintenant accéder à l'api de Kubernetes de la manière suivante :

```
# apk add --update curl
fetch https://dl-cdn.alpinelinux.org/alpine/v3.17/main/x86_64/APKINDEX.tar.gz
fetch https://dl-cdn.alpinelinux.org/alpine/v3.17/community/x86_64/APKINDEX.tar.gz
(1/5) Installing ca-certificates (20220614-r4)
. . .
(4/5) Installing libcurl (7.87.0-r2)
(5/5) Installing curl (7.87.0-r2)
Executing busybox-1.35.0-r29.trigger
Executing ca-certificates-20220614-r4.trigger
OK: 9 MiB in 20 packages

# token=$(cat /etc/secrets/tokens/myToken)

# curl -H "Authorization: Bearer $token" --cacert \
/etc/secrets/tokens/ca.crt \
https://kubernetes.default/api/
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "192.168.56.31:6443"
    }
  ]
}
```

Voilà, notre pod a maintenant accès à l'api. Nous avons installé la commande curl dans notre conteneur afin d'effectuer nos tests, puis nous avons récupérer le token et le certificat.

Avec les autorisations que nous avons donné à notre serviceAccount, nous sommes en mesure d'effectuer les appels API suivants :

- https://kubernetes.default/api/v1/namespaces/rbac/pods/ : récupère la liste des pods présents dans le namespace rbac (list).
- https://kubernetes.default/api/v1/namespaces/rbac/pods/$nomPod : récupère les informations d'un pod dans le namespace (get)
- https://kubernetes.default/api/v1/watch/namespaces/rbac/pods/$nomPod : récupère les évènements survenus sur le pod dans le namespace (watch).

Il est bien entendu possible de créer, modifier, supprimer toute ressource via l'api. Afin de trouver et donner les bons droits à vos serviceaccounts, nous vous invitons une nouvelle fois à aller faire un tour sur https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.28.

Sécurité

Limitations des ressources

- Les Quotas, qu'est ce que c'est
- Mise en place de quotas

Limitations des ressources

Les Quotas, qu'est ce que c'est :

Imaginez que vous êtes administrateur d'un cluster Kubernetes sur lequel travaillent plusieurs équipes.

Une bonne pratique serait de séparer les différentes applications à l'aide de namespaces, comme nous l'avons vu.

Mais si l'une de vos équipe s'amuse a tester le déploiement d'un grand nombre de pods, nécessitant énormément de ressources, et que ce déploiement fait planter le cluster ?

C'est là que les quotas entre en jeu. Nous avons déjà vu précédemment comment limiter les ressources à un pod, mais il est possible d'aller plus loin.

Nous allons pouvoir appliquer des quotas pour :

- Limiter la consommation de ressources cumulées sur un espace de noms.
- Limiter la consommation CPU/RAM de manière globale.
- Limiter le nombre d'occurrence d'objets (pods, déploiements, services).

Mise en place des quotas :

Nous n'allons pas directement appliquer les quotas dans notre déploiement comme nous l'avions vu précédemment, nous allons cette fois-ci faire appel à la ressource LimitRange :

```
$ kubectl create ns quotas
namespace/quotas created

$ cat limitRange.yml
---
apiVersion: v1
kind: LimitRange
metadata:
   name: mes-quotas
   namespace: quotas
spec:
   limits:
     - type: Pod
       max:
         cpu: 2
         memory: 300M
     - type: Container
       max:
         cpu: 1
         memory: 240M
       default:
         cpu: 300m
         memory: 200M
       defaultRequest:
         cpu: 200m
         memory: 100M
       maxLimitRequestRatio:
         cpu: 4
         memory: 2

$ kubectl apply -f limitRange.yml
limitrange/mes-quotas created
```

Nous créeons ici la ressource LimitRange mes-quotas, dans le namespace quotas, avec les limitations suivantes :

- 2 CPU et 300 Mo maximum par pod.
- Pour chaque Conteneur :
  - Allocation maximum de 1 CPU et 240 Mo.
  - Limitation maximum par défaut de 30 % de CPU et 200 Mo.
  - Réservation par défaut de 20 % de CPU et 100 Mo.
  - Ratio Maximum de 4 entre la réservation et la limite haute CPU.
  - Ratio de 2 entre la quantité réservée de mémoire et le maximum.

Testons maintenant en lançant la création d'un déploiement :

```
$ cat deploy_alpine.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myalpine
  namespace: quotas
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myalpine
  template:
    metadata:
      labels:
        app: myalpine
    spec:
      containers:
      - name: alpine
        image: alpine
        command: ["sleep","600"]

$ kubectl describe pod -n quotas myalpine-555ccbfff8-lqxvv
Name:         myalpine-555ccbfff8-lqxvv
Namespace:    quotas
[…]
    Limits:
      cpu:     300m
      memory:  200M
    Requests:
      cpu:        200m
      memory:     100M
[…]
```

On constate que les quotas que nous avons demandés tout à l'heure on bien été appliqués à notre déploiement.

Essayons maintenant de dépasser cette limite :

```
$ cat deploy_alpine.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myalpine
  namespace: quotas
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myalpine
  template:
    metadata:
      labels:
        app: myalpine
    spec:
      containers:
      - name: alpine
        image: alpine
        command: ["sleep","600"]
        resources:
          limits:
            cpu: 2
```

```
$ kubectl apply -f deploy_alpine.yml
deployment.apps/myalpine configured
```

```
$ kubectl -n quotas describe replicasets myalpine-76cc8b5b5d
Name:           myalpine-76cc8b5b5d
Namespace:      quotas
[…]
Controlled By:  Deployment/myalpine
[…]
Events:
  Type     Reason        Age                From                   Message
  ----     ------        ----               ----                   -------
  Warning  FailedCreate  2m23s              replicaset-controller  Error creating: pods
"myalpine-76cc8b5b5d-kvdv6" is forbidden: maximum cpu usage per Container is 1, but limit
is 2
```

Les pods concernés par le déploiement n'ont pas pu être déployés car aucun nœud ne leur offrait suffisamment de CPU. En effet, nous avons demandé 2 CPU au maximum, alors que la LimitRange limitait cette valeur à 1.

Nous allons maintenant parler des ResourceQuota. Cette ressource va nous permettre de définir des quotas de manière plus globale :

```
$ cat rsQuota.yml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: my-quota
  namespace: quotas
spec:
  hard:
    pods: 2

$ kubectl apply -f rsQuota.yml
resourcequota/my-quota created
```

Nous venons d'appliquer un quota maximum de 2 pods pour le namespace quotas.

Re-déployons notre pod alpine :

```
$ cat deploy_alpine.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myalpine
  namespace: quotas
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myalpine
  template:
    metadata:
      labels:
        app: myalpine
    spec:
      containers:
      - name: alpine
        image: alpine
        command: ["sleep","600"]
```

Et essayons maintenant d'effectuer un scale de 3 sur notre déploiement :

```
$ kubectl scale deployment -n quotas --replicas=3 myalpine
deployment.apps/myalpine scaled

$ kubectl get all -n quotas
NAME                            READY   STATUS    RESTARTS   AGE
pod/myalpine-5bf7bff55b-cgt72   1/1     Running   0          77s
pod/myalpine-5bf7bff55b-qb27v   1/1     Running   0          42s

NAME                        READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/myalpine    2/3     2            2           77s

NAME                                   DESIRED   CURRENT   READY   AGE
replicaset.apps/myalpine-5bf7bff55b    3         2         2       77s

$ kubectl describe rs -n quotas myalpine-5bf7bff55b
Name:           myalpine-5bf7bff55b
Namespace:      quotas
[…]
Controlled By:  Deployment/myalpine
[…]
Events:
  Type     Reason            Age         From                   Message
  ----     ------            ----        ----                   -------
  Normal   SuccessfulCreate  2m34s       replicaset-controller  Created pod: myalpine-
5bf7bff55b-cgt72
  Warning  FailedCreate      2m20s       replicaset-controller  Error creating: pods
"myalpine-5bf7bff55b-v29bw" is forbidden: exceeded quota: my-quota, requested: pods=1,
used: pods=2, limited: pods=2
  Warning  FailedCreate      2m20s       replicaset-controller  Error creating: pods
"myalpine-5bf7bff55b-t6bk6" is forbidden: exceeded quota: my-quota, requested: pods=1,
used: pods=2, limited: pods=2
  Warning  FailedCreate      2m20s       replicaset-controller  Error creating: pods
"myalpine-5bf7bff55b-x9bf2" is forbidden: exceeded quota: my-quota, requested: pods=1,
used: pods=2, limited: pods=2
  Warning  FailedCreate      2m20s       replicaset-controller  Error creating: pods
"myalpine-5bf7bff55b-nkdpj" is forbidden: exceeded quota: my-quota, requested: pods=1,
used: pods=2, limited: pods=2
[…]
```

Nous voyons que le déploiement n'a créé que 2 pods sur 3 et, quand nous jetons un coup d'oeil à l'objet ReplicaSet associé, nous voyons que nous avons dépassé le quota et que nous ne pouvons créer le 3eme pod.

Ces quotas peuvent être appliqués à d'autres types de ressources comme les CPUs, la mémoire ou encore les ressources graphiques si les serveurs composants le cluster Kubernetes en disposent.

Je vous invite à aller voir le lien suivant : https://kubernetes.io/docs/concepts/policy/resource-quotas/#enabling-resource-quota si vous souhaitez plus d'informations à ce sujet.

Sécurité

Contrôle des accès réseau

- Connexions entrantes
- Connexions sortantes

Contrôle des accès réseau

Nous avons installé au début de cette formation un CNI (Container Network Interface) afin de gérer la partie réseau de notre cluster.

Ce CNI nous bloque par défaut les connexions entrantes sur notre Cluster. Nous ne pouvons en effet pas interroger depuis l'extérieur les adresses Ips que Calico donne à nos pods lors de leur déploiement.

En revanche, les accès entre pods et les connexions sortantes ne posent aucun problème. Nous allons voir ici comment limiter ses connexions.

Les Network Policies agissent comme un pare-feu, ce qui peut entrer en conflit avec un pare-feu système déjà présent. Ce n'est pas le cas pour nous, mais dans le cas contraire, il faudrait penser à le désactiver.

Connexions entrantes :

commençons par lancer 3 pods :

```
$ kubectl create ns network1
namespace/network1 created

$ kubectl create ns network2
namespace/network2 created

$ kubectl run myalpine1 --image alpine -n network1 -- sleep 600
$ kubectl run myalpine2 --image alpine -n network1 -- sleep 600
$ kubectl run myalpine3 --image alpine -n network2 -- sleep 600

$ kubectl get pods -o custom-columns=NAME:.metadata.name,\
IP:.status.podIP -n network1
NAME        IP
myalpine1   10.0.235.179
myalpine2   10.0.235.180

$ kubectl get pods -o custom-columns=NAME:.metadata.name,\
IP:.status.podIP -n network2
NAME        IP
myalpine3   10.0.189.74
$ kubectl exec -ti -n network1 myalpine1 -- ping -c 1 10.0.235.180
PING 10.0.235.180 (10.0.235.180): 56 data bytes
64 bytes from 10.0.235.180: seq=0 ttl=63 time=0.101 ms
[…]
$ kubectl exec -ti -n network1 myalpine1 -- ping -c 1 10.0.189.74
PING 10.0.189.74 (10.0.189.74): 56 data bytes
64 bytes from 10.0.189.74: seq=0 ttl=62 time=0.759 ms
[…]
$ kubectl exec -ti -n network1 myalpine2 -- ping -c 1 10.0.235.179
PING 10.0.235.179 (10.0.235.179): 56 data bytes
64 bytes from 10.0.235.179: seq=0 ttl=63 time=0.100 ms
[…]
$ kubectl exec -ti -n network1 myalpine2 -- ping -c 1 10.0.189.74
PING 10.0.189.74 (10.0.189.74): 56 data bytes
64 bytes from 10.0.189.74: seq=0 ttl=62 time=0.492 ms
[…]
$ kubectl exec -ti -n network2 myalpine3 -- ping -c 1 10.0.235.179
PING 10.0.235.179 (10.0.235.179): 56 data bytes
64 bytes from 10.0.235.179: seq=0 ttl=62 time=0.467 ms
[…]
$ kubectl exec -ti -n network2 myalpine3 -- ping -c 1 10.0.235.180
PING 10.0.235.180 (10.0.235.180): 56 data bytes
64 bytes from 10.0.235.180: seq=0 ttl=62 time=0.495 ms
[…]
```

Nos pods arrivent à communiquer entre eux sans difficulté.

Nous allons maintenant créer une nouvelle ressource, une NetworkPolicy :

```
$ cat netPolicy.yml
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: ingress-policy
  namespace: network1
spec:
  podSelector: {}
  policyTypes: ["Ingress"]
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          auth: ingress
```

Nous définissons une règle interdisant le trafic entrant à tous les pods du namespace network1, sauf provenant de pods d'un namespace ayant le label auth=ingress.

Vérifions :

```
$ kubectl exec -ti -n network2 myalpine3 -- ping -c 1 10.0.235.179
PING 10.0.235.179 (10.0.235.179): 56 data bytes

--- 10.0.235.179 ping statistics ---
1 packets transmitted, 0 packets received, 100% packet loss
command terminated with exit code 1

$ kubectl exec -ti -n network1 myalpine2 -- ping -c 1 10.0.235.179
PING 10.0.235.179 (10.0.235.179): 56 data bytes

--- 10.0.235.179 ping statistics ---
1 packets transmitted, 0 packets received, 100% packet loss
command terminated with exit code 1
```

Nous voyons que myalpine3 dans le namespace network2 n'a pas accès aux pods du network1, et que les pods du network1 n'ont pas accès entre eux.

Plaçons maintenant un label sur le namespace network2 :

```
$ kubectl label ns network2 auth=ingress
namespace/network2 labeled

$ kubectl exec -ti -n network2 myalpine3 -- ping -c 1 10.0.235.180
PING 10.0.235.180 (10.0.235.180): 56 data bytes
64 bytes from 10.0.235.180: seq=0 ttl=62 time=0.692 ms

--- 10.0.235.180 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.692/0.692/0.692 ms
```

Ca y est, notre pod myalpine3, du namespace network2, a maintenant accès aux pods du network1.

Il est également possible d'autoriser les flux entrants venants de pods du même namespace. Prenons l'exemple suivant :

```
$ cat netPolicyNs.yml
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: ingress-policy-ns
  namespace: network1
spec:
  podSelector: {}
  policyTypes: ["Ingress"]
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: db

$ kubectl apply -f netPolicyNs.yml
networkpolicy.networking.k8s.io/ingress-policy-ns created
```

Nous demandons à ouvrir les flux à tous les pods qui possèdent un label app=db vers l'ensemble des pods du namespace network1.

Vérifions :

```
$ kubectl label pod -n network1 myalpine1 app=db
pod/myalpine1 labeled

$ kubectl exec -ti -n network1 myalpine1 -- ping -c 1 10.0.235.180
PING 10.0.235.180 (10.0.235.180): 56 data bytes
64 bytes from 10.0.235.180: seq=0 ttl=63 time=0.129 ms

--- 10.0.235.180 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.129/0.129/0.129 ms

$ kubectl exec -ti -n network1 myalpine2 -- ping -c 1 10.0.235.179
PING 10.0.235.179 (10.0.235.179): 56 data bytes

--- 10.0.235.179 ping statistics ---
1 packets transmitted, 0 packets received, 100% packet loss
command terminated with exit code 1
```

Nous avons bien ouvert les flux de myalpine1 vers myalpine2. L'inverse en revanche est toujours bloqué.

Connexions sortantes :

Nous venons de voir comment mettre en place des polices réseaux pour les connexions entrantes, mais il est également possible d'en mettre en place pour les connexions sortantes. Nous allons pour cela utiliser les polices « egress » :

```
$ cat denyAllEgress.yml
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: egress-policy-deny
  namespace: network2
spec:
  podSelector: {}
  policyTypes: ["Egress"]
  egress:
  - to:
    ports:
    - protocol: TCP
      port: 53
    - protocol: UDP
      port: 53

$ kubectl apply -f denyAllEgress.yml
networkpolicy.networking.k8s.io/egress-policy-deny created

$ kubectl exec -ti -n network2 myalpine3 -- nslookup google.fr
Server:         10.96.0.10
Address:        10.96.0.10:53

Non-authoritative answer:
Name:   google.fr
Address: 216.58.215.35

Non-authoritative answer:
Name:   google.fr
Address: 2a00:1450:4007:815::2003

$ kubectl exec -ti -n network2 myalpine3 -- ping -c1 google.fr
PING google.fr (216.58.215.35): 56 data bytes

--- google.fr ping statistics ---
1 packets transmitted, 0 packets received, 100% packet loss
command terminated with exit code 1

$ kubectl exec -ti -n network1 myalpine1 -- ping -c1 google.fr
PING google.fr (216.58.215.35): 56 data bytes
64 bytes from 216.58.215.35: seq=0 ttl=52 time=2.972 ms

--- google.fr ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 2.972/2.972/2.972 ms
```

Nous avons créé une règle qui interdit les flux sortants pour tous les pods du namespace network2, excepté en TCP/UDP sur le port 53 pour conserver les résolutions DNS.

On voit bien que les résolutions DNS sur myalpine3 continuent à fonctionner, mais il n'est pas possible en revanche de lancer un ping sur une ip externe, comme nous pouvons encore le faire sur les pods du namespace network1.

Il est toujours possible de mettre en place des selectors comme pour les règles ingress :

```
$ cat denyAllEgress.yml
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: egress-policy-deny
  namespace: network1
spec:
  podSelector: {}
  policyTypes: ["Egress"]
  egress:
  - to:
    ports:
    - protocol: TCP
      port: 53
    - protocol: UDP
      port: 53

$ cat allowEgressFront.yml
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-egress-front
  namespace: network1
spec:
  podSelector:
    matchLabels:
      app: front
  egress:
  - {}
  policyTypes:
  - Egress

$ kubectl apply -f denyAllEgress.yml
networkpolicy.networking.k8s.io/egress-policy-deny created

$ kubectl apply -f allowEgressFront.yml
networkpolicy.networking.k8s.io/allow-egress-front created

$ kubectl get pods -n network1 --show-labels
NAME        READY   STATUS    RESTARTS   AGE   LABELS
myalpine1   1/1     Running   12         14h   app=db,run=myalpine1
myalpine2   1/1     Running   12         14h   app=front,run=myalpine2

$ kubectl exec -ti -n network1 myalpine1 -- ping -c1 google.fr
PING google.fr (216.58.206.227): 56 data bytes

--- google.fr ping statistics ---
1 packets transmitted, 0 packets received, 100% packet loss
command terminated with exit code 1

$ kubectl exec -ti -n network1 myalpine2 -- ping -c1 google.fr
PING google.fr (216.58.206.227): 56 data bytes
64 bytes from 216.58.206.227: seq=0 ttl=52 time=2.877 ms

--- google.fr ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 2.877/2.877/2.877 ms
```

Nous définissons la même règle de flux sortants que tout à l'heure sur le namespace network1, interdisant toute connexion sortante sauf sur le port 53 UDP/TCP.

Nous définissons ensuite une seconde règle autorisant tous les flux sortants sur les pods portant le label app: front. En testant, nous voyons que myalpine1 fait bien une résolution de google.fr mais n'arrive pas à le contacter.

Myalpine2, en revanche, y accède sans souci grâce au label app: front qu'il possède.

Cela nous a permis de voir également que l'on pouvait définir plusieurs règles, et que celles-ci peuvent être complémentaires.

Il est aussi possible de mettre en place à la fois des règles ingress et egress, et même de combiner ces règles dans le même fichier de configuration.

Notes

Aller Plus Loin

Dans ce chapitre nous allons voir des outils et bonnes pratiques.

Aller plus loin

- Les Healthchecks
- Les Déploiements
- Les Jobs
- Statefull / Stateless

Aller plus loin

Les Healthchecks

- Liveness
- Readiness

Les Healthchecks

Depuis le début de cette formation, nous avons vu comment orchestrer le déploiement d'applications conteneurisées.

Pour cela, nous nous basons sur des images la plupart du temps déjà construites. Cependant, ces images ne sont pas toujours parfaites, et parfois, il nous faut de temps en temps redémarrer l'application pour telle ou telle raison, afin de continuer à la faire fonctionner.

De plus, il arrive que les images que nous tentons de déployer ne soient pas fonctionnelles, les techniques que nous allons voir vont donc également nous aider à éviter l'ajout dans notre cluster d'applications défectueuses.

Liveness :

Le principe de liveness est simple : Il va nous permettre de redémarrer automatiquement un pod, en se basant sur un test.

Ce test, c'est l'administrateur qui va le mettre en place. Il peut être de différents types :

- Surveiller la présence d'un port d'écoute.
- Surveiller la présence d'un fichier.
- Réaliser une connexion HTTP.
- Se connecter à une base de données.
- Faire appel à une page de diagnostic.

Pour utiliser la fonctionnalité liveness, nous allons nous servir de la directive livenessProbe.

Partons de la configuration suivante :

```
$ kubectl create ns test-liveness
namespace/test-liveness created

$ cat deploy_alpine_liveness.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myalpine
  namespace: test-liveness
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myalpine
  template:
    metadata:
      labels:
        app: myalpine
    spec:
      containers:
      - name: alpine
        image: alpine
        command: ["/bin/sh"]
        args: ["-c","touch /tmp/test_liveness;sleep 30;rm -rf /tmp/test_liveness; sleep
600"]
        livenessProbe:
          exec:
            command: ["cat","/tmp/test_liveness"]
          initialDelaySeconds: 5
          periodSeconds: 5
```

Nous avons un pod avec un conteneur alpine dans le namespace test-liveness. Ce conteneur va créer le fichier /tmp/test_liveness lors de son déploiement, va attendre 30 secondes puis le supprimer. Il attendra ensuite 10 minutes.

Nous avons ensuite un test livenessProbe qui va tenter de faire un cat du fichier test_liveness.
Nous lui avons passé deux options :

- InitialDelaySeconds : Nombre de secondes avant de lancer le premier check.
- PeriodSeconds : Fréquence des checks.

Il y a également d'autres options possibles :

- timeoutSeconds : A partir de combien de secondes sans réponse du check on considère que l'on est en timeout (1 seconde par défaut).
- SuccessThreshold : Au bout de combien de tests positifs on considère que le conteneur est en bonne santé (1 par défaut).
- FailureThreshold : Au bout de combien de tests négatifs on considère que le conteneur est en mauvaise santé (3 par défaut).

Nous pouvons maintenant lancer le déploiement et analyser ce qui se passe :

```
$ kubectl apply -f deploy_alpine_liveness.yml
deployment.apps/myalpine created

$ kubectl get pods -n test-liveness
NAME                        READY   STATUS    RESTARTS   AGE
myalpine-6bb7bc7ff7-4rsbw   1/1     Running   0          5s

$ kubectl describe pods -n test-liveness myalpine-6bb7bc7ff7-4rsbw
Name:             myalpine-6bb7bc7ff7-4rsbw
Namespace:        test-liveness
Priority:         0
Service Account:  default
Node:             worker2/192.168.56.33
Start Time:       Mon, 20 Feb 2023 14:24:12 +0100
Labels:           app=myalpine
                  pod-template-hash=6bb7bc7ff7
Annotations:      cni.projectcalico.org/containerID:
7dfc02d2ae30f8f85783f8b56a563b4da770fda96079019117e7ea6d4bda60ac
                  cni.projectcalico.org/podIP: 10.244.189.87/32
                  cni.projectcalico.org/podIPs: 10.244.189.87/32
Status:           Running
IP:               10.244.189.87
IPs:
  IP:           10.244.189.87
Controlled By:  ReplicaSet/myalpine-6bb7bc7ff7
Containers:
  alpine:
    Container ID:
containerd://7751eaae54a039648affe339dd17bace89fe661b628d5e2c6d5d2788e6653a82
    Image:         alpine
    Image ID:
docker.io/library/alpine@sha256:69665d02cb32192e52e07644d76bc6f25abeb5410edc1c7a81a10ba3f
0efb90a
    Port:          <none>
    Host Port:     <none>
    Command:
      /bin/sh
    Args:
      -c
      touch /tmp/test_liveness;sleep 30;rm -rf /tmp/test_liveness; sleep 600
```

```
    State:          Running
      Started:      Mon, 20 Feb 2023 14:25:29 +0100
    Last State:     Terminated
      Reason:       Error
      Exit Code:    137
      Started:      Mon, 20 Feb 2023 14:24:15 +0100
      Finished:     Mon, 20 Feb 2023 14:25:28 +0100
    Ready:          True
    Restart Count:  1
    Liveness:       exec [cat /tmp/test_liveness] delay=5s timeout=1s period=5s
#success=1 #failure=3
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-gdpnn (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  kube-api-access-gdpnn:
    Type:                     Projected (a volume that contains injected data from
multiple sources)
    TokenExpirationSeconds:  3607
    ConfigMapName:           kube-root-ca.crt
    ConfigMapOptional:       <nil>
    DownwardAPI:             true
QoS Class:                   BestEffort
Node-Selectors:              <none>
Tolerations:                 node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                             node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type     Reason     Age                  From               Message
  ----     ------     ----                 ----               -------
  Normal   Scheduled  2m17s                default-scheduler  Successfully assigned test-
liveness/myalpine-6bb7bc7ff7-4rsbw to worker2
  Normal   Pulled     2m11s                kubelet            Successfully pulled image
"alpine" in 1.11451623s (1.115305716s including waiting)
  Normal   Pulling    57s (x2 over 2m12s)  kubelet            Pulling image "alpine"
  Normal   Created    56s (x2 over 2m10s)  kubelet            Created container alpine
  Normal   Started    56s (x2 over 2m10s)  kubelet            Started container alpine
  Normal   Pulled     56s                  kubelet            Successfully pulled image
"alpine" in 850.332374ms (850.559219ms including waiting)
  Warning  Unhealthy  12s (x6 over 97s)    kubelet            Liveness probe failed: cat:
can't open '/tmp/test_liveness': No such file or directory
  Normal   Killing    12s (x2 over 87s)    kubelet            Container alpine failed
liveness probe, will be restarted

$ kubectl get pods -n test-liveness
NAME                         READY   STATUS    RESTARTS         AGE
myalpine-6bb7bc7ff7-4rsbw   1/1     Running   1 (30s ago)      110s
```

Une fois le déploiement lancé, nous constatons que le pod est correctement créé. Puis, dans la partie events, nous voyons que les probes liveness commencent à échouer (le fichier est supprimé). Au bout de 3 échecs (valeur par défaut du failureThreshold), le pod est redémarré. On peut le constater à la dernière ligne dans la partie RESTARTS.

Readiness :

Le readiness, défini par la directive readinessProbe, va nous permettre de savoir si notre pod est dans le bon état. Est-il correctement démarré ? Toutes ses dépendances sont-elles prêtes ?

Contrairement au liveness qui ne va faire que redémarrer le conteneur en cas de fail, le readiness va lui rendre indisponible le conteneur. Cela signifie qu'il ne sera plus considéré dans le pool de conteneurs prêts tant que la condition ne sera pas de nouveau satisfaite.

Reprenons notre déploiement de conteneur alpine en lui affectant une close readiness :

```
$ kubectl create ns test-readiness
namespace/test-readiness created

$cat deploy_alpine_readiness.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myalpine
  namespace: test-readiness
spec:
  replicas: 1
  selector:
    matchLabels:
      app: myalpine
  template:
    metadata:
      labels:
        app: myalpine
    spec:
      containers:
      - name: alpine
        image: alpine
        command: ["/bin/sh"]
        args: ["-c","touch /tmp/test_readiness;sleep 30;rm -rf /tmp/test_readiness; sleep
600"]
        readinessProbe:
          exec:
            command: ["cat /tmp/test_readiness"]
          initialDelaySeconds: 5
          periodSeconds: 5
```

Regardons maintenant ce qui se passe :

```
$ kubectl apply -f deploy_alpine_readiness.yml
deployment.apps/myalpine created

$ kubectl get pods -n test-readiness
NAME                        READY   STATUS    RESTARTS   AGE
myalpine-6c66f69dc8-tpcj2   1/1     Running   0          20s

$ kubectl describe pods -n test-readiness myalpine-6c66f69dc8-tpcj2
Name:             myalpine-6c66f69dc8-tpcj2
Namespace:        test-readiness
Priority:         0
Service Account:  default
Node:             worker2/192.168.56.33
Start Time:       Mon, 20 Feb 2023 14:32:45 +0100
Labels:           app=myalpine
                  pod-template-hash=6c66f69dc8
Annotations:      cni.projectcalico.org/containerID:
b21b6b3880e09b83bf8bd63ea5fd8f138e9be65567dddaf74c09e7931158c539
                  cni.projectcalico.org/podIP: 10.244.189.88/32
                  cni.projectcalico.org/podIPs: 10.244.189.88/32
Status:           Running
```

```
IP:               10.244.189.88
IPs:
  IP:           10.244.189.88
Controlled By:  ReplicaSet/myalpine-6c66f69dc8
Containers:
  alpine:
    Container ID:
containerd://e49e2f06b6b888e0388302b7805fc4d75772e75bca0c00b11261775e0e165d90
    Image:         alpine
    Image ID:
docker.io/library/alpine@sha256:69665d02cb32192e52e07644d76bc6f25abeb5410edc1c7a81a10ba3f
0efb90a
    Port:          <none>
    Host Port:     <none>
    Command:
      /bin/sh
    Args:
      -c
      touch /tmp/test_readiness;sleep 30;rm -rf /tmp/test_readiness; sleep 600
    State:          Running
      Started:      Mon, 20 Feb 2023 14:32:47 +0100
    Ready:          False
    Restart Count:  0
    Readiness:      exec [cat /tmp/test_readiness] delay=5s timeout=1s period=5s
#success=1 #failure=3
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-jvsz9 (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             False
  ContainersReady   False
  PodScheduled      True
Volumes:
  kube-api-access-jvsz9:
    Type:                    Projected (a volume that contains injected data from
multiple sources)
    TokenExpirationSeconds:  3607
    ConfigMapName:           kube-root-ca.crt
    ConfigMapOptional:       <nil>
    DownwardAPI:             true
QoS Class:                   BestEffort
Node-Selectors:              <none>
Tolerations:                 node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                             node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type     Reason     Age                From               Message
  ----     ------     ----               ----               -------
  Normal   Scheduled  87s                default-scheduler  Successfully assigned test-
readiness/myalpine-6c66f69dc8-tpcj2 to worker2
  Normal   Pulling    82s                kubelet            Pulling image "alpine"
  Normal   Pulled     81s                kubelet            Successfully pulled image
"alpine" in 911.132281ms (911.523112ms including waiting)
  Normal   Created    81s                kubelet            Created container alpine
  Normal   Started    81s                kubelet            Started container alpine
  Warning  Unhealthy  0s (x12 over 48s)  kubelet            Readiness probe failed: cat:
can't open '/tmp/test_readiness': No such file or directory
```

$ kubectl get pods -n test-readiness

```
NAME                        READY   STATUS    RESTARTS   AGE
myalpine-6c66f69dc8-tpcj2   0/1     Running   0          48s
```

Nous lançons notre déploiement. Notre pod apparaît comme READY 1/1 (1 conteneur prêt sur 1). Une fois qu'on arrive à 3 checks readiness échoués, le conteneur apparaît comme non prêt.

Cette situation va perdurer jusqu'à ce que la condition soit de nouveau vérifiée. Nous pouvons le vérifier simplement en recréant le fichier qui nous sert de test :

```
$ kubectl exec -ti -n test-readiness myalpine-6c66f69dc8-tpcj2 -- sh
      # touch /tmp/test_readiness
```

Vérifions maintenant l'état de notre pod :

```
$ kubectl get pods -n test-readiness
NAME                         READY   STATUS    RESTARTS   AGE
myalpine-6c66f69dc8-tpcj2    1/1     Running   0          6m11s
```

Les Healthchecks sont vraiment très pratique pour tester la bonne santé de nos pods.

Nous avons vu comment effectuer des tests simples en utilisant des commandes, mais il est également possible de faire d'autres types de check :

- httpGet : Effecter un test via HTTP.
- tcpSocket : Effectuer un test en vérifiant la présence d'un port en écoute.

Il est bien entendu également possible de combiner ces 2 tests.

Aller plus loin

Les Déploiements

- Recreate
- Rolling Update
- Rollout

Les déploiements

Nous avons déjà vu ce qu'était un déploiement. Il s'agit d'une manière de déployer nos pods en partant de fichiers de configuration, puis de demander à Kubernetes d'atteindre l'état décrit dans les fichiers. Les déploiements regroupent à la fois les pods et les replicaSets.

Cette section va s'attarder sur des fonctionnalités avancées et intéressantes que nous pouvons utiliser, en matière de mise à jour de nos pods.

Nous allons voir les stratégies de recreate, rolling update et rollout.

Recreate :

Cette stratégie de mise à jour est la plus brutale. Elle consiste à supprimer tous les conteneurs avant de les recréer dans la nouvelle version. Voici comment la mettre en place :

```
$ kubectl create ns recreate
namespace/recreate created

$ cat deploy_alpine_recreate.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myalpine
  namespace: recreate
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myalpine
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: myalpine
    spec:
      containers:
      - name: alpine
        image: alpine:3.9
        imagePullPolicy: Always
        command: ["sleep","600"]

$ kubectl apply -f deploy_alpine_recreate.yml
deployment.apps/myalpine created

$ kubectl get pods -n recreate
NAME                        READY   STATUS    RESTARTS   AGE
myalpine-7c44c4d94b-4nxx7   1/1     Running   0          3s
myalpine-7c44c4d94b-hpk6l   1/1     Running   0          3s
myalpine-7c44c4d94b-hzbxc   1/1     Running   0          3s

$ kubectl exec -ti -n recreate myalpine-7c44c4d94b-4nxx7 -- sh
      # cat /etc/alpine-release
      3.9.6
```

Nous créons le namespace recreate, puis nous déployons 3 instances de pods alpine, dont nous fixons la version à la 3.9. Nous constatons que les 3 pods fonctionnent, et que nous sommes bien en alpine 3.9.6.

Modifions maintenant notre fichier de configuration :

```
 $ cat deploy_alpine_recreate.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myalpine
  namespace: recreate
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myalpine
  strategy:
    type: Recreate
```

```
  template:
    metadata:
      labels:
        app: myalpine
    spec:
      containers:
      - name: alpine
        image: alpine:3.10
        command: ["sleep","600"]

$ kubectl apply -f deploy_alpine_recreate.yml
deployment.apps/myalpine configured

$ kubectl get pods -n recreate
NAME                         READY   STATUS        RESTARTS   AGE
myalpine-7c44c4d94b-4nxx7    1/1     Terminating   0          81s
myalpine-7c44c4d94b-hpk6l    1/1     Terminating   0          81s
myalpine-7c44c4d94b-hzbxc    1/1     Terminating   0          81s

$ kubectl get pods -n recreate
NAME                         READY   STATUS              RESTARTS   AGE
myalpine-84fd5b866-6x52q     0/1     ContainerCreating   0          1s
myalpine-84fd5b866-n84cz     0/1     ContainerCreating   0          1s
myalpine-84fd5b866-sfv5k     0/1     ContainerCreating   0          1s

$ kubectl get pods -n recreate
NAME                         READY   STATUS    RESTARTS   AGE
myalpine-84fd5b866-6x52q     1/1     Running   0          3s
myalpine-84fd5b866-n84cz     1/1     Running   0          3s
myalpine-84fd5b866-sfv5k     1/1     Running   0          3s

$ kubectl exec -n recreate -ti myalpine-84fd5b866-6x52q -- sh
      # cat /etc/alpine-release
      3.10.5
```

Une fois la nouvelle configuration appliquée, Kubernetes va se charger de supprimer les pods existants, puis il va les recréer dans la nouvelle version. Nous constatons que nous sommes bien dans la version demandée.

Cette stratégie engendre cependant une interruption de service, étant donné que tous les pods ont d'abord été supprimés.

Rolling Update :

Voyons maintenant le Rolling Update. Cette stratégie va nous permettre d'effectuer nos mises à jour au fur et à mesure, ce qui va nous permettre d'éviter une interruption de service au cour de la mise à jour, en effectuant une montée de version progressive.

Voyons tout de suite le fichier de configuration :

```
$ kubectl create ns rolling-update
namespace/rolling-update created

$ cat deploy_alpine_rolling_update.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myalpine
  namespace: rolling-update
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myalpine
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 2
      maxUnavailable: 0
  template:
    metadata:
      labels:
        app: myalpine
    spec:
      containers:
      - name: alpine
        image: alpine:3.9
        command: ["sleep","600"]

$ kubectl apply -f deploy_alpine_rolling_update.yml
deployment.apps/myalpine created

$ kubectl get pods -n rolling-update
NAME                        READY   STATUS    RESTARTS   AGE
myalpine-7c44c4d94b-hx6n9   1/1     Running   0          29s
myalpine-7c44c4d94b-t5lrx   1/1     Running   0          29s
myalpine-7c44c4d94b-wnpmw   1/1     Running   0          29s
```

Nous avons cette fois-ci demandé à appliquer la stratégie RollingUpdate. Nous avons également défini 2 paramètres :

- maxSurge : Permet de définir le nombre maximum de pods supplémentaires autorisés.
- maxUnavailable : Permet de définir le nombre maximum de pods arrêtés autorisés.

Tel que nous l'avons défini, Kubernetes va d'abord créer 2 pods en 3.10, puis arrêter 2 pods en 3.9. Il va ensuite faire la même chose pour le dernier pod.

D'autres paramètres peuvent être intéressants :

- minReadySeconds : délai pour lancer un autre update de pod.
- ProgressDeadlineSeconds : délai maximum pour le déploiement du nouveau pod, sinon l'ancien pod reste en place.

Vérifions tout de suite le comportement :

```
$ cat deploy_alpine_rolling_update.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myalpine
  namespace: rolling-update
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myalpine
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 2
      maxUnavailable: 0
  template:
    metadata:
      labels:
        app: myalpine
    spec:
      containers:
      - name: alpine
        image: alpine:3.10
        command: ["sleep","600"]

$ apply -f deploy_alpine_rolling_update.yml
deployment.apps/myalpine configured

$ kubectl get pods -n rolling-update
NAME                        READY   STATUS              RESTARTS   AGE
myalpine-7c44c4d94b-hx6n9   1/1     Running             0          6m10s
myalpine-7c44c4d94b-t5lrx   1/1     Running             0          6m10s
myalpine-7c44c4d94b-wnpmw   1/1     Running             0          6m10s
myalpine-84fd5b866-9b677    0/1     ContainerCreating   0          1s
myalpine-84fd5b866-c5f6q    0/1     ContainerCreating   0          1s

$ kubectl get pods -n rolling-update
NAME                        READY   STATUS              RESTARTS   AGE
myalpine-7c44c4d94b-hx6n9   1/1     Terminating         0          6m12s
myalpine-7c44c4d94b-t5lrx   1/1     Running             0          6m12s
myalpine-7c44c4d94b-wnpmw   1/1     Terminating         0          6m12s
myalpine-84fd5b866-8bfvn    0/1     ContainerCreating   0          2s
myalpine-84fd5b866-9b677    1/1     Running             0          3s
myalpine-84fd5b866-c5f6q    1/1     Running             0          3s

$ kubectl get pods -n rolling-update
NAME                        READY   STATUS        RESTARTS   AGE
myalpine-7c44c4d94b-hx6n9   1/1     Terminating   0          6m17s
myalpine-7c44c4d94b-t5lrx   1/1     Terminating   0          6m17s
myalpine-7c44c4d94b-wnpmw   1/1     Terminating   0          6m17s
myalpine-84fd5b866-8bfvn    1/1     Running       0          7s
myalpine-84fd5b866-9b677    1/1     Running       0          8s
myalpine-84fd5b866-c5f6q    1/1     Running       0          8s
```

```
$ kubectl get pods -n rolling-update
NAME                         READY   STATUS    RESTARTS   AGE
myalpine-84fd5b866-8bfvn     1/1     Running   0          2m11s
myalpine-84fd5b866-9b677     1/1     Running   0          2m12s
myalpine-84fd5b866-c5f6q     1/1     Running   0          2m12s
```

On constate que nous obtenons bien le comportement attendu. Une fois qu'un pod est à l'état Terminating, Kubernetes considère qu'il est arrêté et procède à l'étape suivante.

C'est ce qui explique que nous avons à un moment donné 6 conteneurs, 3 à l'état Terminating et 3 à l'état Running, alors que nous avions demandé un maximum de 5 pods simultanés (3 réplicas + 2 maxSurge).

Rollout :

Kubernetes met à notre disposition un autre outil puissant nous permettant d'interagir avec nos mises à jour de déploiement, grâce à la sous-commande rollout.

Nous allons par exemple pouvoir suivre les mises à jours :

```
$  kubectl apply -f deploy_alpine_rolling_update.yml
deployment.apps/myalpine configured
$ kubectl -n rolling-update rollout status deployment myalpine
Waiting for deployment "myalpine" rollout to finish: 2 out of 3 new replicas have been
updated...
Waiting for deployment "myalpine" rollout to finish: 2 out of 3 new replicas have been
updated...
Waiting for deployment "myalpine" rollout to finish: 2 out of 3 new replicas have been
updated...
Waiting for deployment "myalpine" rollout to finish: 2 out of 3 new replicas have been
updated...
Waiting for deployment "myalpine" rollout to finish: 2 out of 3 new replicas have been
updated...
Waiting for deployment "myalpine" rollout to finish: 2 old replicas are pending
termination...
Waiting for deployment "myalpine" rollout to finish: 1 old replicas are pending
termination...
Waiting for deployment "myalpine" rollout to finish: 1 old replicas are pending
termination…
deployment "myalpine" successfully rolled out
```

Nous pouvons suivre ici en direct la mise à jour de nos pods et les suppressions des anciens.

Nous pouvons également accéder à un historique de nos déploiements :

```
$ kubectl -n rolling-update rollout history deployment myalpine
deployment.apps/myalpine
REVISION  CHANGE-CAUSE
2         <none>
3         <none>
```

Là ou cela devient intéressant, c'est qu'il est possible d'annoter nos mises à jour, et ainsi d'avoir une historique un peu plus parlant :

```
$ cat deploy_alpine_rolling_update.yml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myalpine
  namespace: rolling-update
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myalpine
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 2
      maxUnavailable: 0
  template:
    metadata:
      labels:
        app: myalpine
      annotations:
        kubernetes.io/change-cause: "Mise a jour en 3.10"
```

```
    spec:
      containers:
      - name: alpine
        image: alpine:3.10
        command: ["sleep","600"]

$ kubectl apply -f deploy_alpine_rolling_update.yml
deployment.apps/myalpine configured

$ kubectl -n rolling-update rollout history deployment myalpine
deployment.apps/myalpine
REVISION  CHANGE-CAUSE
2         <none>
3         <none>
4         Mise a jour en 3.10
```

Notez ici l'utilisation de la directive annotation dans la partie métadata, ce qui nous as permis de commenter notre mise à jour.

Il est également possible d'obtenir des informations détaillées sur une révision :

```
$ kubectl -n rolling-update rollout history \
              deployment myalpine --revision 4
deployment.apps/myalpine with revision #4
Pod Template:
  Labels:       app=myalpine
        pod-template-hash=555d5668b8
  Annotations:  kubernetes.io/change-cause: Mise a jour en 3.10
  Containers:
   alpine:
    Image:      alpine:3.10
    Port:       <none>
    Host Port:  <none>
    Command:
      sleep
      600
    Environment:        <none>
    Mounts:     <none>
  Volumes:      <none>
```

Mais la fonctionnalité la plus intéressante de cette sous-commande est sans doute le rollback. Il est en effet très aisé de revenir en arrière :

```
$ kubectl rollout -n rolling-update undo \
                      --to-revision 3 deployment myalpine
deployment.apps/myalpine rolled back
```

Cela peut s'avérer très pratique en cas de problème de mise à jour pour effectuer un retour arrière rapide.

Aller plus loin

Les Jobs

- L'objet job
- Job ponctuel
- Parallélisme

Les Jobs

Nous allons voir une nouvelle fonctionnalité dans Kubernetes, les Jobs.

Nous avions jusqu'à présent manipulé des déploiements, ou le principe est d'atteindre un état et le conserver. Cela signifie que si je souhaite un pod, celui-ci sera recréé indéfiniment même si l'utilité de ce pod est d'effectuer une action et ensuite s'arrêter.

C'est dans ce cas de figure que les jobs entrent en jeu.

L'objet Job :

Dans Kubernetes, l'objet Job s'occupe de l'exécution du ou des pods dont il a la charge jusqu'à ce que ce/ces pods se soient exécutés avec succès.

Pour cela, le job va se baser sur un modèle de spécification de jobs. Il en existe 3 différents :

- Job Ponctuel : Un seul pod s'exécutant une seule fois jusqu'à ce qu'il se termine avec succès.
- Achèvements fixes parallèles : un ou plusieurs pods s'exécutant une ou plusieurs fois jusqu'à atteindre un nombre d'achèvements fixe.
- File d'attente de travaux (jobs parallèles) : Un ou plusieurs pods s'exécutant plusieurs fois jusqu'à ce qu'ils se terminent avec succès.

Nous allons maintenant voir comment mettre en œuvre les Jobs ponctuels et les Jobs à achèvements fixes parallèles.

Job ponctuel :

Comme d'habitude, notre job peut être lancé en ligne de commande, mais il est beaucoup plus pratique d'utiliser un fichier de configuration :

```
$ kubectl create ns oneshot
namespace/oneshot created


$ cat job_ponctuel.yml
---
apiVersion: batch/v1
kind: Job
metadata:
  name: oneshot
  namespace: oneshot
  labels:
    app: oneshot
spec:
  template:
    metadata:
      labels:
        app: oneshot
    spec:
      containers:
      - name: myalpine
        image: alpine
        command: ["echo","Execution du job"]
      restartPolicy: OnFailure


$ kubectl apply -f job_ponctuel.yml
job.batch/oneshot created


$ kubectl get jobs -n oneshot
NAME      COMPLETIONS   DURATION   AGE
oneshot   1/1           15s        15s
```

```
$ kubectl describe jobs -n oneshot
Name:             oneshot
Namespace:        oneshot
Selector:         controller-uid=45c1a44d-4a08-4ca6-a5b5-27f4bf03b325
Labels:           app=oneshot
Annotations:      batch.kubernetes.io/job-tracking:
Parallelism:      1
Completions:      1
Completion Mode:  NonIndexed
Start Time:       Mon, 20 Feb 2023 14:40:12 +0100
Completed At:     Mon, 20 Feb 2023 14:40:27 +0100
Duration:         15s
Pods Statuses:    0 Active (0 Ready) / 1 Succeeded / 0 Failed
Pod Template:
  Labels:  app=oneshot
           controller-uid=45c1a44d-4a08-4ca6-a5b5-27f4bf03b325
           job-name=oneshot
  Containers:
   myalpine:
    Image:      alpine
    Port:       <none>
    Host Port:  <none>
    Command:
      /bin/sh
    Args:
      -c
      echo Debut; sleep 10; echo Fin
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age   From            Message
  ----    ------            ----  ----            -------
  Normal  SuccessfulCreate  27s   job-controller  Created pod: oneshot-d94hm
  Normal  Completed         12s   job-controller  Job completed


$ kubectl get pods -n oneshot
NAME            READY   STATUS      RESTARTS   AGE
oneshot-d94hm   0/1     Completed   0          41s

$ kubectl logs -n oneshot oneshot-d94hm
Execution du job
```

Nous créons donc un job oneshot dans le namespace de même nom. Notez l'utilisation de l'API batch/v1.

Nous pouvons accéder aux informations sur le job grâce à la commande kubectl get jobs ou obtenir les informations détaillées de celui-ci avec un kubectl describe jobs.

Nous pouvons remarquer la bonne exécution du job, et que celui-ci apparaît comme complété.

Nous pouvons également regarder du côté du pod créé par le job : Nous voyons qu'ils est en status Completed, et que l'affichage des logs nous montre que la commande echo s'est correctement exécutée.

Notez également l'utilisation dans notre fichier de configuration de la directive restartPolicy: OnFailure. Par cela nous indiquons à Kubernetes que nous voulons relancer le job jusqu'à ce qu'il réussisse.

Une autre possibilité aurait été de définir cette directive à Never, afin de ne pas retenter indéfiniment l'exécution de notre job.

Parallélisme :

Voyons maintenant comment paralléliser nos jobs. Prenons tout d'abord l'exemple suivant :

```
$ cat job_parallele.yml
---
apiVersion: batch/v1
kind: Job
metadata:
  name: para
  namespace: para
  labels:
    app: para
spec:
  parallelism: 5
  completions: 10
  template:
    metadata:
      labels:
        app: para
    spec:
      containers:
      - name: myalpine
        image: alpine
        command: ["/bin/sh"]
        args: ["-c","sleep 30;echo Execution du Job."]
      restartPolicy: OnFailure
```

Notre exemple est très similaire au précédent, à ceci près que nous avons ajouté 2 directives :

- parallelism : Le nombre d'exécution en parallèle que nous souhaitons
- completions : Le nombre d'exécution total que nous souhaitons.

Nous avons également étoffé un peu notre commande afin d'avoir un temps d'observation suffisant.

Voyons maintenant ce qui se passe :

```
$ kubectl apply -f job_parallele.yml
job.batch/para created

$  kubectl get jobs -n para
NAME   COMPLETIONS   DURATION   AGE
para   0/10          7s         7s

$ kubectl get pods -n para
NAME         READY   STATUS    RESTARTS   AGE
para-25n49   1/1     Running   0          17s
para-j6b4p   1/1     Running   0          17s
para-nkv5t   1/1     Running   0          17s
para-qlgvz   1/1     Running   0          17s
para-qm7rx   1/1     Running   0          17s

$ kubectl get jobs -n para
NAME   COMPLETIONS   DURATION   AGE
para   5/10          37s        37s
```

```
$ kubectl get pods -n para
NAME         READY   STATUS              RESTARTS   AGE
para-25n49   0/1     Completed           0          39s
para-5tzk5   1/1     Running             0          5s
para-dkf7n   1/1     Running             0          5s
para-hb579   1/1     Running             0          4s
para-j6b4p   0/1     Completed           0          39s
para-kwsdc   1/1     Running             0          4s
para-nkv5t   0/1     Completed           0          39s
para-qlgvz   0/1     Completed           0          39s
para-qm7rx   0/1     Completed           0          39s
para-vdlwh   0/1     ContainerCreating   0          2s

$ kubectl get jobs -n para
NAME   COMPLETIONS   DURATION   AGE
para   10/10         70s        70s

$ kubectl get pods -n para
NAME         READY   STATUS      RESTARTS   AGE
para-25n49   0/1     Completed   0          71s
para-5tzk5   0/1     Completed   0          37s
para-dkf7n   0/1     Completed   0          37s
para-hb579   0/1     Completed   0          36s
para-j6b4p   0/1     Completed   0          71s
para-kwsdc   0/1     Completed   0          36s
para-nkv5t   0/1     Completed   0          71s
para-qlgvz   0/1     Completed   0          71s
para-qm7rx   0/1     Completed   0          71s
para-vdlwh   0/1     Completed   0          34s
```

Nous avons bien le comportement attendu, à savoir :

- Exécution de 5 pods en parallèle.
- Une fois ceux-ci terminés, exécution des 5 suivants.
- Une fois les 10 terminés, fin du Job.

Aller plus loin

Stateful / Stateless

- Principe
- Exemple

Stateful / Stateless

Principe :

Les déploiements que nous avons utilisé jusqu'à présent étaient de type stateless. Cela signifie qu'il n'y a pas de notion d'ordre dans la création des pods, ni dans leur nommage.

Le problème avec ce type de déploiement, c'est que lorsque l'on commence à attacher des volumes à ces pods, cela se fera de manière anarchique.

Pour pallier à cela, il est possible de créer des déploiements Stateful, à l'aide de la ressource StatefulSet. Nous allons voir ensemble comment les mettre en œuvre.

Exemple :

Pour mettre en pratique les StatefulSet, nous allons prendre le cas du SGBD Cassandra. Il s'agit d'une base de données NoSQL distribuée.

Nous allons commencer par créer une StorageClass, pour notre base de données. Nous avons déjà vu les StorageClass lors du chapitre sur les PersistentVolumes, avec l'utilisation du StorageClass « Manual ». Pour changer, nous allons cette fois-ci créer notre propre StorageClass.

```
$ kubectl create ns stateful
namespace/stateful created

$ cat storageClass.yml
---
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: cassandra-sc
  namespace: stateful
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer

$ kubectl apply -f storageClass.yml
storageclass.storage.k8s.io/cassandra-sc created

$ kubectl get storageclass -n stateful
NAME           PROVISIONER                    RECLAIMPOLICY   VOLUMEBINDINGMODE
ALLOWVOLUMEEXPANSION   AGE
cassandra-sc   kubernetes.io/no-provisioner   Delete          WaitForFirstConsumer
false                  10s
```

Le « WaitForFirstConsumer » indique que l'on attende au moins un pod avant d'activer la StorageClass.

Nous allons maintenant créer 2 volumes persistents :

```
$ cat pv.yml
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: cassandra-pv1
  namespace: stateful
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: cassandra-sc
  hostPath:
    path: /srv/cassandra
    type: DirectoryOrCreate
  nodeAffinity:
    required:
      nodeSelectorTerms:
        - matchExpressions:
            - key: kubernetes.io/hostname
              operator: In
              values:
                - worker1
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: cassandra-pv2
  namespace: stateful
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: cassandra-sc
  hostPath:
    path: /srv/cassandra
```

```
    type: DirectoryOrCreate
  nodeAffinity:
    required:
      nodeSelectorTerms:
        - matchExpressions:
            - key: kubernetes.io/hostname
              operator: In
              values:
                - worker2

$ kubectl apply -f pv.yml
persistentvolume/cassandra-pv1 created
persistentvolume/cassandra-pv2 created

$ kubectl get pv -n stateful
NAME            CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS       CLAIM
STORAGECLASS   REASON   AGE
cassandra-pv1   1Gi        RWO            Retain           Available
cassandra-sc            13s
cassandra-pv2   1Gi        RWO            Retain           Available
cassandra-sc            13s
```

Nous créons ici 2 volumes persistents, en utilisant notre StorageClass. Nous avons également définit une directive nodeAffinity, afin de forcer le premier volume à se placer sur worker1, et le deuxième sur worker2.

Nous allons maintenant créer un nouveau type de service, le service Headless. Il s'agit en fait d'un service cluster Ip, mais pour lequel nous ne souhaitons pas d'adresse IP disponible pour le cluster. Cela va servir à fournir un DNS interne pour les pods de notre statefulset, afin qu'il puissent communiquer entre eux :

```
$ cat headless.yml
---
apiVersion: v1
kind: Service
metadata:
  name: svc-cassandra
  namespace: stateful
  labels:
    app: cassandra
spec:
  clusterIP: None
  ports:
    - port: 9042
  selector:
    app: cassandra

$ kubectl apply -f headless.yml
service/svc-cassandra created

$ kubectl get svc -n stateful
NAME            TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
svc-cassandra   ClusterIP   None         <none>        9042/TCP   6s
```

Il est maintenant tant de créer notre StatefulSet :

```
$ cat sts.yml
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: cassandra
  namespace: stateful
  labels:
    app: cassandra
spec:
  serviceName: svc-cassandra
  replicas: 2
  selector:
    matchLabels:
      app: cassandra
  template:
    metadata:
      labels:
        app: cassandra
    spec:
      dnsPolicy: ClusterFirstWithHostNet
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - cassandra
            topologyKey: kubernetes.io/hostname
      terminationGracePeriodSeconds: 1800
      containers:
      - name: cassandra
        image: gcr.io/google-samples/cassandra:v13
        imagePullPolicy: Always
        ports:
        - containerPort: 7000
          name: intracom
        - containerPort: 7001
          name: tls-intracom
        - containerPort: 7199
          name: jmx
        - containerPort: 9042
          name: cql
        resources:
          limits:
            cpu: "500m"
            memory: 1Gi
          requests:
            cpu: "500m"
            memory: 1Gi
        securityContext:
          capabilities:
            add:
              - IPC_LOCK
        lifecycle:
          preStop:
            exec:
              command:
              - /bin/sh
              - -c
              - nodetool drain
        env:
          - name: MAX_HEAP_SIZE
            value: 512M
```

```
          - name: HEAP_NEWSIZE
            value: 100M
          - name: CASSANDRA_SEEDS             # node init cluster
            value: "cassandra-0.svc-cassandra.stateful.svc.cluster.local"
          - name: CASSANDRA_CLUSTER_NAME      # cluster name
            value: "K8Demo"
          - name: CASSANDRA_DC                # split by DC
            value: "DC1-K8Demo"
          - name: CASSANDRA_RACK              # split by rack
            value: "Rack1-K8Demo"
          - name: POD_IP
            valueFrom:
              fieldRef:
                fieldPath: status.podIP
        readinessProbe:
          exec:
            command:
            - /bin/bash
            - -c
            - /ready-probe.sh                 # state of node
          initialDelaySeconds: 30
          timeoutSeconds: 30
        volumeMounts:
        - name: cassandra-data
          mountPath: /cassandra_data
  volumeClaimTemplates:
  - metadata:
      name: cassandra-data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: cassandra-sc
      resources:
        requests:
          storage: 1Gi
```

```
$ kubectl apply -f sts.yml
statefulset.apps/cassandra created

$ kubectl get sts -n stateful
NAME        READY   AGE
cassandra   2/2     2m11s

$ kubectl get pods -n stateful
NAME          READY   STATUS    RESTARTS   AGE
cassandra-0   1/1     Running   0          2m26s
cassandra-1   1/1     Running   0          2m4s
```

Nous allons demander 2 replicas de notre StatefulSet, réparti sur les deux nœuds grâce à la directive « affinity ». Ces replicas vont consommer les pvs que nous avons créé.

Nous pouvons déjà constater que les pods sont créés dans l'ordre. Tant que cassandra-0 n'est pas créé, Kubernetes ne va pas créer cassandra-1. De plus, les noms n'ont plus d'id généré, mais un -0, -1… Si nous supprimons cassandra-1, nous constatons qu'il est recréé avec le même nom.

Enfin, nous pouvons nous connecter sur un pod, et vérifier que les enregistrements DNS sont bien fait automatiquement, grâce à notre service « Headless » :

```
$ kubectl exec -n stateful -ti cassandra-0 -- bash
# apt update && apt install -y dnsutils
# nslookup cassandra-0.svc-cassandra
Server:            10.96.0.10
Address:     10.96.0.10#53

Name:  cassandra-0.svc-cassandra.stateful.svc.cluster.local
Address: 10.244.189.85
```

```
# nslookup cassandra-1.svc-cassandra
Server:             10.96.0.10
Address:     10.96.0.10#53

Name:  cassandra-1.svc-cassandra.stateful.svc.cluster.local
Address: 10.244.235.147
```

Si nous supprimons un pod, celui-ci sera recréé, et nous aurons de nouveau son enregistrement DNS valide.

Synthèse :

```
$ kubectl -n stateful get sc,pv,pvc,svc,sts,pods
NAME                                          PROVISIONER                    RECLAIMPOLICY
VOLUMEBINDINGMODE      ALLOWVOLUMEEXPANSION   AGE
storageclass.storage.k8s.io/cassandra-sc   kubernetes.io/no-provisioner   Delete
WaitForFirstConsumer   false                  44m


NAME                              CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
CLAIM                                   STORAGECLASS   REASON   AGE
persistentvolume/cassandra-pv1   1Gi        RWO            Retain           Bound
stateful/cassandra-data-cassandra-1   cassandra-sc           29m
persistentvolume/cassandra-pv2   1Gi        RWO            Retain           Bound
stateful/cassandra-data-cassandra-0   cassandra-sc           29m


NAME                                                STATUS   VOLUME          CAPACITY
ACCESS MODES   STORAGECLASS   AGE
persistentvolumeclaim/cassandra-data-cassandra-0   Bound    cassandra-pv2   1Gi
RWO            cassandra-sc   29m
persistentvolumeclaim/cassandra-data-cassandra-1   Bound    cassandra-pv1   1Gi
RWO            cassandra-sc   5m31s


NAME                    TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
service/svc-cassandra   ClusterIP   None         <none>        9042/TCP   29m


NAME                           READY   AGE
statefulset.apps/cassandra   2/2     4m25s


NAME              READY   STATUS    RESTARTS   AGE
pod/cassandra-0   1/1     Running   0          4m25s
pod/cassandra-1   1/1     Running   0          3m36s
```

Annexes

Annexes

Helm

- Présentation
- Installation de Helm
- Déploiement d'un application avec Helm

Helm

Présentation :

Helm est un outil qui a été introduit par la communauté afin de répondre à un besoin de simplification de l'installation d'applications dans un cluster Kubernetes.

En effet, depuis le début de cette formation, nous avons vu un grand nombre de fonctionnalités, et que pour déployer une stack complète d'applications, il faudrait faire appel à un grand nombre de fichiers de configurations, les maintenir et penser à tous les outils à mettre en œuvre pour un bon fonctionnement de l'application.

Helm fonctionne à l'aide de Charts (packages), qui sont une aide précieuse pour s'affranchir des problématiques d'installation initiale.

Nous allons voir comment l'installer sur notre cluster, et ensuite nous verrons comment déployer une application en s'appuyant dessus.

Installation de Helm :

Nous allons maintenant passer à l'installation de Helm. L'installation doit s'effectuer sur une station ou la commande kubectl fonctionne. Helm a en effet besoin du même contexte que kubectl pour fonctionner.

L'installation s'effectue en 3 étapes :

- Récupération du binaire :

```
$ wget https://get.helm.sh/helm-v3.2.1-linux-amd64.tar.gz
--2023-02-20 15:39:15--  https://get.helm.sh/helm-v3.2.1-linux-amd64.tar.gz
Resolving get.helm.sh (get.helm.sh)... 152.199.21.175,
2606:2800:233:1cb7:261b:1f9c:2074:3c
Connecting to get.helm.sh (get.helm.sh)|152.199.21.175|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 12927632 (12M) [application/x-tar]
Saving to: 'helm-v3.2.1-linux-amd64.tar.gz'

helm-v3.2.1-linux-amd64.tar.gz  100%
[=====================================================>]  12,33M  6,66MB/s    in 1,9s

2023-02-20 15:39:18 (6,66 MB/s) - 'helm-v3.2.1-linux-amd64.tar.gz' saved
[12927632/12927632]
```

- Décompression du fichier tar.gz :

```
$ tar xfvz helm-v3.2.1-linux-amd64.tar.gz
linux-amd64/
linux-amd64/README.md
linux-amd64/helm
linux-amd64/LICENSE
```

- Copie du binaire dans le répertoire /usr/local/bin :

```
$ sudo cp linux-amd64/helm /usr/local/bin/
```

Voilà, Helm est installé, nous pouvons vérifier son bon fonctionnement en tapant la commande suivante :

```
$ helm
The Kubernetes package manager

Common actions for Helm:

- helm search:    search for charts
- helm pull:      download a chart to your local directory to view
- helm install:   upload the chart to Kubernetes
- helm list:      list releases of charts
[…]
```

Cette commande nous affiche l'aide de Helm.

Déploiement d'une application avec Helm :

Nous allons prendre pour exemple l'application WordPress. Il s'agit d'une application permettant de réaliser des sites web.

Cette application est composée de deux briques :

- Une partie frontale de présentation (en PHP).
- Une partie de stockage des données (basée sur MariaDB).

Avant de l'installer, il nous faut chercher si un package remplit ses conditions. Nous pouvons effectuer une recherche sur le hub de Helm :

```
$ helm search hub wordpress
URL                                               CHART VERSION   APP VERSION
DESCRIPTION
https://hub.helm.sh/charts/presslabs/wordpress-...      v0.8.4          v0.8.4
Presslabs WordPress Operator Helm Chart
https://hub.helm.sh/charts/presslabs/wordpress-...      v0.8.5          v0.8.5          A
Helm chart for deploying a WordPress site on ...
https://hub.helm.sh/charts/bitnami/wordpress            9.2.5           5.4.1
Web publishing platform for building blogs and …
```

Nous voyons qu'il y a plusieurs charts de disponibles. Nous allons utiliser la chart développée par bitnami, mais nous allons d'abord ajouter son dépôt :

```
$ helm repo add bitnami https://charts.bitnami.com/bitnami
"bitnami" has been added to your repositories

$ helm repo list
NAME    URL
bitnami https://charts.bitnami.com/bitnami
```

Une fois le dépôt prêt, nous allons devoir effectuer quelques préparatifs.

Toutes les étapes énoncées ci-après peuvent être retrouvées en consultant le README du charts présent ici :https://github.com/bitnami/charts/tree/master/bitnami/wordpress/

Nous allons tout d'abord créer un namespace pour l'occasion :

```
$ kubectl create ns wordpress
namespace/wordpress created
```

Pour conserver la persistance des données, nous allons devoir créer un PersistentVolume :

```
$ cat pv-wordpress.yml
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-wordpress
  namespace: wordpress
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
    - ReadOnlyMany
  hostPath:
```

```
    path: /srv/wordpress/app

$ kubectl apply -f pv-wordpress.yml
persistentvolume/pv-wordpress created
```

Nous allons également créer un deuxième PV, qui nous servira pour la base de données :

```
$ cat pv-bdd-wordpress.yml
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-bdd-wordpress
  namespace: wordpress
spec:
  capacity:
    storage: 8Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /srv/wordpress/bdd
$ kubectl apply -f pv-bdd-wordpress.yml
persistentvolume/pv-bdd-wordpress created
```

Pensez à créer les répertoires correspondant sur les nœuds. De plus, le fichier README nous apprend que l'utilisateur des conteneurs est le 1001:1001 (uid:gid). Il faut donc donner les bons droits à nos répertoires.

Enfin, l'application a besoin également d'un répertoire wordpress dans l'arborescence créée. Il faut donc créer le répertoire /srv/wordpress/app/wordpress avec les bons droits.

Nous pouvons maintenant créer le PersistentVolumeClaim associé au PV de l'application :

```
$ cat pvc-wordpress.yml
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-wordpress
  namespace: wordpress
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi

$ kubectl apply -f pvc-wordpress.yml
```

Nous allons d'abord « puller » le chart wordpress afin de modifier certaines variables de configuration et l'adapter à notre environnement :

```
$ helm pull --untar bitnami/wordpress
```

Nous avons maintenant un répertoire wordpress. C'est le fichier values.yaml à l'intérieur de ce répertoire qui nous intéresse. Nous allons l'éditer.

A l'intérieur, il y a un grand nombre de variables que l'on peut modifier afin d'adapter la configuration. Nous allons nous intéresser tout d'abord à la partie service.

Par défaut, le service utilisé pour exposer l'application est le LoadBalancer. Nous n'avons pas de controller Ingress à disposition et nous ne pouvons donc pas utiliser ce service. Nous allons donc remplacer 'type: LoadBalancer' par 'type: NodePort' :

```
service:
  type: NodePort
```

Il faut également renseigner le nom de notre PersistentVolumeClaim dans le fichier de configuration :

```
existingClaim: pvc-wordpress
```

Nous pouvons maintenant lancer l'installation de notre chart :

```
$ helm install -f wordpress/values.yaml --namespace wordpress my-
wordpress bitnami/wordpress
NAME: my-wordpress
NAMESPACE: wordpress
STATUS: deployed
REVISION: 1
NOTES:
** Please be patient while the chart is being deployed **

To access your WordPress site from outside the cluster follow the steps below:

1. Get the WordPress URL by running these commands:

   export NODE_PORT=$(kubectl get --namespace wordpress -o
jsonpath="{.spec.ports[0].nodePort}" services my-wordpress)
   export NODE_IP=$(kubectl get nodes --namespace wordpress -o
jsonpath="{.items[0].status.addresses[0].address}")
   echo "WordPress URL: http://$NODE_IP:$NODE_PORT/"
   echo "WordPress Admin URL: http://$NODE_IP:$NODE_PORT/admin"

2. Open a browser and access WordPress using the obtained URL.

3. Login with the following credentials below to see your blog:

  echo Username: user
  echo Password: $(kubectl get secret --namespace wordpress my-wordpress -o
jsonpath="{.data.wordpress-password}" | base64 --decode)
```

Après un certain moment, les 2 pods apparaissent en READY :

```
$ kubectl get pods  -n wordpress
NAME                            READY   STATUS    RESTARTS   AGE
my-wordpress-566d7c44d7-fl77f   1/1     Running   0          104s
my-wordpress-mariadb-0          1/1     Running   0          104s
```

Lors du lancement de notre commande 'helm install', nous avons eu les éléments pour accéder à l'application. Récupérons d'abord l'url :

```
$ export NODE_PORT=$(kubectl get --namespace wordpress -o
jsonpath="{.spec.ports[0].nodePort}" services my-wordpress)
$ export NODE_IP=$(kubectl get nodes --namespace wordpress -o
jsonpath="{.items[0].status.addresses[0].address}")
$ echo "WordPress URL: http://$NODE_IP:$NODE_PORT/"
WordPress URL: http://192.168.56.31:31428/
```

Nous avons accès à notre application.

Pour accéder à l'interface d'administration, il suffit de se connecter à l'url http://192.168.56.31:31428/admin et de saisir les identifiants générés. Les commandes permettant de les obtenir sont fournies en résultat du 'helm install'.

Nous avons réussi à déployer assez facilement une application avec une base de données, et un service pour exposer le tout.

Annexe

Dashboard

- Installation du Dashboard
- Découverte des fonctionnalités

Dashboard

Kubernetes propose une autre manière d'administrer que d'utiliser la ligne de commande ou les fichiers de configuration.

Il s'agit du Dashboard Kubernetes, une Web UI qui va nous permettre d'administrer notre cluster de manière graphique.

Attention, le Dashboard est un élément pouvant apporter des vulnérabilités à notre cluster. Il convient d'être très prudent lorsque l'on souhaite le déployer dans un environnement de production.

Nous allons voir comment l'installer et s'en servir.

Installation du Dashboard :

Kubernetes met à notre disposition un fichier de configuration yaml pour déployer le dashboard. Une première possibilité serait de le lancer directement, mais, dans ce cas, nous aurions un dashboard déployé avec un service ClusterIp. Pour des raisons de praticité, nous allons récupérer le fichier yaml et le modifier pour déployer notre dashboard avec un service NodePort :

```
$ wget \
https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/\
deploy/recommended.yaml
--2023-02-20 15:46:26--  http://%20/
Resolving   ( )... failed: Name or service not known.
```

```
wget: unable to resolve host address ‘ ’
--2023-02-20 15:46:26--
https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/deploy/recommended.yaml
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.109.133,
185.199.108.133, 185.199.111.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|
185.199.109.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 7621 (7,4K) [text/plain]
Saving to: ‘recommended.yaml’

recommended.yaml                100%
[=====================================================>]   7,44K  --.-KB/s    in 0,003s

2023-02-20 15:46:27 (2,40 MB/s) - ‘recommended.yaml’ saved [7621/7621]

FINISHED --2023-02-20 15:46:27--
Total wall clock time: 0,2s
Downloaded: 1 files, 7,4K in 0,003s (2,40 MB/s)
```

```
$ mv recommended.yaml dash.yml
```

Nous allons maintenant modifier la section service pour lui ajouter le type NodePort :

```
kind: Service
apiVersion: v1
metadata:
  labels:
    k8s-app: kubernetes-dashboard
  name: kubernetes-dashboard
  namespace: kubernetes-dashboard
spec:
  ports:
    - port: 443
      targetPort: 8443
  selector:
    k8s-app: kubernetes-dashboard
  type: NodePort
```

Nous pouvons maintenant déployer notre dashboard :

```
$ kubectl apply -f dash.yml
namespace/kubernetes-dashboard created
serviceaccount/kubernetes-dashboard created
service/kubernetes-dashboard created
secret/kubernetes-dashboard-certs created
secret/kubernetes-dashboard-csrf created
secret/kubernetes-dashboard-key-holder created
configmap/kubernetes-dashboard-settings created
role.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrole.rbac.authorization.k8s.io/kubernetes-dashboard unchanged
rolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrolebinding.rbac.authorization.k8s.io/kubernetes-dashboard unchanged
deployment.apps/kubernetes-dashboard created
service/dashboard-metrics-scraper created
deployment.apps/dashboard-metrics-scraper created

$ kubectl get svc -n kubernetes-dashboard
NAME                        TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
AGE
dashboard-metrics-scraper   ClusterIP   10.100.231.115   <none>        8000/TCP        8s
kubernetes-dashboard        NodePort    10.98.80.251     <none>        443:30959/TCP   8s
```

Nous pouvons désormais accéder à notre dashboard à l’adresse : https://192.168.56.32:30959.

Pensez à remplacer l'adresse ip par celle de votre master.

Un ServiceAccount, « kubernetes-dashboard », a été créé dans le namespace kubernetes-dashboard, nous allons créer un ClusterRoleBinding afin de donner les autorisations nécessaire à ce ServiceAccount, sinon, nous n'aurions aucune ressource visibles sur notre dashboard :

```
$ cat rbac-admin.yml
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: dash-admin-rbac
  namespace: kubernetes-dashboard
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
  - kind: ServiceAccount
    name:  kubernetes-dashboard
    namespace:  kubernetes-dashboard

$ kubectl apply -f rbac-admin.yml
clusterrolebinding.rbac.authorization.k8s.io/dash-admin-rbac created
```

Enfin, depuis la version 1.24 de Kubernetes, les Tokens des ServiceAccounts ne sont plus générés automatiquement, il nous faut donc en créer un :

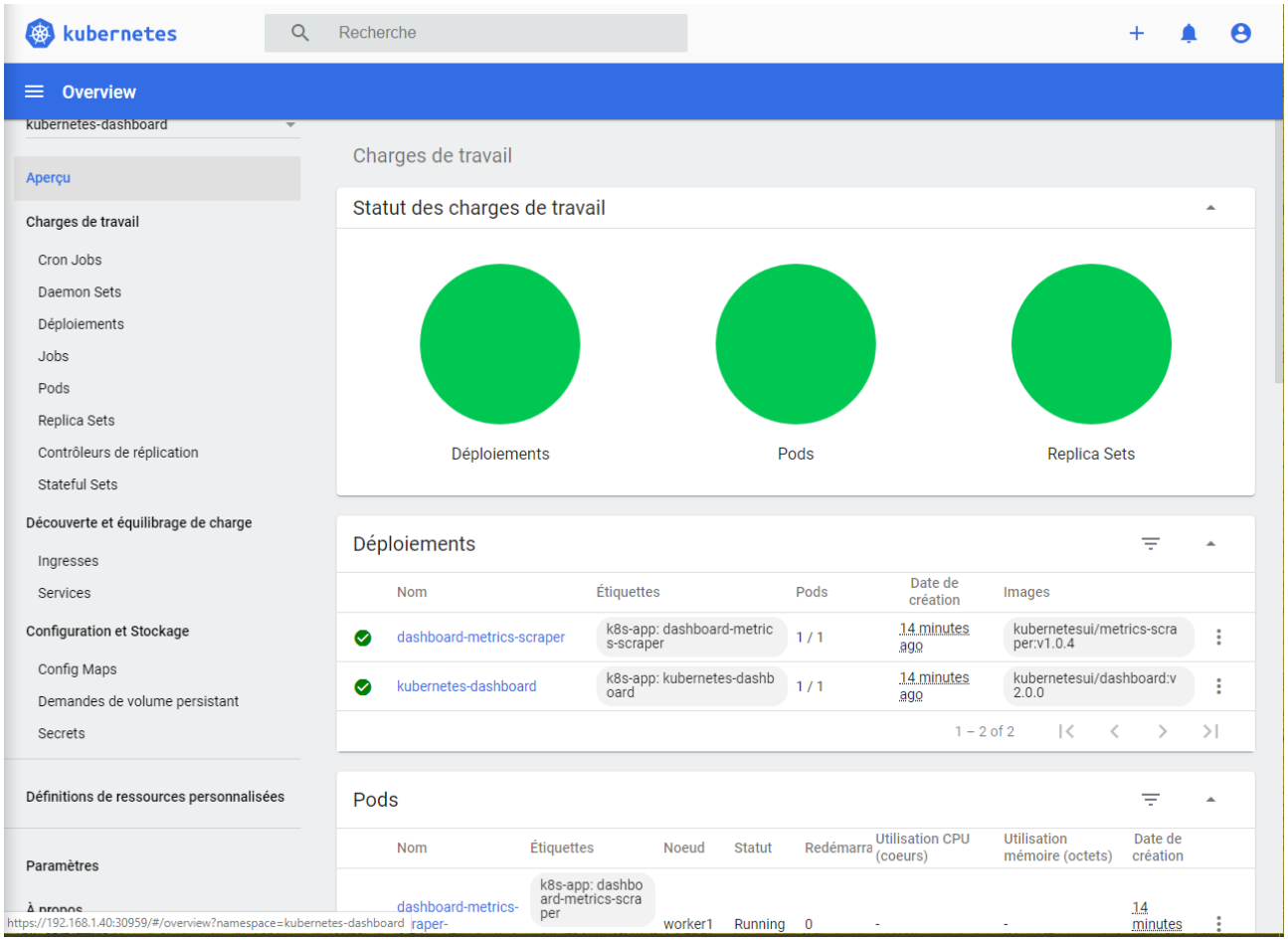
```
$ cat secret.yml
---
apiVersion: v1
kind: Secret
metadata:
  name: token
  namespace: kubernetes-dashboard
  annotations:
    kubernetes.io/service-account.name: kubernetes-dashboard
type: kubernetes.io/service-account-token

$ kubectl apply -f secret.yml

$ kubectl describe -n kubernetes-dashboard token
Name:         token
Namespace:    kubernetes-dashboard
Labels:       <none>
Annotations:  kubernetes.io/service-account.name: kubernetes-dashboard
              kubernetes.io/service-account.uid: d3915121-0cb0-4068-8021-3a108b08bff3

Type:  kubernetes.io/service-account-token

Data
====
namespace:  20 bytes
token:
eyJhbGciOiJSUzI1NiIsImtpZCI6ImFZT2lDNkpPTEVOVHJlaEQ4TnUtbGRKS2JxdjlyT2ZiZkxUOW43SE1KeU0if
Q.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9
uYW1lc3BhY2UiOiJrdWJlcm5ldGVzLWRhc2hib2FyZCIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2Vj
cmV0Lm5hbWUiOiJ0b2tlbiIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2VydmljZS1hY2NvdW50Lm5hb
WUiOiJrdWJlcm5ldGVzLWRhc2hib2FyZCIsImt1YmVybmV0ZXMuaW8vc2VydmljZWFjY291bnQvc2VydmljZS1hY2
NvdW50LnVpZCI6ImQzOTE1MTIxLTBjYjAtNDA2OC04MDIxLTNhMTA4YjA4YmZmMyIsInN1YiI6InN5c3RlbTpzZXJ
2aWNlYWNjb3VudDprdWJlcm5ldGVzLWRhc2hib2FyZDprdWJlcm5ldGVzLWRhc2hib2FyZCJ9.38k0sNzzsl09gXZ
```

```
R0V3d6ouvm_a6YXbCT3zW8wc1vnZdOhDM1ZNuieKsS8ClZ2yZ_mw-ahA-l4hIoi3efeKwrtPjN62xqGf-
MHolZlpRHbRpLSBCiHR6Vi6RttVgWkCy0ChZaOMDQPuTiWYYM2dk6vUjEUInv-
BCHJ1kxZoRmudEY7rH4MTWW6YGS20K_gBXF93MeGDLlzbRzpDfrJF5y2zBb7gH7eKCsK7vuAUPNBwir8aZ3vJeR31
jhH9TICUgukp40RQOcL1KYTVtt_QXzMdpmsaJsiGzsS4YE84XMSBsb8e7bsiH-
LUbgBggL8fr9hLZQrt0OAetTJe3Hw_4eA
ca.crt:     1099 bytes
```

Nous vous laissons explorer cette interface, vous pouvez y retrouver l'ensemble des informations du cluster, ainsi que toutes les ressources qui y existent.

Vous avez également la possibilité de modifier ou supprimer ces ressources, et même d'en créer de nouvelles.

L'utilisation de ce dashboard est assez intuitif.



Annexes

Prometheus / Grafana

- Principes
- Installation
- Utilisation

Prometheus / Grafana

Avant de terminer cette formation, nous allons faire un tour du côté du monitoring, à l'aide d'une stack particulière, puissante et adaptée : Prometheus / Grafana.

Nous verrons l'utilité de tels outils, mais également comment l'installer, et ce qu'il est déjà capable de nous montrer en configuration initiale.

Nous n'irons pas très loin dans l'utilisation, et notamment sur la partie configuration, car ce n'est pas le but de la formation.

Ce chapitre est présent pour vous faire découvrir l'outil et que vous puissiez ensuite le découvrir par vous-même.

Principes :

Prometheus est un outil de collecte de métriques. Il est composé de plusieurs éléments :

- Prometheus server : Le moteur permettant de stocker les métriques collectées.
- Jobs exporters : Ce sont des agents sur lesquels Prometheus viendra collecter les différentes métriques.
- Web UI : L'interface web permettant de visualiser les métriques.
- Alertmanager : Plugin gérant l'envoi d'alertes via divers canaux (Emails, Slack…).

Il va collecter à intervalle régulier des métriques provenant de nos pods/noeuds… et nous permettre de les visualiser.

Il va également les rendre accessibles auprès d'applications de visualisations des données comme Grafana. En effet, même s'il reste possible de visualiser les données récoltées directement sur Prometheus, la visualisation est très rudimentaire et l'on préférera une application avec un meilleur design.

Grafana est donc une application de visualisation de données, qui s'interface très bien avec Prometheus.

Nous allons donc voir comment installer ce couple d'applications.

Installation :

Pour installer Prometheus et Grafana, nous allons une nouvelle fois utiliser Helm.

En effet, même s'il est possible d'installer les applications from scratch, la configuration n'est pas aisée. La communauté Prometheus a mis à disposition sur Helm un charts déjà pré-configuré, et qui comporte de nombreux outils :

- Prometheus et Prometheus-operator.
- Grafana.
- Kube-state-metric (Job exporter permettant de récupérer des métriques sur le cluster Kubernetes en passant par l'API).
- 1 node exporter par nœud (Job exporter permettant de récupérer des métriques systèmes sur les nœuds).
- Alert manager.

Pour pouvoir installer le charts, nous allons devoir utiliser le repo stable de Helm :

```
$ helm repo add stable https://charts.helm.sh/stable
```

Ainsi que celui de la communauté Prometheus :

```
$ helm repo add prometheus-community https://prometheus-
community.github.io/helm-charts
```

Il nous faut également mettre à jour les dépôts Helm :

```
$ helm repo update
```

Nous allons maintenant installer le charts :

```
$ helm install my-monitor prometheus-community/kube-prometheus-stack
NAME: prometheus
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
kube-prometheus-stack has been installed. Check its status by running:
  kubectl --namespace default get pods -l "release=prometheus"

Visit https://github.com/prometheus-operator/kube-prometheus for instructions on how to
create & configure Alertmanager and Prometheus instances using the Operator.

$ kubectl get pods
NAME                                                     READY   STATUS    RESTARTS   AGE
alertmanager-prometheus-kube-prometheus-alertmanager-0   2/2     Running   0          46s
prometheus-grafana-6f6754b569-k6n55                      3/3     Running   0
112s
prometheus-kube-prometheus-operator-599b7945f9-k7bg8     1/1     Running   0
112s
prometheus-kube-state-metrics-548959f4fc-xwvsd           1/1     Running   0
112s
prometheus-prometheus-kube-prometheus-prometheus-0       2/2     Running   0          46s
prometheus-prometheus-node-exporter-6cnpd                1/1     Running   0
112s
prometheus-prometheus-node-exporter-7cq62                1/1     Running   0
111s
prometheus-prometheus-node-exporter-hk5ff                1/1     Running   0
111s
```

Nous voyons qu'un bon nombre de pods a été déployé. Nous allons maintenant nous intéresser à prometheus et grafana.
Regardons tout d'abord les services créés :

```
$ kubectl get svc
NAME                                     TYPE        CLUSTER-IP       EXTERNAL-IP
PORT(S)                      AGE
alertmanager-operated                    ClusterIP   None             <none>
9093/TCP,9094/TCP,9094/UDP   2m20s
kubernetes                               ClusterIP   10.96.0.1        <none>
443/TCP                      108m
prometheus-grafana                       ClusterIP   10.97.176.158    <none>
80/TCP                       2m54s
prometheus-kube-prometheus-alertmanager  ClusterIP   10.103.183.164   <none>
9093/TCP                     2m54s
prometheus-kube-prometheus-operator      ClusterIP   10.101.206.87    <none>
443/TCP                      2m54s
prometheus-kube-prometheus-prometheus    ClusterIP   10.111.82.54     <none>
```

```
9090/TCP                      2m54s
prometheus-kube-state-metrics             ClusterIP   10.104.82.3      <none>
8080/TCP                      2m54s
prometheus-operated                       ClusterIP   None             <none>
9090/TCP                      2m20s
prometheus-prometheus-node-exporter       ClusterIP   10.98.41.198     <none>
9100/TCP                      2m54s
```

Tous les services ont été créés en ClusterIp. C'est fonctionnel, mais ce n'est pas très pratique pour que nous puissions accéder à nos applications. Nous pourrions faire comme pour Wordpress et faire un pull du charts pour pouvoir modifier le fichier value.yaml avant l'installation mais nous allons faire autrement.

Nous allons en effet créer 2 services, en NodePort, pour pouvoir accéder aux applications. Regardons d'abord les ports utilisés par les services :

```
$ kubectl describe svc prometheus-kube-prometheus-prometheus
Name:              prometheus-kube-prometheus-prometheus
Namespace:         default
Labels:            app=kube-prometheus-stack-prometheus
                   app.kubernetes.io/instance=prometheus
                   app.kubernetes.io/managed-by=Helm
                   app.kubernetes.io/part-of=kube-prometheus-stack
                   app.kubernetes.io/version=30.2.0
                   chart=kube-prometheus-stack-30.2.0
                   heritage=Helm
                   release=prometheus
                   self-monitor=true
Annotations:       meta.helm.sh/release-name: prometheus
                   meta.helm.sh/release-namespace: default
Selector:          app.kubernetes.io/name=prometheus,prometheus=prometheus-kube-
prometheus-prometheus
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.111.82.54
IPs:               10.111.82.54
Port:              http-web  9090/TCP
TargetPort:        9090/TCP
Endpoints:         10.244.169.215:9090
Session Affinity:  None
Events:            <none>

$ kubectl describe svc prometheus-grafana
Name:              prometheus-grafana
Namespace:         default
Labels:            app.kubernetes.io/instance=prometheus
                   app.kubernetes.io/managed-by=Helm
                   app.kubernetes.io/name=grafana
                   app.kubernetes.io/version=8.3.4
                   helm.sh/chart=grafana-6.21.0
Annotations:       meta.helm.sh/release-name: prometheus
                   meta.helm.sh/release-namespace: default
Selector:          app.kubernetes.io/instance=prometheus,app.kubernetes.io/name=grafana
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.97.176.158
IPs:               10.97.176.158
Port:              http-web  80/TCP
TargetPort:        3000/TCP
Endpoints:         10.244.169.214:3000
Session Affinity:  None
Events:            <none>
```

Prometheus écoute donc sur le port 9090, et grafana sur le port 3000. Nous récupérons également

les selectors utilisés afin de pouvoir cibler les mêmes pods.

Nous pouvons maintenant déployer nos 2 services :

```
$ cat svc-prometheus.yml
---
kind: Service
apiVersion: v1
metadata:
  name: svc-prometheus
  namespace: monitoring
spec:
  type: NodePort
  ports:
  - port: 9090
    targetPort: 9090
  selector:
     app: prometheus
     prometheus: my-monitor-prometheus-oper-prometheus

$ cat svc-grafana.yml
---
kind: Service
apiVersion: v1
metadata:
  name: svc-grafana
  namespace: monitoring
spec:
  type: NodePort
  ports:
  - port: 3000
    targetPort: 3000
  selector:
    app.kubernetes.io/instance: my-monitor
    app.kubernetes.io/name: grafana

$ kubectl apply -f svc-prometheus.yml
service/svc-prometheus created

$ kubectl apply -f svc-grafana.yml
service/svc-grafana created

$ kubectl get svc -n monitoring
NAME                                      TYPE        CLUSTER-IP       EXTERNAL-IP
PORT(S)                      AGE
alertmanager-operated                     ClusterIP   None             <none>
9093/TCP,9094/TCP,9094/UDP   9m6s
kubernetes                                ClusterIP   10.96.0.1        <none>
443/TCP                      115m
prometheus-grafana                        ClusterIP   10.97.176.158    <none>
80/TCP                       9m40s
prometheus-kube-prometheus-alertmanager   ClusterIP   10.103.183.164   <none>
9093/TCP                     9m40s
prometheus-kube-prometheus-operator       ClusterIP   10.101.206.87    <none>
443/TCP                      9m40s
prometheus-kube-prometheus-prometheus     ClusterIP   10.111.82.54     <none>
9090/TCP                     9m40s
prometheus-kube-state-metrics             ClusterIP   10.104.82.3      <none>
8080/TCP                     9m40s
prometheus-operated                       ClusterIP   None             <none>
9090/TCP                     9m6s
prometheus-prometheus-node-exporter       ClusterIP   10.98.41.198     <none>
9100/TCP                     9m40s
svc-grafana                               NodePort    10.104.173.171   <none>
3000:30106/TCP               12s
svc-prometheus                            NodePort    10.100.107.122   <none>
9090:31365/TCP               20s
```

Nos deux services sont correctement déployés, et nous pouvons accéder à prometheus sur l'ip http://192.168.56.32:31365 et à grafana sur l'ip http://192.168.56.32:30106.

Bien sur, n'oubliez pas de remplacer l'adresse ip par celle de votre master.

Pour Prometheus, c'est terminé, puisque l'accès à la WEB UI est direct.

En revanche, pour grafana, il va nous falloir nous authentifier. Un secret a été généré lors du déploiement contenant le mot de passe du compte admin, nous allons le récupérer :

```
$ pass=$(kubectl get secrets -n monitoring my-monitor-grafana -o yaml|awk
'$1 ~ /^admin-password:/ {print $2}')
$ echo $pass|base64 --decode
prom-operator
```

Voilà,il ne nous reste plus qu'à nous logguer sur l'interface web avec le compte admin.

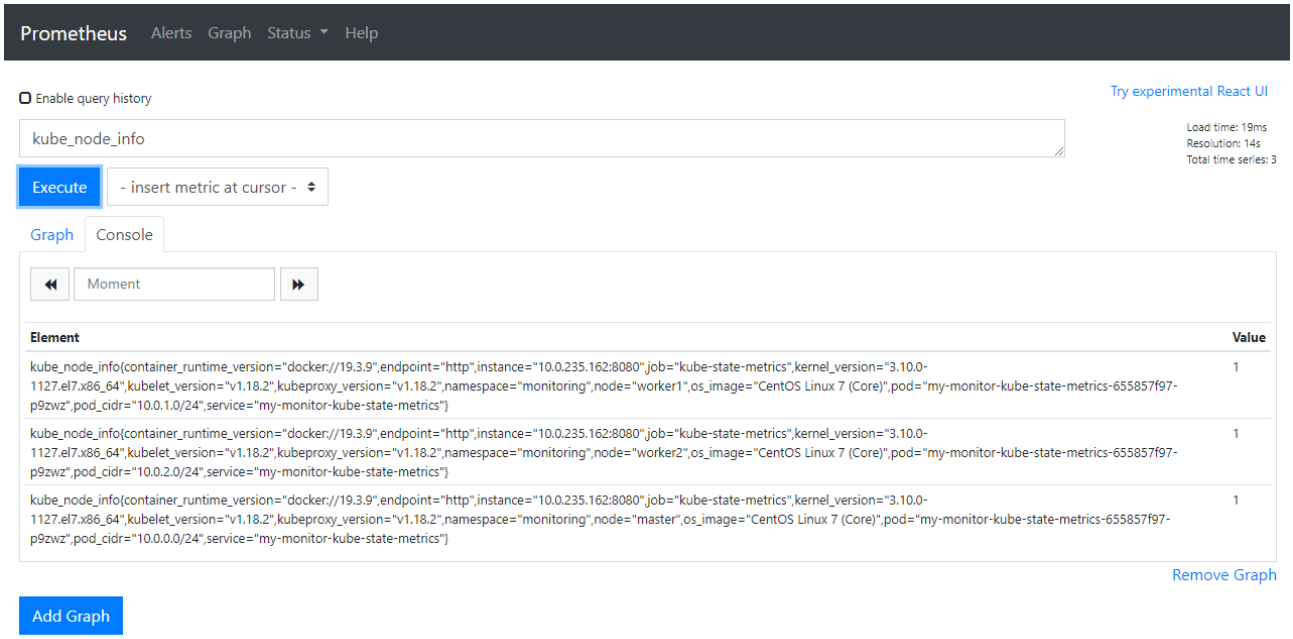
Utilisation :

Nous allons commencer par Prometheus.

Une fois arrivé sur l'interface web, vous avez accès à un champs de recherche. En commençant à taper des lettres à l'intérieur, vous pourrez voir apparaître les différentes métriques qui remontent.

Il suffira d'en sélectionner une et d'appuyer sur Execute pour voir apparaître les valeurs.

Par exemple, la métrique kube_node_info donne le résultat suivant :



Vous voyez que nous avons plusieurs informations qui remontent sur les trois nœuds composant notre cluster.

Une autre métrique intéressante est container_memory_usage_bytes, qui nous permet de connaître l'utilisation RAM des conteneurs présents sur notre Cluster.

Passez sur l'onglet Graph plutôt que console afin d'avoir une meilleure vue.

Il est également possible d'aller voir les alertes pré-configurées qui remontent dans l'onglet Alert, ou encore d'obtenir des informations sur le status de notre instance Prometheus, sur les règles d'alertes qui existent et plus encore.

N'hésitez pas à découvrir les possibilités qu'offrent cet outil vraiment pratique.

Vous remarquerez également qu'un grand nombre de remontées sont faites sans aucune action de notre part. Cela est dû d'une part au fait que le charts est pré-configuré, et d'autre part par l'intégration native de remontées vers Prometheus par des conteneurs que vous pouvez déployer.

En effet, cet outil est tellement démocratisé que bien souvent, les images docker intègrent directement un Job exporter qui expose des métriques que Prometheus peut ensuite récupérer.

Voyons maintenant Grafana.

Une fois la connexion effectuée, vous arrivez sur une page vous demandant d'aller créer des utilisateurs afin de ne pas garder le compte admin par défaut, et allez explorer le repo grafana.

Ce repo comporte un grand nombre de plugin pour s'interfacer avec divers outils, dont prometheus.

Ceci étant, le charts helm est déjà configuré pour que Grafana utilise l'instance de Prometheus que nous avons déployé. C'est notamment pour cela que Grafana ne nous demande pas de passer par la phase « Create a data source », qui consisterait à interfacer notre instance Grafana avec le Prometheus qui tourne sur notre cluster.

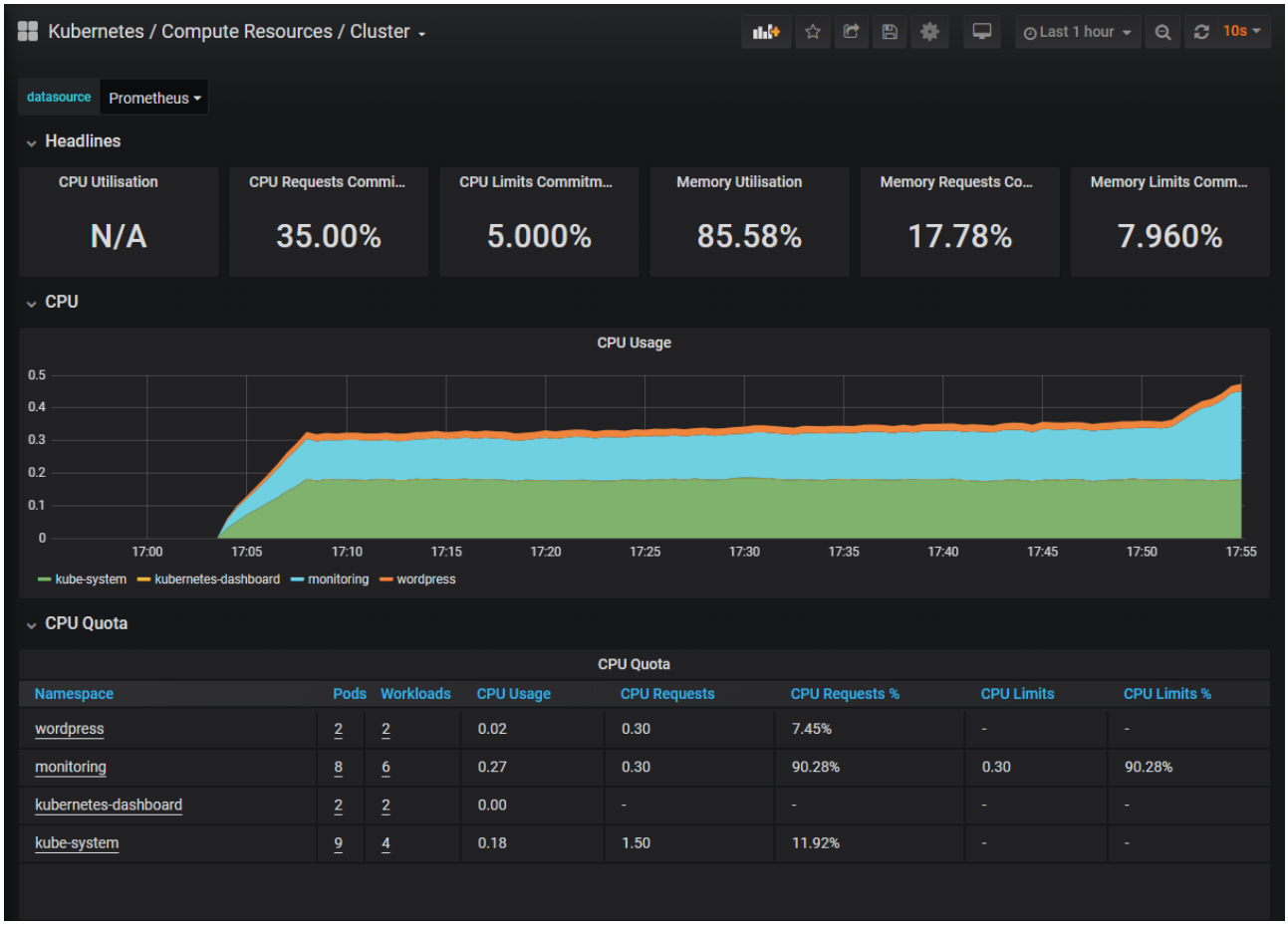
Nous allons donc passer directement au coeur de Grafana, à savoir les dashboards. En effet, Grafana repose sur l'utilisation de Dashboard, qui vont contenir des graphes affichant les données qui nous intéresse.

Il est possible de créer ses propres graphes, mais également d'en récupérer sur le net et de les importer. Concernant notre charts, il arrive avec un certains nombre de Dashboard déjà configurés.

Nous allons en regarder quelques uns.

Cliquez sur le Home en haut de la page web, et vous verrez apparaître la liste des dashboards existants.

Si nous nous rendons par exemple sur le dashboard Kubernetes/Compute Resources/Cluster, nous avons une vu d'ensemble sur les différents namespaces qui composent notre cluster, avec bon nombre d'informations :



Pour une vue détaillée par nœud, nous pourrons nous rendre sur le dashboard Kubernetes/Compute Resources/Node (Pods). En sélectionnant notre datasource en haut « Prometheus » et le node « worker1 », nous pourrons voir ce qui tourne sur ce nœud, combien de ressources sont consommées…

N'hésitez pas à aller faire un tour sur les différents dashboards pour voir la quantité énorme d'informations qui sont remontées.

Ainsi se termine notre tour d'horizon de la stack Prometheus/Grafana, mais n'hésitez pas à creuser le sujet, tant les possibilités sont importantes.

Notes

Fin de session de Formation

Je vous recommande de relire ce support de cours d'ici les deux semaines à venir, et de refaire des exercices.

Il ne vous reste plus qu'à mettre en œuvre ces nouvelles connaissances au sein de votre entreprise.

Merci, et à bientôt.

Jean-Marc Baranger

Theo Schomaker

Steeven Herlant