etna
L'ÉCOLE DES TECHNOLOGIES NUMÉRIQUES AVANCÉES

SDC-168H

Ingénierie des infrastructures du cloud

GoKubeYourself

Rédigé par:

Luis-Edouard SANABRIA
Panna ABDUL HAKIM
Anwar BENIHISSA
Mostafa AIT BENALI
Hémeric AISSI

Contexte de cette documentation

Dans le cadre de notre apprentissage sur les outils du cloud nous avons fait face à la difficulté de trouver des documentations complètes et à jour sur la mise en place de son propre cluster privé kubernetes en utilisant des outils open source kubernetes. La mise en place d'un cluster privé avec des outils open source est un plus pour les étudiants car cela nous permet de nous exercer sans avoir à payer ou à être pressé par des limites de crédits gratuits (Google Cloud) Nous avons donc décidé de mettre à disposition de la communauté une documentation complète sur l'installation et la configuration d'un cluster privé kubernetes avec des outils open sources, nous fournirons également une documentation sur la mise en place de l'intégration et du déploiement continu toujours avec des outils open source.

Nous avons décidé de corser le jeu en mettant en place une application web permettant d'effectuer toutes ces installations en graphique.

Kubernetes

Le tableau suivant présente et définit certains concepts clés de Kubernetes.

Concept	Définition
Cluster	Ensemble de machines qui exécutent des applications conteneurisées gérées par Kubernetes.
Namespace	Un cluster virtuel, dont plusieurs peuvent être pris en charge par un seul cluster physique.
Node	L'une des machines physiques ou virtuelles qui composent un cluster.
Pod	L'objet Kubernetes le plus petit et le plus simple. Un pod représente un ensemble de conteneurs en cours d'exécution sur votre cluster.
Deployment	Un objet API qui gère une application répliquée.
Workload	Les charges de travail sont des objets qui définissent des règles de déploiement pour les pods.

Rancher K3S

1. Introduction à Rancher K3S:

Rancher v2 est issue de l'orchestrateur des conteneurs Kubernetes. Ce changement de technologie sous-jacente pour la v2 est un écart important par rapport à la v1.6, qui prenait en charge plusieurs orchestrateurs de conteneurs populaires. Étant donné que Rancher est désormais entièrement basé sur Kubernetes, il est utile de connaître les bases de Kubernetes.

2. Installer Rancher v2.x

K3s Kubernetes configuration requise :

Ces exigences en matière de CPU et de mémoire s'appliquent à chaque hôte d'un cluster Kubernetes K3s sur lequel le serveur Rancher est installé :

DEPLOYMENT SIZE	CLUSTERS	NODES	VCPUS	RAM	DATABASE SIZE
Small	Up to 150	Up to 1500	2	8 GB	2 cores, 4 GB + 1000 IOPS
Medium	Up to 300	Up to 3000	4	16 GB	2 cores, 4 GB + 1000 IOPS
Large	Up to 500	Up to 5000	8	32 GB	2 cores, 4 GB + 1000 IOPS
X-Large	Up to 1000	Up to 10,000	16	64 GB	2 cores, 4 GB + 1000 IOPS
XX-Large	Up to 2000	Up to 20,000	32	128 GB	2 cores, 4 GB + 1000 IOPS

Vous pouvez commencer le téléchargement de k3s avec la commande suivante:

```
$ curl -sfL https://get.k3s.io | sh -
```

Pour utiliser kubectl en mode user normal il faut donner les droits au fichier:

```
$ chmod 644 /etc/rancher/k3s/k3s.yaml
```

Il faut maintenant ajouter ce fichier dans le fichier .zshrc totalement en bas:

```
$ nano ~/.zshrc
```

```
$ `export KUBECONFIG="/etc/rancher/k3s/k3s.yaml" `
```

Puis lancer la commande suivante pour faire valider la modification:

```
$ source .zshrc
```

La version qui est installée est la version courante de rancher k3s. Or nous voulons la 2.1. Les fichiers relatifs à cette version se trouve dans le répertoire **rancher-2.1**, il va falloir copier tous ces fichiers dans le répertoire manifests de rancher:

```
$ sudo cp -r manifests/ /var/lib/rancher/k3s/server/
```

Et maintenant delete tous les fichiers qui sont dedans afin qu'ils soient de nouveau pris en compte. (Tous les fichiers dans /var/lib/rancher/k3s/server/manifests sont automatiquement appliqués sur le cluster)

```
$ sudo kubectl delete -f /var/lib/rancher/k3s/server/manifests/
```

Le cluster rancher k3s est maintenant disponible dans la version que nous désirons.

Vous pouvez tester la disponibilité du cluster

```
$ kubectl get all
```

Minikube

1- Prérequis

Pour installer Minikube sur votre serveur, celui-ci doit avoir une certaine configuration. Il faut vérifier sur votre serveur que la virtualisation est bien prise en charge avec cette commande :

```
$ grep -E --color 'vmx|svm' /proc/cpuinfo
```

La sortie ne doit pas être vide. Votre serveur doit également avoir au minimum deux CPUs. Une fois que ces prérequis sont vérifiés vous pouvez passer à l’étape suivante.

2- Installer minikube sur votre serveur

Sur votre espace utilisateur vous disposez d'un bouton vous permettant de créer un nouveau cluster sur un serveur en particulier. Cliquez sur celui-ci puis rentrez les informations nous permettant de lancer nos scripts en arrière-plan.
Nous commencerons par installer Docker puis Minikube.
Les scripts d'installation sont disponibles dans les fichiers dockerinstall.sh et install.sh.
Vous pouvez suivre en live l'évolution de l'installation.
Une fois celle-ci terminée nous vous invitons à redémarrer votre serveur.

Helm

Helm est un outil d'empaquetage open source. Il pourra nous aider à installer et à gérer le cycle de vie d'applications Kubernetes. Un peu comme les gestionnaires de package Linux tels que APT et Yum, Helm sert à gérer les graphiques Kubernetes, qui sont des packages de ressources Kubernetes préconfigurés.
Un helm hub est disponible afin de centraliser tous les packages.
Avant de continuer nous devons définir qu'est ce qu'un chart?
Helm utilise un format d'emballage appelé "chart". Un "chart" est un ensemble de fichiers qui décrivent un ensemble connexe de ressources Kubernetes. Un seul "chart" peut être utilisé pour déployer quelque chose de simple, comme un pod memcached, ou quelque chose de complexe, comme une pile d'applications Web complète avec des serveurs HTTP, des bases de données, des caches, etc.

1- Installation et configuration de Helm

La documentation de helm est assez bien fournie. Il suffit d'exécuter les commandes suivantes pour installer helm:

```
$ curl -fsSL -o get_helm.sh
https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3
$ chmod 700 get_helm.sh
$ ./get_helm.sh
```

Les namespaces

Un namespace est un moyen de séparer logiquement un cluster physique en plusieurs clusters virtuels. Nous vous recommandons de créer un namespace pour chaque type d'applications de votre cluster. Ainsi vous pouvez avoir un namespace pour les bases de données, un autre

pour le monitoring ou un autre pour vos API. Vous pouvez également avoir un namespace de test, de développement ou de production.
Pour créer un namespace vous devez créer un fichier .yaml qui se présentera ainsi:

```
apiVersion: v1 # for versions before 1.9.0 use apps/v1beta2
kind: Namespace
metadata:
  name: nom_du_namespace
```

Ensuite lancer la commande pour appliquer:

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

Les volumes persistants

Nous en aurons besoin pour que nos données soient conservées même quand les pods des applications s'éteignent. Le volume est fait pour stocker de manière persistante de la donnée.
Pour en créer un nous aurons besoin d'un Persistant Volume Claim (PVC) afin de demander des ressources. Le PVC est un fichier .yaml se présentant ainsi:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nom_du_pvc
  namespace: namespace_où_déployer_le_pvc
  labels:
    app: nom_de_l_application
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi #vous mettez la taille dont vous avez besoin
```

Il faut ensuite appliquer ce fichier avec:

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

Notre PVC sera lié à l'application qui va l'utiliser grâce au fichier deployment.yaml

Déploiement d'une application

Afin de déployer nos applications nous allons utiliser helm. Helm nous fournit de base des packages d'applications appelés déjà toutes faites. Il est possible de les déployer directement en suivant la documentation liée à l'application que vous désirez. Ici nous allons partir d'un chart vierge et créer notre propre application.

1- Le chart helm

Le chart helm regroupe deux fichiers fondamentaux: les fichiers deploiement.yaml, service.yaml. Un autre fichier est optionnel suivant le besoin: ingressroute.yaml.
Nous allons créer un chart puis le modifier suivant nos besoins. Pour créer un chart helm d'une application postgres par exemple il faut:

```
$ helm create postgres
```

Un dossier postgres est ainsi créé. Il faut ensuite se déplacer dans ce dossier puis dans le dossier "templates" pour avoir les fichiers importants.

2- Deploiement.yaml

Le déploiement contrôle et pilote des Replica Sets et des pods. On y indique l'image à utiliser, le namespace où déployer l'application, le volume à lier à celle-ci et bien d'autres informations. Un deployment.yaml pour une application Postgres se présenterait ainsi:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
  namespace: postgres
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
```

```
      app: postgres
  spec:
    containers:
      - name: postgres
        image: postgres:12
        imagePullPolicy: "IfNotPresent"
        ports:
          - containerPort: 5432
        env:
          - name: POSTGRES_DB
            value: food_green
          - name: POSTGRES_USER
            valueFrom:
              secretKeyRef:
               name: postgres-pass
               key: username
          - name: POSTGRES_PASSWORD
            valueFrom:
              secretKeyRef:
                name: postgres-pass
                key: password
        volumeMounts:
          - mountPath: /var/lib/postgresql/data
            name: postgredb
    volumes:
      - name: postgredb
        persistentVolumeClaim:
          claimName: postgres-pv-claim
```

Ici nous utilisons un secret créé auparavant afin de ne pas afficher les informations sensibles en clair. Afin de créer le secret il faut créer un fichier .yaml ainsi:

```
apiVersion: v1
kind: Secret
metadata:
  name: postgres-sonar-pass
  namespace: sonarqube
type: Opaque
data:
  username: c29uYXI=
```

```
  password: UXFGWm4wdFVL
```

Et appliquer le fichier:

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

Pour plus de détails sur les secrets:
https://kubernetes.io/fr/docs/concepts/configuration/secret/#cr%C3%A9er-vos-propres-secrets

3- Service.yaml

Le service décrit la façon dont un pod ou un groupe de pods peut être accédé via le réseau interne. Toujours pour notre application postgres il se présente ainsi:

```
apiVersion: v1
kind: Service
metadata:
  name: postgres
  namespace: postgres
  labels:
    app: postgres
spec:
  ports:
   - port: 5432
  selector:
   app: postgres
```

On y retrouve principalement le nom de l'application à laquelle on veut accéder et le port d'accès.

Pour une application sans interface web les fichiers deployment.yaml et service.yaml suffisent. Quand il s'agit d'accéder à une interface graphique depuis l'extérieur il faut faire appel à un troisième fichier: ingressroute.yaml.

4- Ingressroute.yaml

Un Ingress est un objet Kubernetes qui gère l'accès externe aux services dans un cluster, généralement du trafic HTTP.

Un Ingress peut fournir un équilibrage de charge, une terminaison TLS et un hébergement virtuel basé sur un nom.

Utiliser ses propres images

Nous vous rappelons qu'une image Docker est une sorte d'image disque qui contient l'arborescence d'une distribution linux, le code source d'une application et des binaires capables de la faire tourner.
Docker met à disposition un registre d'images publiques : Docker Hub. Vous pouvez y stocker des images mais celles-ci seront automatiquement publiques. Et pour un projet privé rien de mieux qu'un registre privé.
A ce propos Gitlab nous propose gratuitement un registre privé pour nos projets.
Vous pouvez gratuitement créer un compte sur gitlab.com, y créer un projet et profiter du registre qui y est lié.

1- Créer son image Docker

L'image Docker se construit à partir d'un fichier du nom Dockerfile que l'on retrouve généralement à la racine des sources d'une application.
Une fois le Dockerfile créé il faut build l'application donc lancer cette commande à la raçine du projet:

```
$ docker build -t registry.gitlab.com/username_gitlab/nom_app:tag .
```

Une fois l'image créée il faut s'identifier sur le registre de Gitlab:

```
docker login registry.gitlab.com
```

Et renseigner ses informations de connexion.
Une fois cela fait on peut maintenant push l'image sur le registre:

```
$ docker push registry.gitlab.com/username_gitlab/nom_app:tag
```

2- Utiliser les images du registre privé

Il semblerait que les clusters ne font confiance qu'aux images venant du docker hub mais on peut régler ça. Avant d'utiliser les images d'un registry privé il faut créer un secret, un imagepullsecret:

```
$ kubectl create secret generic nom_du_secret
--from-file=.dockerconfigjson=$HOME/.docker/config.json
--type=kubernetes.io/dockerconfigjson -n <le namespace spécifique>
$ kubectl patch serviceaccount default -p '{"imagePullSecrets": [{"name":
"project_name"}]}' -n <le namespace spécifique>
```

La CI/CD avec Gitlab

L'intégration continue va permettre de lancer vos tests et vos builds directement sur le serveur via des pipelines. Les pipelines sont des groupes de jobs qui vont définir les scripts à exécuter sur le serveur. Pour gérer vos pipelines, il faut mettre en place un GitLab Runner. Le GitLab Runner va gérer vos jobs et les lancer automatiquement quand une branche sera envoyée sur le dépôt ou lorsqu'elle sera mergée, par exemple. Vous pouvez également lancer les jobs à la main ou changer complètement la configuration.

Pour l'installation suivre manuelle suivre la documentation de gitlab. Pour l'avoir testée nous ne vous la conseillons pas.

1- Connecter son cluster kubernetes à Gitlab

Une autre manière d'installer le gitlab runner est de le faire en mode graphique. On va commencer par relier notre cluster à gitlab en allant dans: "Opération -> kubernetes", on choisit d'ajouter un cluster et on suit cette documentation de gitlab.
Une fois le cluster ajouté il faut installer l'application "Gitlab runner". Les erreurs rencontrées sont basiques et on peut trouver des résolutions en faisant des logs sur le pod qui pose problème

2- Le fichier .gitlab-ci.yaml

Dans ce fichier nous avons mis tous les détails pour les jobs buildImage et deploy. Ces jobs utilisent le gitlab-runner créé précédemment.

La partie buildImage utilise kanico. Au début nous utilisions docker pour build une image docker mais avions du mal avec le job suivant.

kaniko est un outil pour construire des images docker à partir d'un Dockerfile, à l'intérieur d'un conteneur ou d'un cluster Kubernetes.

kaniko résout deux problèmes avec la méthode de construction de Docker-in-Docker :

- Docker-in-Docker nécessite un mode privilégié pour fonctionner, ce qui est un problème de sécurité important.
- Docker-in-Docker encourt généralement une pénalité de performance et peut être assez lent.

La partie build utilise une image helm. On fait un upgrade de l'application utilisant l'image qui vient d'être créée.

Le monitoring de ses pods

1- Prometheus

Prometheus est une boîte à outils open source de surveillance et d'alerte des systèmes conçue à l'origine par SoundCloud.

2- Grafana

Grafana vous permet d'interroger, de visualiser, d'alerter et de comprendre vos métriques, peu importe où elles sont stockées. Créez, explorez et partagez des tableaux de bord avec votre équipe et favorisez une culture axée sur les données.

3- Installation de l'opérateur prometheus

Nous allons installer Prometheus et Grafana sur le cluster K3S en utilisant Helm.
Ensuite, assurez-vous que votre cluster K8S est opérationnel. Utilisez Helm pour installer prometheus-operator avec cette commande :

```
helm install stable/prometheus-operator --generate-name
```

Avec cet opérateur, nous pouvons voir que Prométhée et Grafana sont déjà exposés, mais uniquement en interne. Tout ce que nous avons à faire est de modifier les ServiceTypes de ClusterIP vers LoadBalancer.
Modifions le service de Prometheus et modifions le type de **ClusterIP** à **LoadBalancer**. Notez que votre nom de service est différent du mien.

```
kubectl edit svc prometheus-operator-xxxxx-prometheus
```

Une fois terminé, vérifiez à nouveau le service. Ouvrez l'URL ELB dans le navigateur pour confirmer qu'il fonctionne correctement.

Ensuite, répétez l'étape pour exposer Grafana sur ELB.

La dernière étape consiste à obtenir le nom d'utilisateur et le mot de passe de Grafana.

```
kubectl get secret prometheus-operator-xxxxxx-grafana -o
```

```
jsonpath="{.data.admin-password}" | base64 --decode ; echo
```

Le compte par défaut est admin.