
kubernetes
by Google

Votre formation – Présentation

Public :

- Développeurs, architectes techniques, administrateurs et responsables d'exploitation et de production, chefs de projet.

Objectifs :

- Comprendre le positionnement de Kubernetes et la notion d'orchestration
- Installer Kubernetes et ses différents composants
- Utiliser les fichiers descriptifs YAML
- Définir les bonnes pratiques pour travailler avec Kubernetes

Votre formation – Présentation

Pré-requis :

- Connaissances systèmes Linux/Windows
- Notions sur les réseaux TCP/IP
- Utilisation de la ligne de commande et du script Shell en environnement Linux.

Formateur Kubernetes

Votre formation - Logistique

Horaires de la formation

- 9h00-12h30
- 14h00-17h30

Pauses

- 2 x 15 min

Merci d’éteindre vos téléphones portables

Votre formation - Programme

Introduction à Kubernetes

Les solutions d’installation

Les fichiers descriptifs

Architecture de Kubernetes

Objets de base

Exploiter Kubernetes

Gestion avancée des conteneurs

Kubernetes en production

Introduction à Kubernetes

De la virtualisation à conteneurisation.

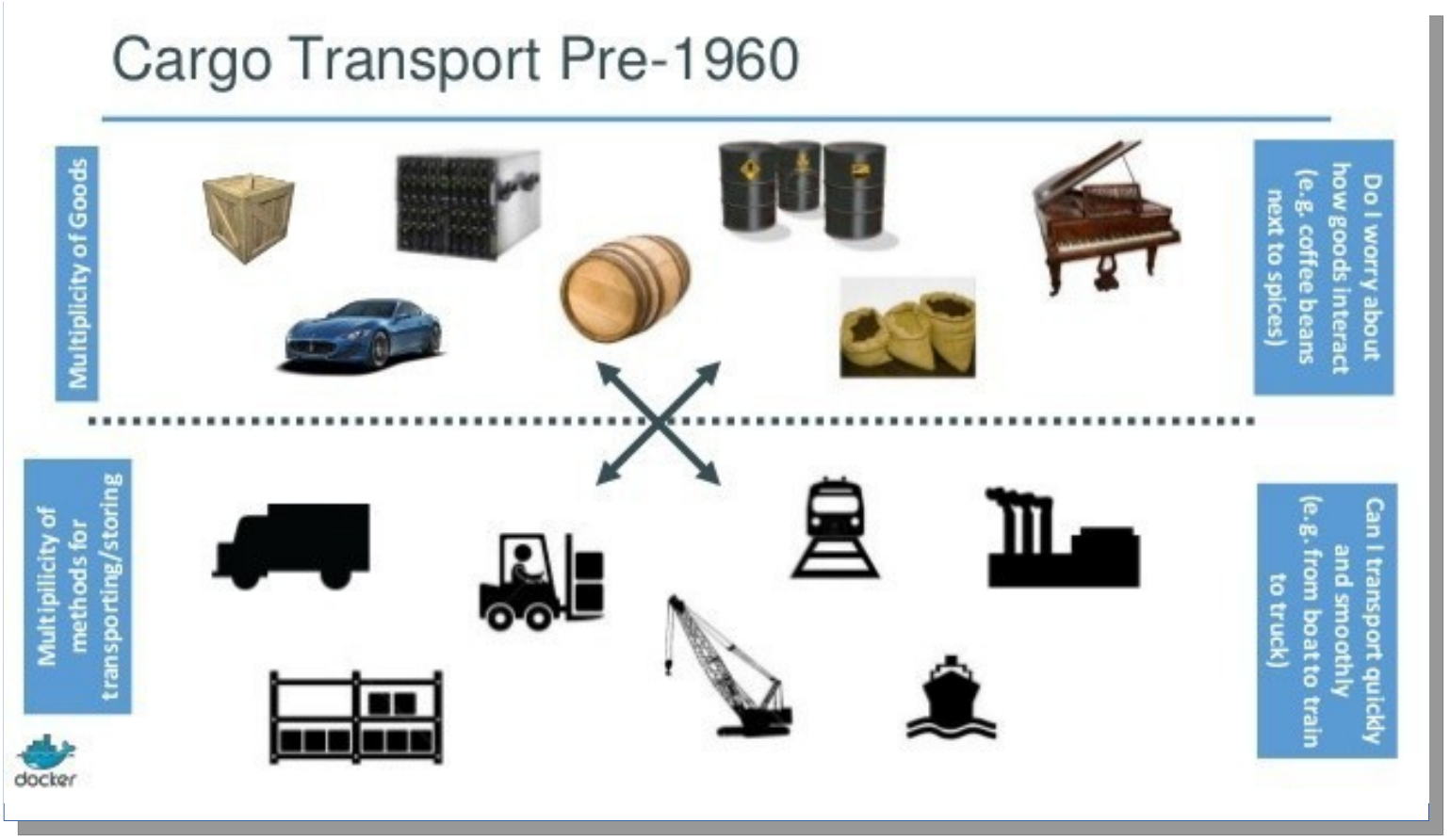
Le couple Docker/Kubernetes.

De la virtualisation à Docker

L'architecture des applications change rapidement

De la virtualisation à Docker

Analogie – origine du nom



De la virtualisation à Docker

Analogie – origine du nom

De la virtualisation à Docker

Solution: Des conteneurs Docker

- Package : une application et ses dépendances
- Isoler les applications les unes des autres
- Agnostique sur le contenu
- Création d'un standard pour le format des containers (OCI)
- Facilement portable sur différents environnements
- Flexibilité/modularité des composants
- Ex. toolkit comme https://mobyproject.org/
- Automatisable/Scriptable

De la virtualisation à Docker

Du point de vue du développeur ...

- Build once… run anywhere
- un environnement portable pour l'exécution des apps
- Pas de risque d'oublier des dépendances, packages, … durant les déploiements
- Chaque application s'exécute dans son propre conteneur : avec ses propres versions des librairies
- Facilite l'automatisation des tests
- élimine les problèmes de compatibilité entres les plate-formes
- Coût ressource très bas pour lancer un container. On peut en lancer des dizaines sur un poste développeur (laptop)
- Permet de tester des technologies ou faire des prototypes rapidement et à très bas coût.

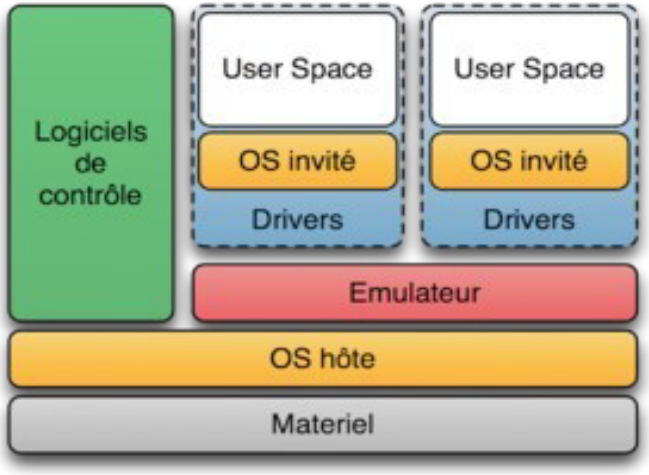
De la virtualisation à Docker

Du point de vue de l'admin sys ...

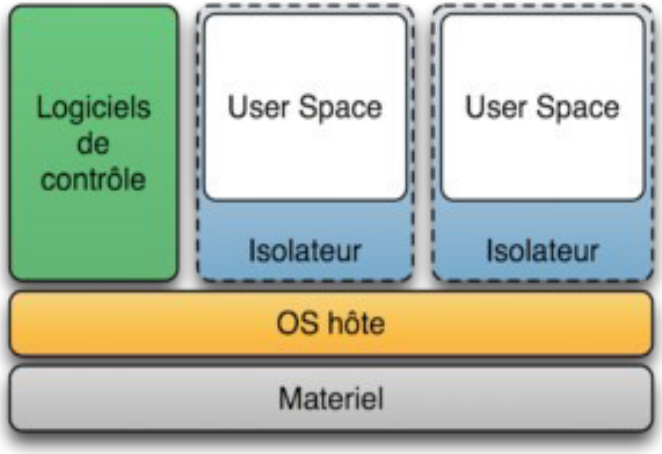
- Configure once…run anything
- Rend le workflow plus consistant, prédictible, répétable
- Élimine les inconsistances entre les environnements de dev/test/prod
- Améliore la rapidité et la fiabilité du déploiement continu (continuous deployment)
- Réduction des pb de performances (Ex. avec les VM); réduction des coûts (hébergements cloud, …)

De la virtualisation à Docker

Comparaison



Virtualisation



Conteneurs

- Pas seulement Docker : terme générique désignant les technologies intégrées au noyau Linux depuis ~10 ans
- Les conteneurs sous Microsoft Windows Server depuis Windows 2016 serveur

De la virtualisation à Docker

~~Virtualisation~~ → Conteneurs

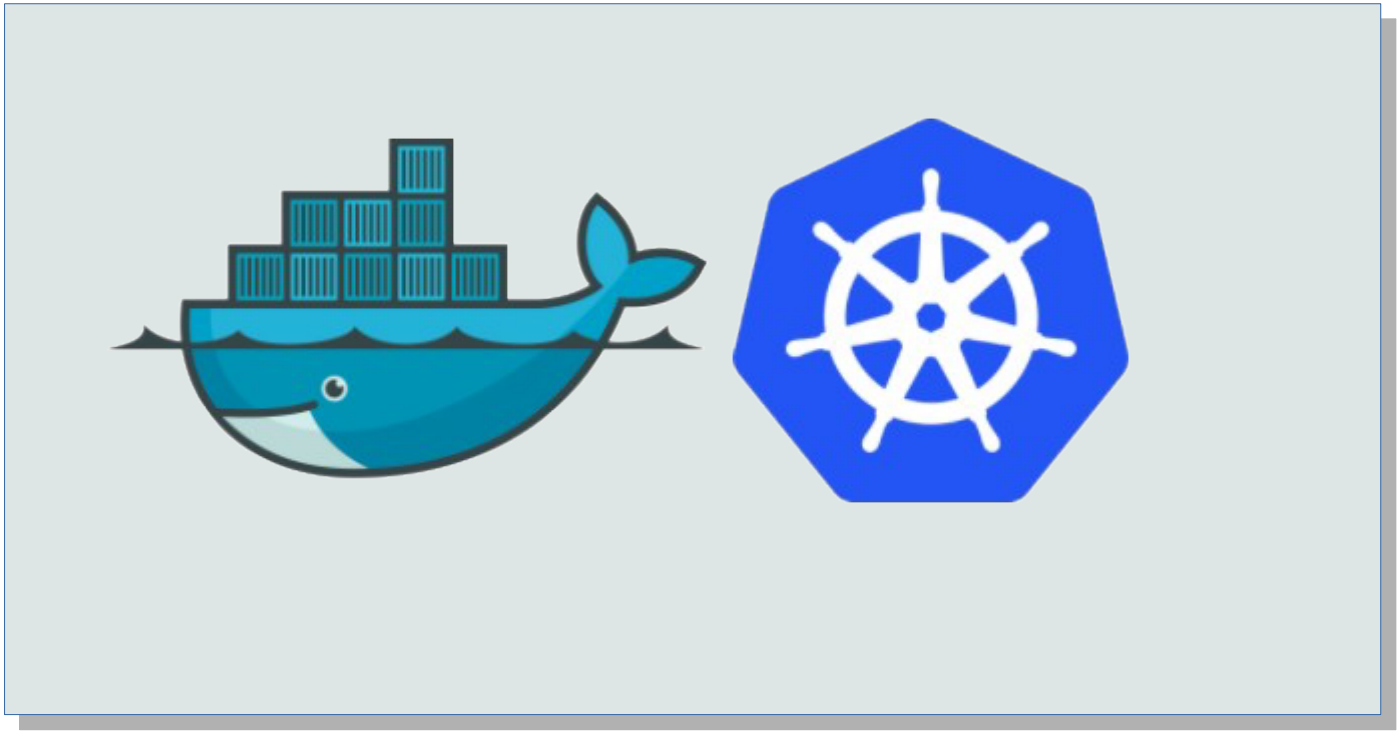
- Le système invité partage le noyau du système host
- Très performant : tout passe par le même noyau
- Peu coûteux : s'intègre au host, peut se partager sur plusieurs serveurs
- Peu isolant : une faille peut mener à une prise de privilège (*escalation*)
- Utile du développement à la production en passant par les tests, solution de plus en plus utilisée

De la virtualisation à Docker

Qu'est-ce que Kubernetes ?

- Kubernetes est un système permettant d'exécuter et de coordonner des applications conteneurisées sur un cluster de machines.
- Il gère le cycle de vie des applications et services conteneurisés à l'aide de méthodes qui offrent prévisibilité, évolutivité et haute disponibilité.
- En tant qu'utilisateur de Kubernetes, vous pouvez :
  - définir comment vos applications doivent fonctionner
  - comment elles doivent pouvoir interagir avec d'autres applications ou avec le monde extérieur.
  - faire évoluer vos services vers le haut ou vers le bas
  - effectuer des mises à jour progressives
  - basculer le trafic entre les différentes versions de vos applications.

Le couple Docker/Kubernetes.



Le couple Docker/Kubernetes.

- Une des sociétés à avoir démocratisé l'utilisation des conteneurs Linux est **Docker**.
- **Docker** n'a pas créé la technologie : c'est un ensemble d'outils et d'API qui ont rendu les conteneurs beaucoup plus facilement gérables.
- Le succès de Docker est dû au fait qu'il est arrivé au bon moment, pile quand l'industrie cherchait un moyen de mieux gérer le Cloud et ses workloads Web.
- **Docker** est plus qu'une trousse à outils. La société a aussi fédéré tout un écosystème, foisonnant, qui a commencé à contribuer à un ensemble divers d'outils de gestion des cycles de vie des conteneurs.
- En juin 2014, la DockerCon, la première conférence jamais organisée sur le sujet, a par exemple pu se targuer de la présence d'acteurs majeurs comme Google, IBM, Microsoft, Amazon, Facebook, Twitter ou Rackspace, tous venus témoigner de leurs utilisations des conteneurs.

Le couple Docker/Kubernetes.

Comment Google fait tourner des workloads critiques dans des conteneurs ?

- Si Docker est dans la lumière des projecteurs, une autre entreprise – qui maîtrise l'art du dimensionnement – est passé maître dans l'art d'utiliser les conteneurs pour ses workloads de production. Cette société s'appelle **Google**.
- Google gère en effet chaque semaine plus de **deux milliards de conteneurs**. Des services comme Gmail, Search, Apps ou Maps tournent tous dans des conteneurs.
- Demandez à n'importe quel administrateur qui a eu à gérer plusieurs centaines de VMs et il vous dira que c'est un vrai cauchemar.
- Google a ainsi conçu plusieurs outils pour gérer un nombre très important de conteneurs. Lorsqu'il s'est lancé en tant que fournisseur de Cloud (avec App Engine et Compute Engine) Google a ouvert ces outils de gestion aux développeurs. Ce qui bénéficie à la communauté mais qui permet aussi de différencier Google Cloud Platform.

Le couple Docker/Kubernetes.

Kubernetes : une nouvelle ère pour les conteneurs

- Un des principaux outils que Google a rendu open-source s'appelle **Kubernetes** - qui signifie en grec *pilote* ou *barreur*.
- **Kubernetes fonctionne en complément de Docker**.
  - Docker permet de gérer le cycle de vie des conteneurs,
  - Kubernetes apporte l'orchestration et la gestion de clusters de conteneurs.
- Les clients qui provisionnent des VMs sur AWS, Azure ou sur n'importe quel Cloud Public, n'ont que faire de ce qui se passe du côté du hardware. Du moment qu'ils ont les VMs qu'ils souhaitent, avec les bonnes performances qui vont avec, ils ne se préoccupent pas de savoir comment cela marche « en dessous ». Savoir si les serveurs physiques d'AWS sont de chez HP, Dell ou IBM est sans importance.

Solutions d'installation

- Un bon moyen de commencer à travailler avec Kubernetes, c'est d'utiliser **MicroK8s, Minikube, K3s, Docker** ou **Kind**.
- MicroK8s sera déployé et utilisé durant la formation
- MicroK8s est un Low-Ops, production minimale de Kubernetes,
  - pour les développeurs, le cloud, les clusters, les stations de travail, Edge et IoT.
- MicroK8s permet d'avoir un cluster léger hautement disponible

Solutions d'installation

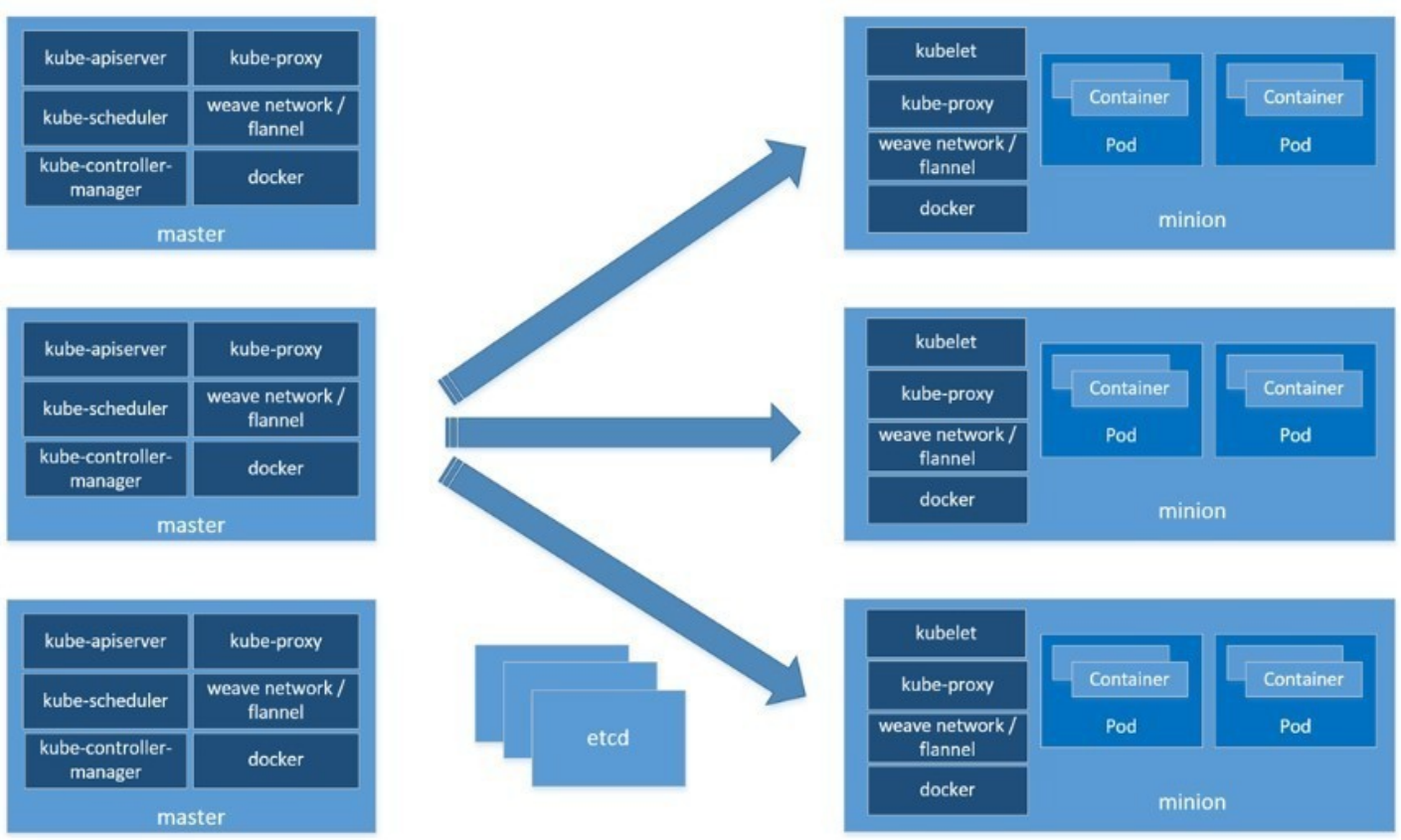
MicroK8s

Une nouvelle solution pour exécuter un cluster local Kubernetes léger.

- Développé par l'équipe Kubernetes de Canonical.
- Conçu pour être une installation Kubernetes en amont rapide et légère, isolée de votre environnement local.
  - Cet isolement est obtenu en empaquetant tous les binaires pour Kubernetes, Docker.io, iptables et CNI dans un seul package snap (disponible uniquement dans Ubuntu et les distributions compatibles).

Solutions d'installation

On-Premise



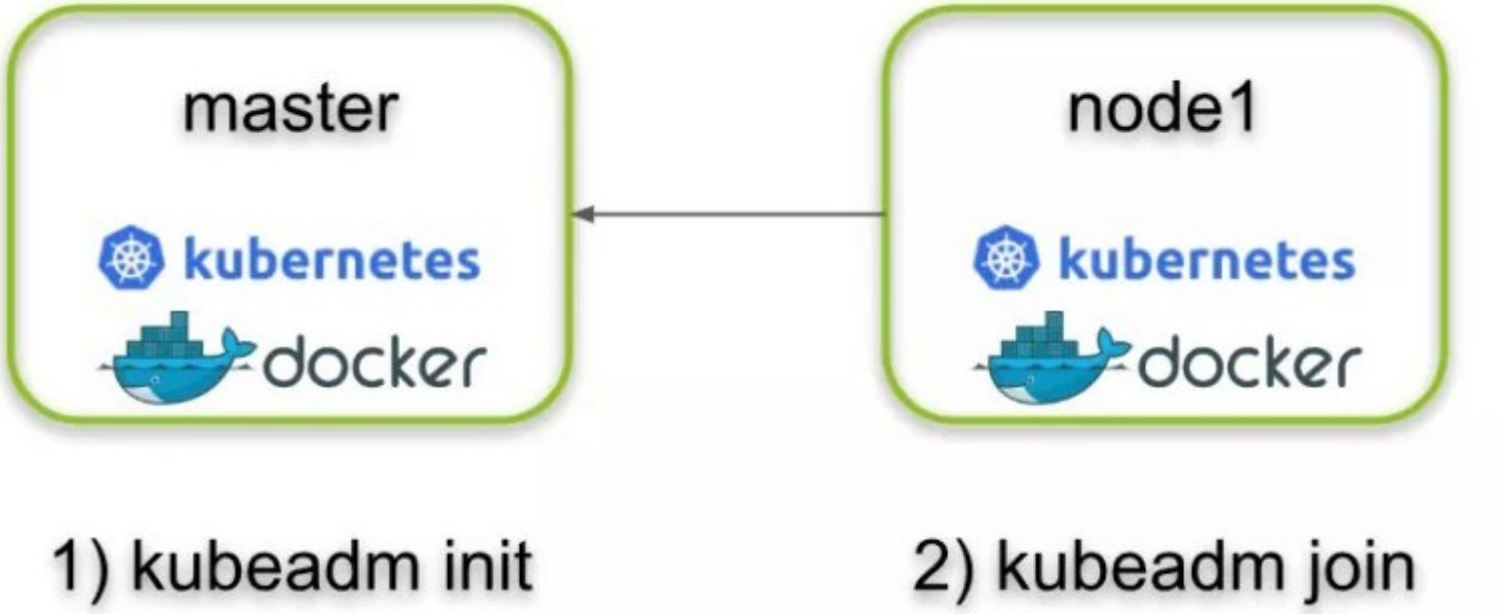
Solutions d'installation

Kubespray à la rescousse

- **Kubespray** utilise **Ansible** ou script **Terraform** pour déployer un cluster Kubernetes hébergé sur GCE, Azure, OpenStack, AWS, vSphere, Packet (bare metal), Oracle Cloud Infrastructure (expérimentale) ou Baremetal.

Solutions d'installation

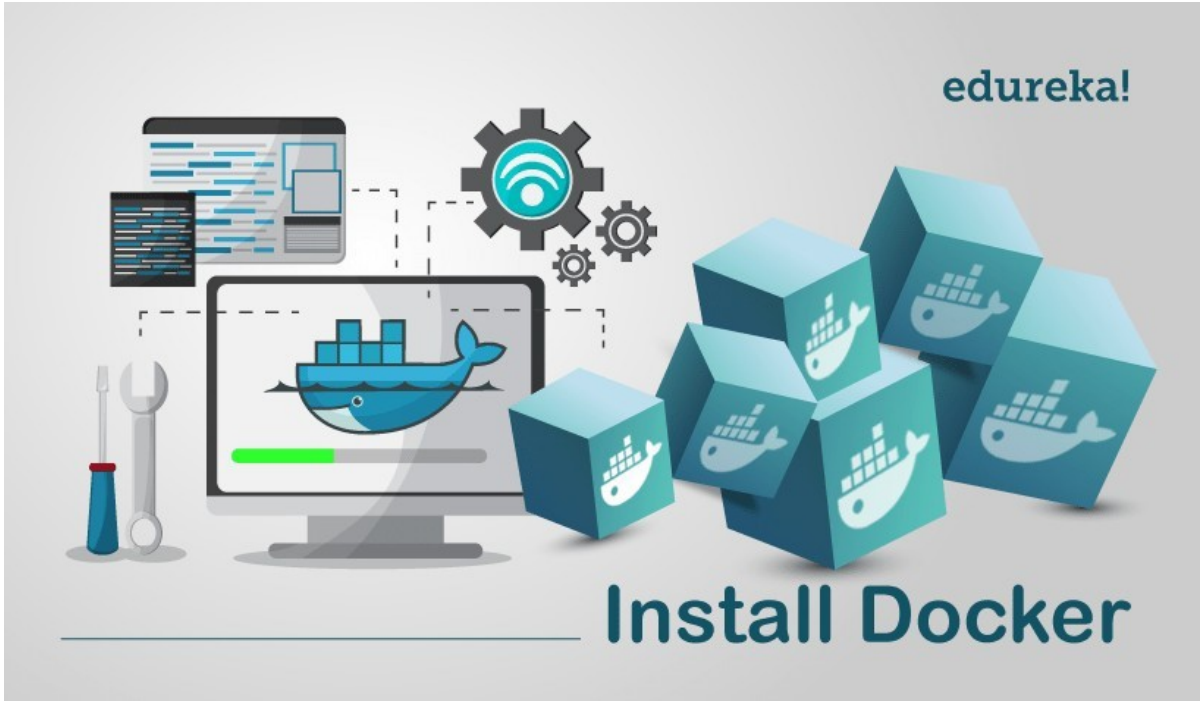
Déploiement et publication manuelle.



Solutions d'installation

Installation et configuration de docker.

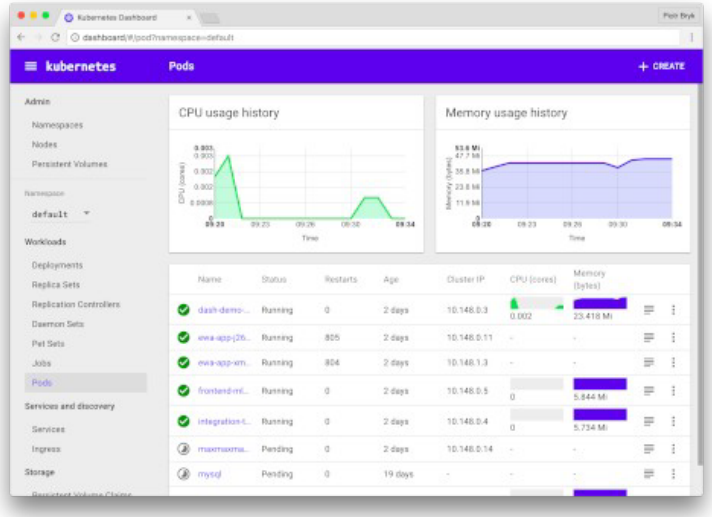
- Il convient d'installer en premier lieu Docker car ***K8s*** (Kubernetes) utilise Docker

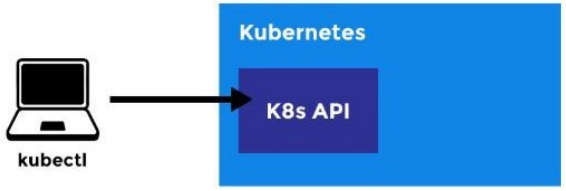


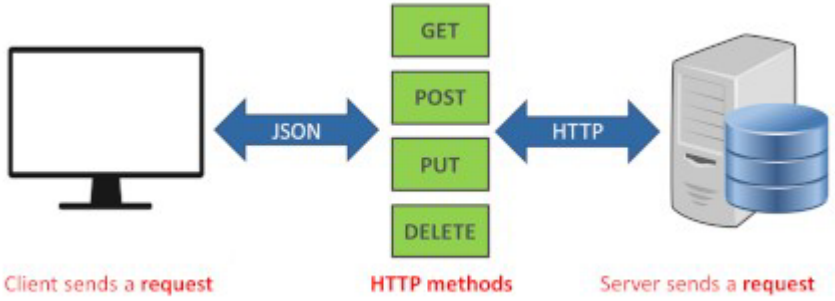
Accéder à Kubernetes

Accéder au cluster Kubernetes :

- CLI (kubectl),
- GUI (dashboard)
- APIs.
-







Solutions d'installation

Détail et introspection de Microk8s

Inspecting Certificates

Inspecting services

Service snap.microk8s.daemon-cluster-agent is running

Service snap.microk8s.daemon-containerd is running

Service snap.microk8s.daemon-apiserver is running

Service snap.microk8s.daemon-apiserver-kicker is running

Service snap.microk8s.daemon-control-plane-kicker is running

Service snap.microk8s.daemon-proxy is running

Service snap.microk8s.daemon-kubelet is running

Service snap.microk8s.daemon-scheduler is running

Service snap.microk8s.daemon-controller-manager is running

Copy service arguments to the final report tarball

Inspecting AppArmor configuration

Gathering system information

Copy processes list to the final report tarball

Copy snap list to the final report tarball

Copy VM name (or none) to the final report tarball

Copy disk usage information to the final report tarball

Copy memory usage information to the final report tarball

Copy server uptime to the final report tarball

Copy current linux distribution to the final report tarball

Copy openSSL information to the final report tarball

Copy network configuration to the final report tarball

Solutions d'installation

Détail et introspection de Microk8s

```
Inspecting kubernetes cluster
  Inspect kubernetes cluster

Inspecting juju

  Inspect Juju

Inspecting kubeflow

Inspect Kubeflow



Building the report tarball
  Report tarball is at /var/snap/microk8s/2074/inspection-

report20210317_124235.tar.gz
```

Premiers essais

Bonjour le monde en version MicroK8s:

- Une fois que votre cluster Kubernetes est en cours d'exécution et que **kubectl** est configuré, vous pouvez exécuter votre première application en quelques étapes. Cela peut être fait en utilisant *les commandes impératives* qui n'ont pas besoin de fichiers de configuration.
- Pour exécuter une application, vous devez fournir un nom de déploiement (nginx), l'emplacement de l'image du conteneur (easylinux/kubernetes:nginx) et le port (80)

Premiers essais

Exemple :

Si vous utilisez un cluster personnalisé Kubernetes (Microk8s, Kubeadm ou autre). Il n'y a pas de LoadBalancer intégré (contrairement à AWS ou Google Cloud). Dans ce cas vous devez utiliser NodePort ou un contrôleur Ingress.

```
$ kubectl create deployment nginx –image=easylinux/kubernetes:nginx
deployment.apps/nginx created

$ kubectl get deployments
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
nginx   1/1     1            1           13s

$ kubectl expose deployment/nginx --type=NodePort --port 80
service/bootcamp exposed
```

Premiers essais

$ **kubectl get services**

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
nginx	LoadBalancer	10.97.48.179	<none>	80:31497/TCP	39s
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	6d22h

$ **kubectl describe service nginx**

```
Name:                     nginx
Namespace:                default
Labels:                   app=nginx
Annotations:              <none>
Selector:                 app=nginx
Type:                     NodePort
IP:                       10.97.48.179
Port:                     <unset>  80/TCP
TargetPort:  NodePort:    80/TCP
Endpoints:  Session       <unset>  31497/TCP
Affinity:                 172.17.0.6:80
External Traffic Policy: Events:  None
                          Cluster
                          <none>
```

$ microk8s enable dashboard

$ microk8s dashboard-proxy

Premiers essais

Scale Up

Une fois que cela a fonctionné, vous pouvez multiplier votre application avec :

```
$ kubectl scale deployment/nginx --replicas=4
```

Et vérifiez le résultat avec :

```
$ kubectl get deployments
NAME            DESIRED         CURRENT   UP-TO-DATE   AVAILABLE   AGE
Nginx           4               4         4            4           30s
```

Pour basculer sur une nouvelle version de l'application, exécutez :

```
$ kubectl set image deployment/nginx nginx=easylinux/kubernetes:nginx-v2
```

Le nettoyage est enfin fait avec :

```
$ kubectl delete deployment nginx

$ kubectl delete service nginx
```

Premiers essais

Suppression d'un déploiement.

Si une application n’est plus nécessaire, il suffit de la supprimer.

```
$ kubectl delete service backend
$ kubectl delete deployment backend
```

NB : pas d’équivalent à docker stop car il n’y a pas de sens en production à ‘suspendre’ une application.

Travaux pratiques

Déploiement d'une plateforme de test.

- Validation du poste de travail
- Mise en œuvre Microk8s
- Tests de fonctionnement

Les fichiers descriptifs

Syntaxe.

- **YAML**, acronyme de **Y**et **A**nother **M**arkup **L**anguage dans sa version 1.01. C'est un format de représentation de données par sérialisation Unicode.
- YAML a été proposé par Clark Evans en 2013, et implémenté par ses soins ainsi que par Brian Ingerson et Oren Ben-Kiki.
- Son objet est de représenter des informations plus élaborées que le simple CSV en gardant cependant une lisibilité presque comparable, et bien plus grande en tout cas que du XML.

Les fichiers descriptifs

Syntaxe.

- L'idée de YAML est que presque toute donnée peut être représentée par combinaison de listes, tableaux associatifs (dictionnaires) et données scalaires. YAML décrit ces formes de données (les représentations YAML), ainsi qu'une syntaxe pour présenter ces données sous la forme d'un flux de caractères (le flux YAML).

Les fichiers descriptifs

Syntaxe.

- La syntaxe du flux YAML est relativement simple, efficace, moins verbeuse que du XML, moins compacte cependant que du CSV.
  - Les commentaires sont signalés par le signe dièse (**#**) et se prolongent sur toute la ligne. Si par contre le dièse apparaît dans une chaîne, il signifie alors un nombre littéral.
  - Il est possible d'inclure une syntaxe JSON dans une syntaxe YAML.
  - Les éléments de listes sont dénotés par le tiret (**-**), suivi d'une espace, à raison d'un élément par ligne.
  - Les dictionnaires sont de la forme clé: valeur, à raison d'un couple par ligne.
  - Les scalaires peuvent être entourés de guillemets doubles (**"**), ou simples (**'**), sachant qu'un guillemet se banalise avec un antislash (\), alors qu'une apostrophe se banalise avec une seconde apostrophe.
  - L'indentation, par des **espaces**, manifeste une arborescence.

La syntaxe YAML se distingue de JSON par le fait qu'il se veut plus facilement lisible par une personne. Il se distingue du XML par le fait qu'il s'intéresse d'abord à la sérialisation de données, et moins à la documentation.

Les fichiers descriptifs

Syntaxe

- Exemple :

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: petstore
    annotations:
        sidecar.istio.io/inject: "false"
    spec:
        selector:
            matchLabels:
                app: petstore
            template:
                metadata:
                    labels:
                        app: petstore
                        version: v1

                spec:
                    containers:
                        - name: front
                          image: dgageot/lecacy-docker-front:v1
                        - name: petstore
                          image: easylinux/petstore
                          imagepullPolicy: IfNotPresent
```

Desired state # Current(actual) state of a Kubernetes Object

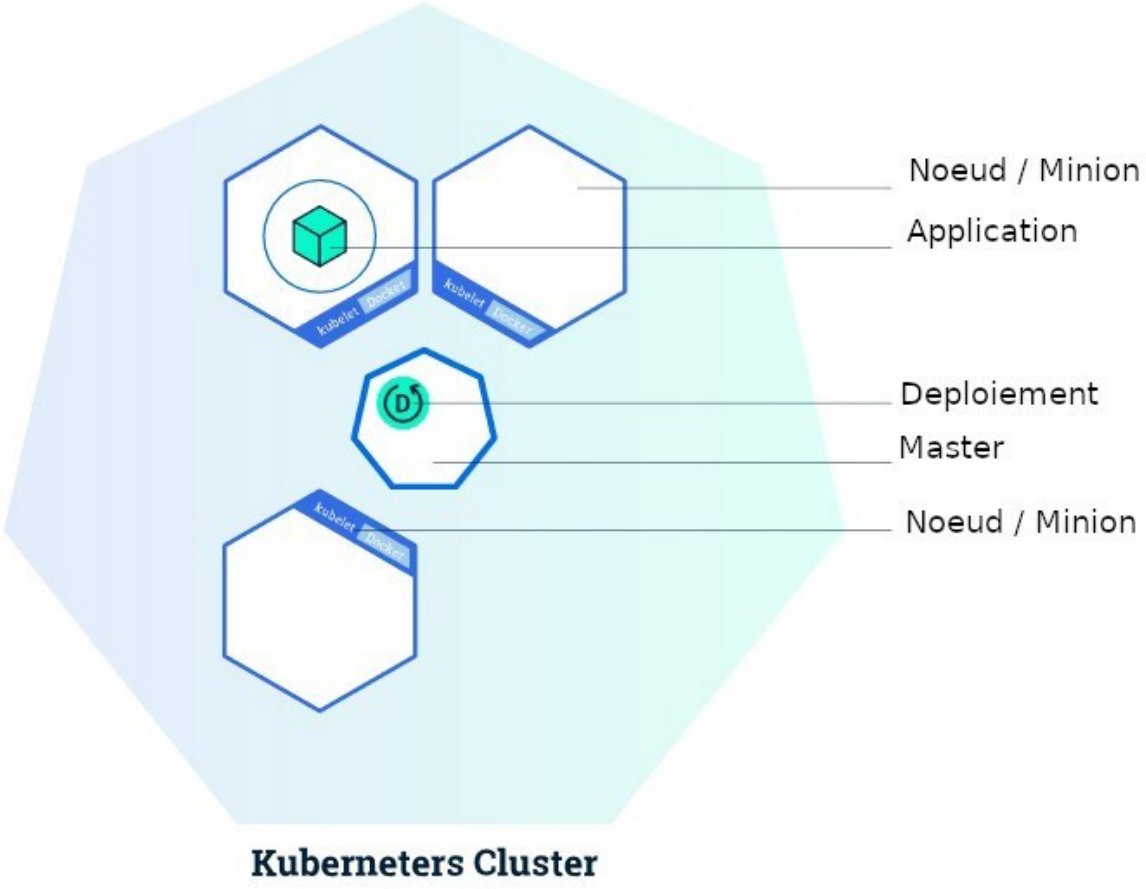
Travaux pratiques

- Un premier déploiement

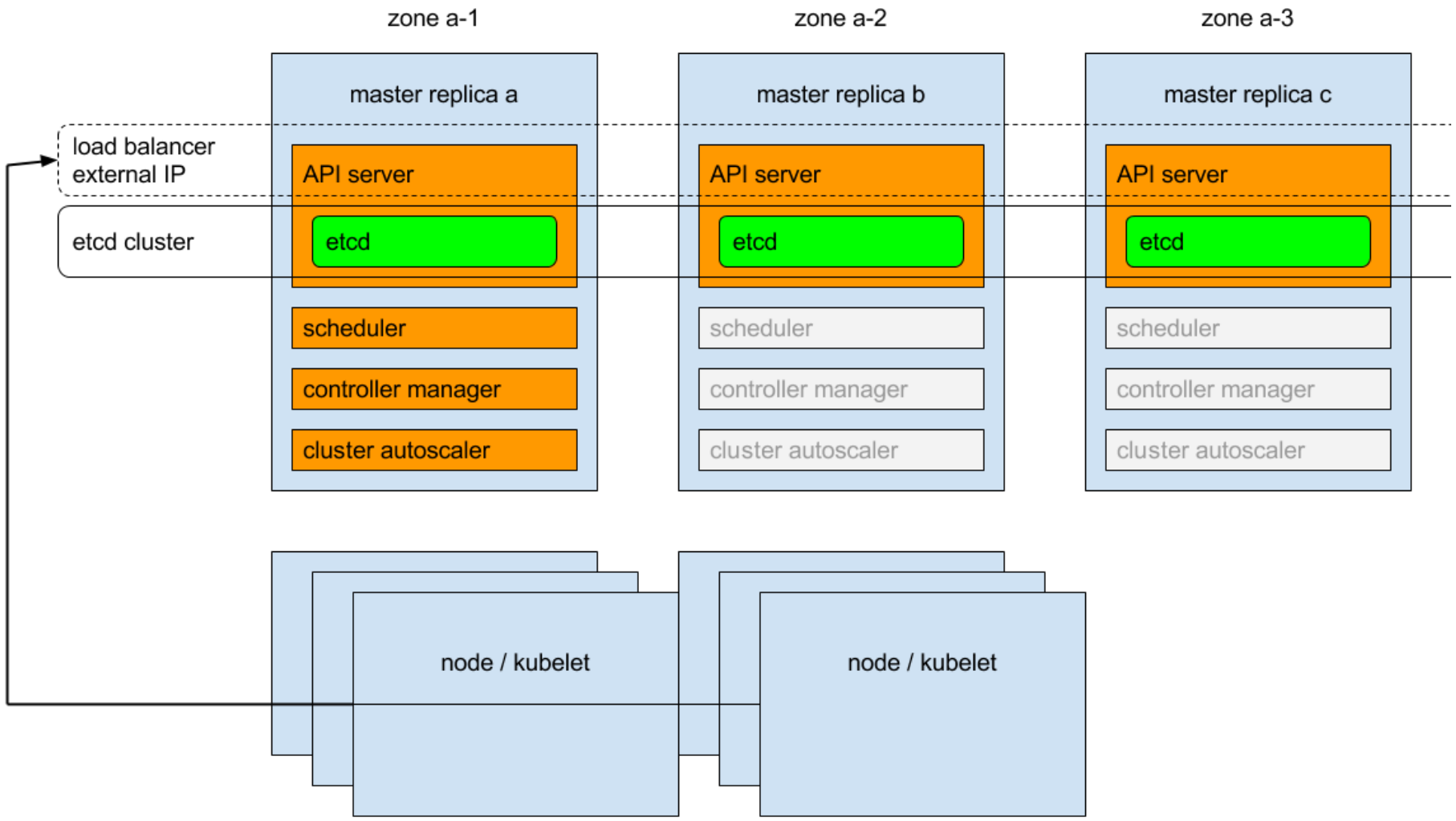
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
    name: nginx-deployment
    labels:
      app: nginx
spec:
    replicas: 3
    selector:
        matchLabels:
            app: nginx
    template:
        metadata:
                labels:
                  app: nginx
            spec:
                containers:
                  - name: nginx
                    image: nginx:1.14.2
                    ports:
                    - containerPort: 81
```

- Ecrire un autre ‘fichier manifest’ pour déployer une application de votre choix

Architecture Kubernetes



Architecture Kubernetes



Architecture Kubernetes

Le Master :

- Le serveur maître sert de plan de contrôle primaire pour les clusters Kubernetes.
- Il sert de point de contact principal pour les administrateurs et les utilisateurs, et fournit également de nombreux systèmes à l'échelle du cluster pour les nœuds ‘worker’.
- Il détermine les meilleurs moyens de planifier les conteneurs de ‘worker’, authentifier les clients et les nœuds, ajuster le réseau à l'échelle du cluster et gérer les responsabilités de mise à l'échelle et de vérification de la bonne santé de l’ensemble.
- Ces composants peuvent être installés sur une seule machine ou répartis sur plusieurs serveurs.

Architecture Kubernetes

Le Master : **etcd**

- Le projet **etcd** est une solution de stockage clé/valeur légère et distribuée qui peut être configurée pour s'étendre sur plusieurs nœuds.
- Kubernetes utilise **etcd** pour stocker les données de configuration accessibles par chacun des nœuds du cluster.
- Il aide également à maintenir l'état du cluster grâce à des fonctions telles que l'élection du chef et le verrouillage distribué.
- Il fournit une API HTTP/JSON.
- **etcd** peut être configuré sur un seul serveur maître ou, dans les scénarios de production, réparti sur plusieurs machines

Architecture Kubernetes

Le Master : **kube-apiserver**

- Service maître le plus important. C'est un serveur **API**.
- C'est le principal point de gestion de l'ensemble du cluster car il permet à un utilisateur de configurer les charges de travail et les unités organisationnelles de Kubernetes.
- Il garantit que le stockage **etcd** et les services actifs sont synchrones.
- Il sert de pont entre les divers composants pour maintenir la bonne santé du cluster.
- Le serveur **API** implémente une interface **RESTful**, ce qui signifie que de nombreux outils et bibliothèques différents peuvent facilement communiquer avec lui. Un client appelé **kubectl** est disponible comme moyen par défaut pour interagir avec le cluster Kubernetes depuis un ordinateur local.

Architecture Kubernetes

Le Master : **kube-controller-manager**

- Il gère les différents contrôleurs qui régulent l'état du cluster, gèrent les cycles de vie des charges de travail et exécutent les tâches de routine.
- Lorsqu'un changement est détecté, le contrôleur lit les nouvelles informations et met en œuvre la procédure qui permet d'atteindre l'état souhaité. Cela peut impliquer la mise à l'échelle d'une application vers le haut ou vers le bas, l'ajustement des ‘endpoints’, etc.

Architecture Kubernetes

Le Master : **kube-scheduler**

- Le processus qui dispatche les charges de travail à des nœuds spécifiques du cluster est **l'ordonnanceur**. Ce service
  - lit les exigences opérationnelles d'une charge de travail,
  - analyse l'environnement d'infrastructure actuel et
  - place le travail sur un ou plusieurs nœuds acceptables.
- Le sheduler est responsable du suivi de la capacité disponible sur chaque hôte pour s'assurer que les charges de travail ne dépassent pas les ressources disponibles. Le planificateur doit connaître la capacité totale ainsi que les ressources déjà allouées aux charges de travail existantes sur chaque serveur.

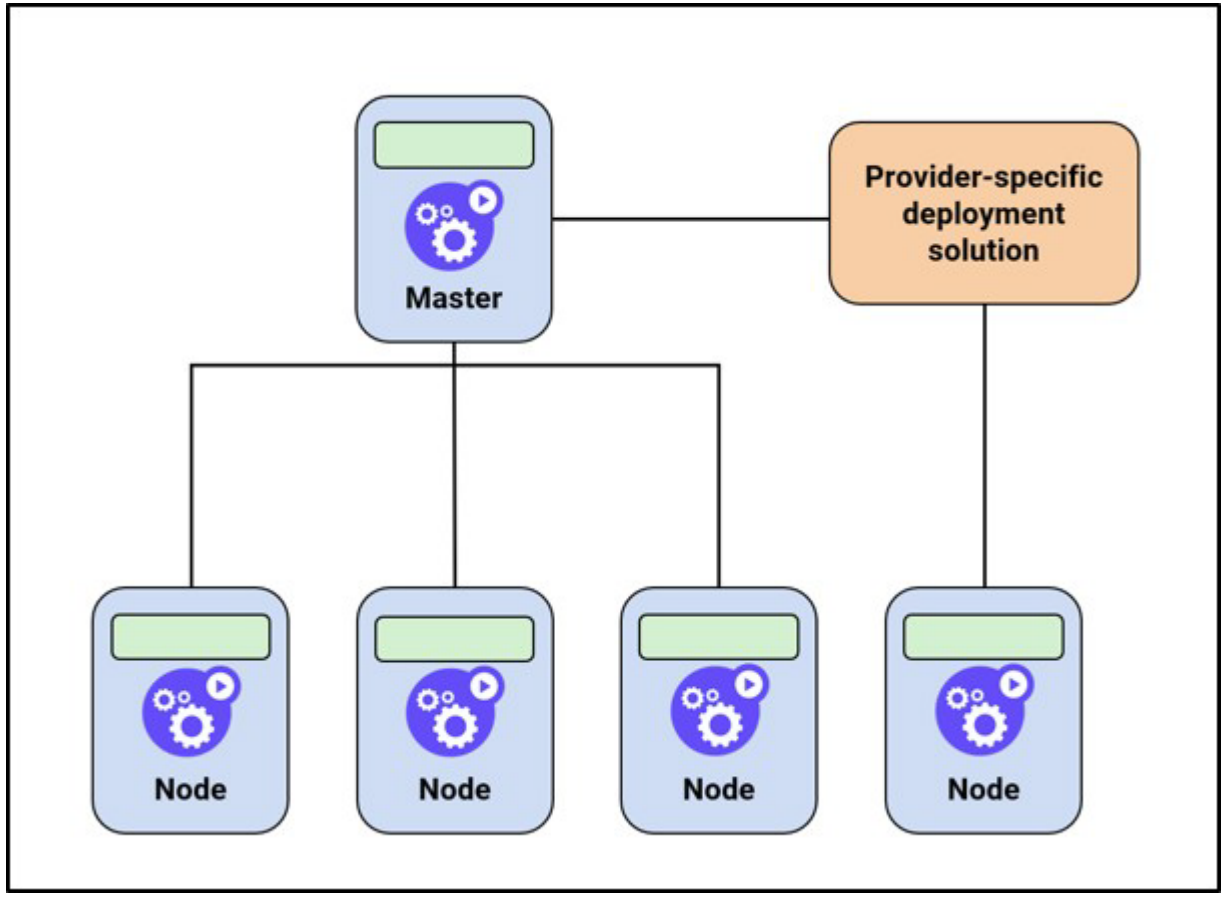
Architecture Kubernetes

Le Master : **cloud-controller-manager**

- Kubernetes peut être déployé dans de nombreux environnements.
- Les gestionnaires de contrôleurs de cloud agissent comme la colle qui permet à Kubernetes d'interagir avec des fournisseurs ayant des capacités, des fonctionnalités et des API différentes.

Architecture Kubernetes

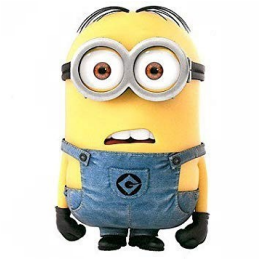
Architecture d'un minion :



Architecture Kubernetes

Architecture d'un minion : **Runtime**

- La base de chaque nœud est un runtime de conteneur. Généralement, cette exigence est satisfaite en installant et en exécutant Docker, mais des alternatives comme rkt, runc, podman ou cri-o sont également possibles.
- Le runtime est responsable du démarrage et de la gestion des conteneurs, applications encapsulées dans un environnement d'exploitation léger relativement isolé.



Architecture Kubernetes

Architecture d'un minion : **kubelet**

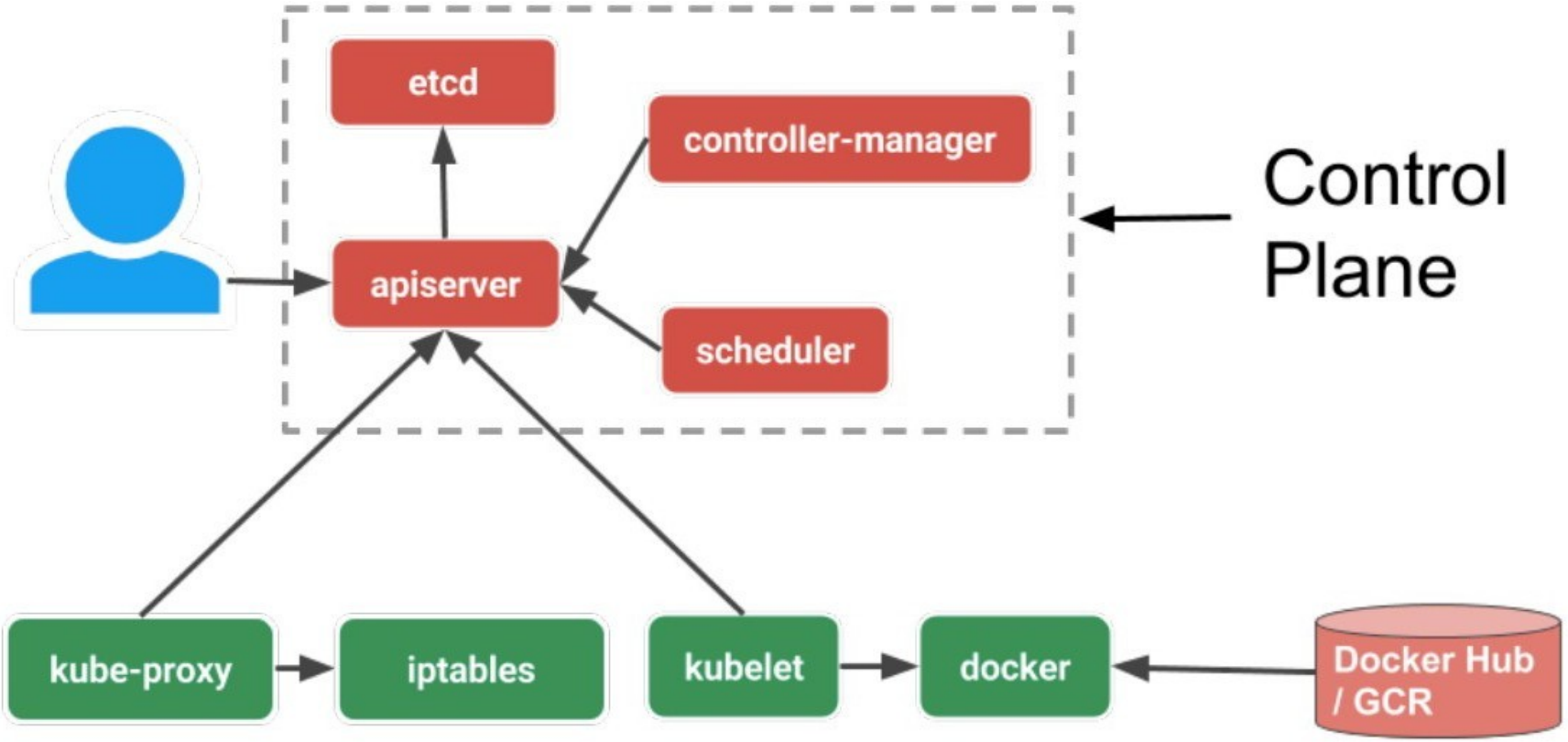
- Point de contact principal pour chaque nœud avec le cluster.
- Ce service est chargé de relayer les informations vers et depuis les services gestionnaires, ainsi que d'interagir avec **etcd** pour lire les détails de configuration ou écrire de nouvelles valeurs.
- **kubelet** communique avec les composants maîtres pour s'authentifier au cluster et recevoir les commandes et le travail à effectuer.
- Le processus **kubelet** assume la responsabilité de maintenir l'état du travail sur le serveur de nœuds. Il contrôle le temps d'exécution du conteneur pour lancer ou détruire les conteneurs selon les besoins.

Architecture Kubernetes

Architecture d'un minion : **kube-proxy**

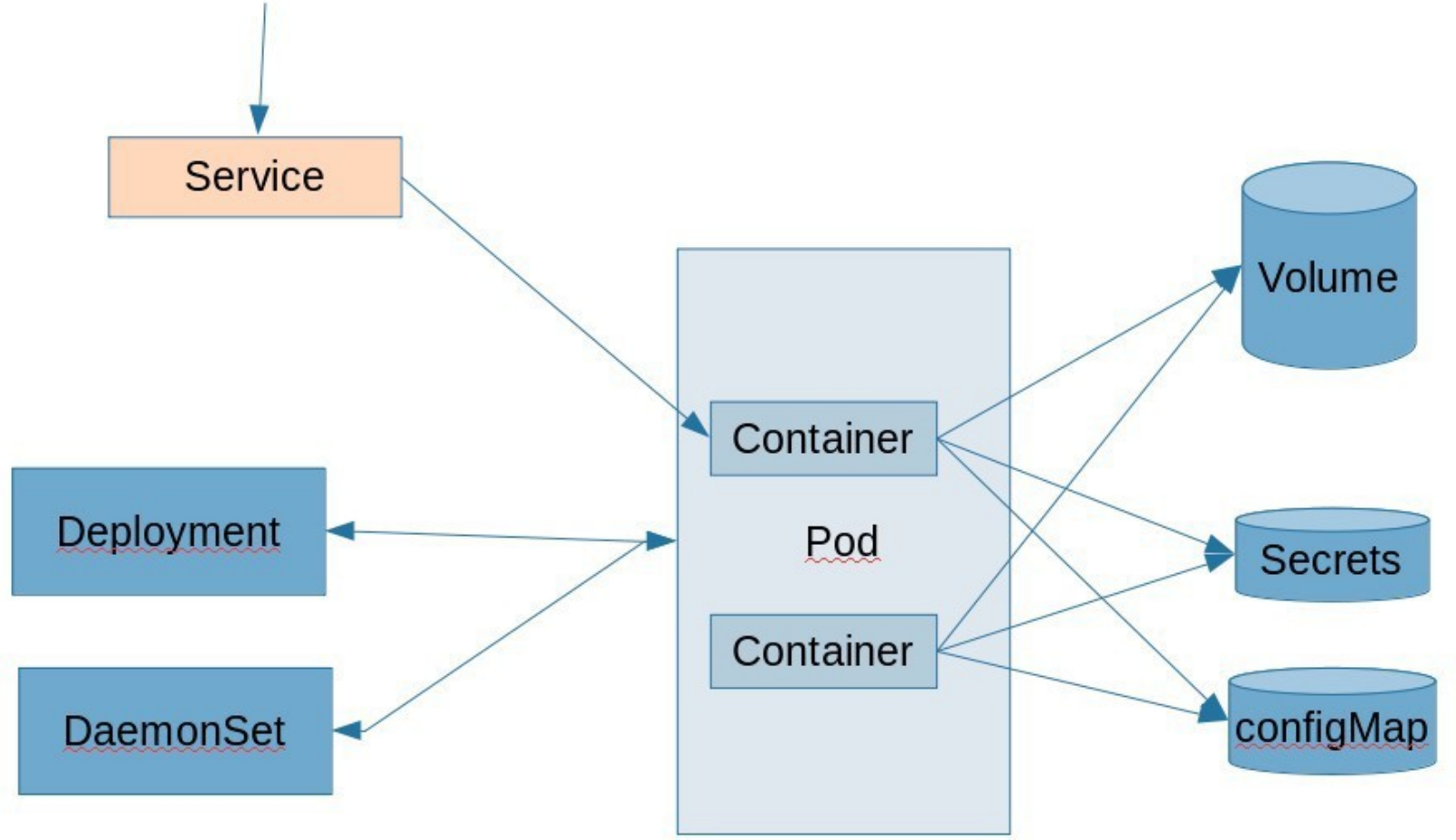
- Pour gérer les sous-réseaux d'hôtes individuels et rendre les services disponibles aux autres composants, un petit service proxy appelé **kube-proxy** est exécuté sur chaque minion. Ce processus achemine les demandes aux conteneurs appropriés, peut effectuer un équilibrage de charge primitif et est généralement chargé de s'assurer que l'environnement réseau est prévisible et accessible, mais isolé le cas échéant.

Architecture Kubernetes
Résumé



Objets de base

Objets de base



Objets de base

Pod

- Un **pod** est l'unité la plus basique avec lequel Kubernetes fonctionne. Les conteneurs eux-mêmes ne sont pas assignés à des hôtes. Au lieu de cela, un ou plusieurs conteneurs étroitement couplés sont encapsulés dans un objet appelé **pod**.
- Un **pod** représente généralement un ou plusieurs conteneurs qui doivent être contrôlés comme une seule application.
- Les pods se composent de conteneurs qui fonctionnent en étroite collaboration, ont un cycle de vie commun et doivent toujours être placés sur le même nœud.
- Ils sont gérés comme une unité indivisible et partagent leur environnement, leurs volumes et leur espace IP.

Objets de base

Pod

```
apiVersion: v1
kind: Pod
metadata:
    name: rss-site
labels:
    app: rss
spec:
    containers:
    - name: front-end
      image: nginx
      ports:
        - containerPort: 80
    - name: rss-reader
      image: nickchase/rss-php-nginx:v1
      ports:
        - containerPort: 88
```

Objets de base

Les PODs sont mortels, leur IP change, ...

Les PODs peuvent être ‘scalé’ (on augmente ou diminue leur nombre en fonction de la charge)

Il faut dès lors :

- Informer les autres conteneurs
- Répartir la charge de façon adéquate

Objets de base

Deployment

Un Deployment est un des objets permettant de lancer des Pods. Dans les bonnes pratiques de Kubernetes il est encouragé d'utiliser des Deployments.

Lors de la création d'un Deployment le contrôleur associé va créer un ReplicaSet à partir de votre configuration. Le contrôleur associé au ReplicaSet va créer une série de Pods à partir de la configuration du ReplicaSet.

Les avantages à employer un Deployment au lieu de créer directement un ReplicaSet sont :

- **Historisation de l'objet** : chaque changement dans l'objet (via un "apply" ou un "edit") va créer une sauvegarde de la version précédente.
- **Gestion du rollout et du rollback** : en lien avec le point précédent, vous pourrez revenir en arrière sur une configuration.

Objets de base

Deployment

Le Deployment est un objet stable depuis la version 1.9 de Kubernetes, disponible sur l'api "apps/v1".

NB :

Il ne faut pas gérer les ReplicaSets ou les Pods appartenant à un déploiement.

Objets de base

Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: influx-app
name: influxdb
spec:
  selector:
    matchLabels:
      app: influx-app
  template:
    metadata:
    labels:
        app: influx-app
      name: influxdb
    spec:
      containers:
```

Définition Pod

Objets de base

Service

Un pod doit communiquer avec l'extérieur.
Un service permet de 'publier' (exposer) une application.

• **Port** : Port est le numéro de port qui rend un service visible aux autres services fonctionnant dans le même cluster K8s. En d'autres termes, si un service veut invoquer un autre service fonctionnant dans le même cluster Kubernetes, il pourra le faire en utilisant le port spécifié comme "port" dans le fichier de spécifications du service.

  • **targetPort** : Le port cible est le port du POD sur lequel le service est exécuté.

• **Nodeport** : Nodeport est le port sur lequel le service peut être atteint par des utilisateurs externes en utilisant kube-proxy. Normalement laissé à la discrétion de Kubernetes (entre 30000 et 32000).

Objets de base

Service

```
apiVersion: v1
kind: Service
metadata:
  name: example-service
  labels:
    app: example-service
spec:
  ports:
  - port: 80
    protocol : <TCP|UDP>
    name :
    targetPort: 4000
  type: <NodePort|clusterIP|loadBalancer|externalName>
  selector:
    app: example-pod
```

Objets de base

hostPath

Un volume hostPath monte un fichier ou un répertoire du système de fichiers du nœud hôte dans votre Pod.

Certaines utilisations d'un hostPath :

- exécuter un conteneur qui a besoin d'accéder au fonctionnement interne de Docker
  Ex. : utiliser un chemin hôte de /var/lib/docker ou /var/run/docker.sock
- exécuter cAdvisor dans un conteneur, pour accéder à /sys
- permettre à un Pod de spécifier si un hostPath donné doit exister avant l'exécution du Pod, s'il doit être créé, et qu'il doit exister.

Objets de base

hostPath

En plus du chemin (obligatoire), l'utilisateur peut spécifier un type pour un volume hostPath.

Les valeurs prises en charge pour le type de zone sont :

Valeur	Description
vide	pour la rétrocompatibilité, ce qui signifie qu'aucun contrôle ne sera effectué avant le montage du volume hostPath.
DirectoryOrCreate	Si rien n'existe au niveau du chemin donné, un répertoire vide y sera créé selon les besoins avec l'autorisation fixée à 0755, ayant le même groupe et le même propriétaire avec Kubelet.
Directory	Un répertoire doit exister sur le chemin donné
FileOrCreate	Si rien n'existe sur le chemin donné, un fichier vide y sera créé au besoin avec la permission 0644, ayant le même groupe et le même propriétaire que Kubelet.
File	Un fichier doit exister au chemin donné
Socket	Un socket UNIX doit exister sur le chemin donné
CharDevice	Un périphérique de caractères doit exister sur le chemin donné
BlockDevice	Un périphérique de bloc doit exister sur le chemin donné

Objets de base

hostPath

```
apiVersion: v1
kind: Pod
metadata:
    name: test-pd
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-pd
      name: test-volume

  volumes:
  - name: test-volume
    hostPath:
      path: /data
      type: Directory
```

Objets de base

Les plugins de volume Kubernetes

Kubernetes propose beaucoup de plugins de volume :

- Stockage distant :
  - Disque persistant GCE
  - AWS
  - Azure (FileSystem et disque de données)
  - Dell EMC ScaleIO
  - iSCSI
  - Flocker
  - NFS
  - vSphere
  - GlusterFS
  - Ceph File et RBD
  - Cinder
  - Quobyte Volume
  - FibreChannel
  - VMware Photon PD
  - ...

Objets de base

PersistentVolume / persistentVolumeClaim / storageClass

Un PersistentVolume (PV) est un élément de stockage dans le cluster qui a été provisionné manuellement par un administrateur, ou dynamiquement provisionné par Kubernetes à l'aide d'une StorageClass.

Une PersistentVolumeClaim (PVC) est une demande de stockage par un utilisateur qui peut être satisfaite par un PersistentVolume.

PersistentVolumes et PersistentVolumeClaims sont indépendants du cycle de vie des Pods et préservent les données en redémarrant, reprogrammant et même en supprimant les Pods.

Objets de base

PersistentVolume / persistentVolumeClaim / storageClass

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  namespace: kube-system
  name: speed
  annotations:
    storageclass.beta.kubernetes.io/is-default-class: "true"
  labels:
    addonmanager.kubernetes.io/mode: EnsureExists
provisioner: k8s.io/minikube-hostpath
```

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nextcloud-db-claim
  labels:
    app: nextcloud
spec:
  storageClassName: speed
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  resources:
    requests:
      storage: 1Gi
```

Objets de base

PersistentVolume / persistentVolumeClaim / storageClass

```
apiVersion: apps/v1
kind: Deployment
…
spec:
…
  template:
…
    spec:
      containers:
…
        volumeMounts:
        - name: nextcloud-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: nextcloud-storage
        persistentVolumeClaim:
          claimName: nextcloud-db-claim
```

Objets de base

Quel futur pour le stockage Kubernetes ?

Le stockage Kubernetes tend vers :

- Container Storage Interface (CSI)
  - Standardisation de plugins de volume block et fichier “Out-of-Tree”
- Stockage local
  - Rendre le stockage local d’un noeud disponible en tant que volume persistant
- Isolation des capacités
  - Mettre en place des limites afin qu’un unique pod ne puisse consommer toutes les ressources de stockage d’un noeud via overlay FS, des logs, etc.

Objets de base

Secret et ConfigMap

Il est souvent nécessaire de référencer des données "spéciales", tels que des clés d'API, des jetons et d'autres secrets. Le comportement des applications peut être personnalisé à l'aide de paramètres de configuration, par exemple, un fichier PHP.ini, ou des variables d'environnement.

Pour éviter de coder ces références en dur dans votre logique applicative. Kubernetes gère 2 types d’objets : **secrets** et **configMap**.

Objets de base

Secret et ConfigMap

```
apiVersion: v1
kind: Secret
metadata:
  name: next-conf
type: Opaque
data:
  DB: ZGIw
  ADMIN: YWRtaW4=
```

Couple Clé / Valeur (echo -n ‘secret’ | base64)

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  A_SPECIFIC_VAR: FOOBAR
```

Objets de base

Secret et ConfigMap

```
apiVersion: apps/v1
kind: Deployment
...
spec:
...
  template:
...
    spec:
      containers:
        - name: mypod
          image: easylinux/kubernetes:nginx
          volumeMounts:
            - mountPath: "/etc/mypassword"
              name: credential
      volumes:
        - name: credential
          secret:
            secretName: next-conf
```

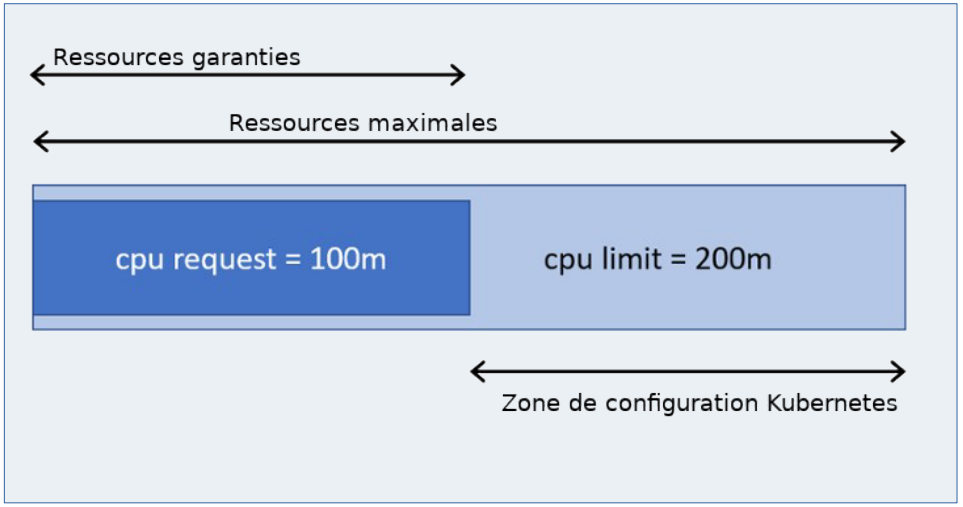
```
apiVersion: apps/v1
kind: Deployment
...
  template:
 ...
    spec:
      containers:
        - name: mypod
          image: easylinux/kubernetes:nginx
          env:
            - name: NEXT_DB
              valueFrom:
                secretKeyRef:
                name: next-conf
                key: NEXT_DB
```

Objets de base

Limiter les ressources

Limitation par conteneur

En production, il est nécessaire de contrôler la consommation de ressource d'une application.



* 200m signifie 200 milliCPU, soit 0.2 CPU

Objets de base

Limiter les ressources

```
apiVersion: apps/v1
kind: Deployment
...
  containers:
  - name: frugal
    resources:
      limits:
        cpu: 200m
        memory: 500Mi
      requests:
        cpu: 100m
        memory: 500Mi
```

* 200m signifie 200 milliCPU, soit 0.2 CPU

Objets de base

Limiter les ressources

Limitation par conteneur et QosClass

- Si les valeurs Requests et Limits (cpu et memory) sont égales, alors, le pod sera **Garanteed**.
- Si un pod à une demande mémoire (requests et limits) mais pas de CPU, alors il est **Burstable**.
- Si aucune demande de ressource n'est faite, alors il est **BestEffort**.

Objets de base

DaemonSet

Un DaemonSet garantit que tous (ou certains) nœuds exécutent une copie et une seule d'un Pod. Au fur et à mesure que des nœuds sont ajoutés au cluster, des Pods y sont instanciés. Au fur et à mesure que les nœuds sont retirés du cluster, les Pods sont détruits.

Supprimer un DaemonSet nettoiera les Pods qu'il a créés.

Objets de base

DaemonSet

Quelques utilisations typiques d'un DaemonSet sont :

- ➔ exécuter un démon de stockage en cluster, tels que glusterd ou ceph, sur chaque noeud.
- ➔ L'exécution d'un démon de collecte de logs sur chaque noeud, tels que fluentd ou logstash.
- ➔ Exécuter un démon de surveillance de nœuds, tels que Prometheus Node Exporter, collectd, Dynatrace OneAgent, AppDynamics Agent, Datadog agent, New Relic agent, Ganglia gmond ou Instana agent.

Objets de base

DaemonSet

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  labels:
    app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - name: fluentd-elasticsearch
```

Objets de base

DaemonSet

Toleration Key	Effect	Version	Description
node.kubernetes.io/not-ready	NoExecute	1.13+	
node.kubernetes.io/unreachable	NoExecute	1.13+	
node.kubernetes.io/disk-pressure	NoSchedule	1.8+	
node.kubernetes.io/memory-pressure	NoSchedule	1.8+	
node.kubernetes.io/unschedulable	NoSchedule	1.12+	
node.kubernetes.io/network-unavailable	NoSchedule	1.12+	

Objets de base

Service **Stateful** vs. Service **Stateless**

- Lors de la réalisation d'un service pour une architecture, il arrive de devoir introduire, dans certaines situations, les notions de contexte ou de session. Nous allons voir quels sont les impacts de tels mécanismes et qu'il suffit d'appliquer la célèbre méthode KISS (Keep It Simple, Stupid) pour s'en sortir.

Que signifient les termes *contexte* et *session* ?

- Le contexte d'un service est l'environnement fonctionnel (par exemple un cas d'utilisation) ou technique (par exemple une application web J2EE) dans lequel celui-ci est invoqué.
- La session permet de conserver des informations au travers de plusieurs invocations d'un même service jusqu'à la libération de cette session.

Le contexte et la session nuisent à la compréhension et à la maintenabilité.

Objets de base

Où gérer le contexte ou la notion de session ?

- Qui mieux que l'appelant sait dans quel contexte il se trouve ? L'appelé ne devrait jamais être influencé par le contexte ou les données de la session.
- Introduire des services contextuels (ou **stateful**) dans une architecture ne doit pas se faire à la légère et peut grièvement impacter la réutilisabilité, la maintenance et les performances des services.

En résumé, il faut donc :

- définir des services simples qui répondent à un seul besoin,
- utiliser le passage d'informations explicite par paramètres

Il faut garder en tête qu'une bonne définition des services est importante. En définissant des services réutilisables, nous évitons de passer notre temps à faire, défaire et refaire les mêmes services. On peut ainsi se concentrer sur les services qui répondent aux nouveaux besoins métiers et apportent une réelle valeur ajoutée.

Travaux pratiques

Utilisation de deployment.

Création de

- Pod
- Deployment
- Service
- HostPath
- PersistentVolume
- ConfigMap
- Secret

Déploiement d'un cluster Kubernetes

Déploiement : Clustering avec MicroK8s

- Chaque nœud d'un cluster MicroK8s nécessite son propre environnement pour fonctionner, qu'il s'agisse d'une machine virtuelle ou d'un conteneur distinct sur une seule machine ou d'une machine différente sur le même réseau.

Déploiement d'un cluster Kubernetes

Préparation des nœuds.

- Chaque VM Multipass va tourner sur un Ubuntu 18.04 LTS ou Ubuntu 20.04 LTS
- Installer Docker (optionnel) car MicroK8s vient avec Containerd (qui est un conteneur runtime)
- Vérifier la visibilité réseau
- Installer MicroK8s

Déploiement d'un cluster Kubernetes

Déploiement : Ajouter un nœud worker au master-node

Lancer ‘microk8s add-node’ sur le master

```
Join node with:
microk8s join ip-172-31-20-243:25000/DDOkUupkmaBezNnMheTBqFYHLWINGDbf

If the node you are adding is not reachable through the default
interface you can use one of the following:

microk8s join 10.1.84.0:25000/DDOkUupkmaBezNnMheTBqFYHLWINGDbf
microk8s join 10.22.254.77:25000/DDOkUupkmaBezNnMheTBqFYHLWINGDbf
```

Déploiement d'un cluster Kubernetes

Déploiement : Ajouter un nœud worker au master-node

Lancer microk8s join ip …

sur le worker-node

```
microk8s join ip-172-31-20-243:25000/DDOkUupkmaBezNnMheTBqFYHLWINGDbf
```

Déploiement d'un cluster Kubernetes

Mise en place du Dashboard et du réseau :

- installer une couche réseau Weave / Flannel, …
- Installer le Dashboard.
- Activation de différents plugins (services).

Déploiement d'un cluster Kubernetes

Travaux pratiques

- Mise en place d'un cluster de 3 noeuds

Exploiter Kubernetes

Namespace

Pour gérer efficacement Kubernetes, il est conseillé de créer des espaces par usage, puis de donner des accès et de contraintes d'utilisation pour chacun de ces espaces.

Exemple :

```
apiVersion: v1
kind: Namespace
metadata:
  name: <nom namespace>
```

Exploiter Kubernetes

Limiter les ressources – Quota

Un quota de ressources, défini par un objet ResourceQuota, fournit des contraintes qui limitent la consommation globale de ressources par **namespace**. Il peut limiter la quantité d'objets qui peuvent être créés dans un espace de noms par type, ainsi que la quantité totale de ressources de calcul qui peuvent être consommées par les ressources dans ce projet.

Exploiter Kubernetes

Limiter les ressources – Quota

Utilisation :

- ➔ Des équipes différentes travaillent dans des espaces différents.
- ➔ L'administrateur crée un quota de ressources pour chaque espace de nommage.
- ➔ Les utilisateurs créent des ressources (pods, services, etc) dans l'espace de nommage et le système de quota suit l'utilisation pour s'assurer qu'elle ne dépasse pas les limites de ressources dures définies dans un quota de ressources.

Exploiter Kubernetes

Limiter les ressources – Quota

Utilisation :

- Si la création ou la mise à jour d'une ressource viole une contrainte de quota, la demande échoue avec le code d'état HTTP 403 FORBIDDEN avec un message expliquant la contrainte qui a été violée.
- Si le quota est activé dans un espace de noms pour calculer les ressources telles que le processeur et la mémoire, les utilisateurs doivent spécifier des requêtes ou des limites pour ces valeurs ; sinon, le système de quota peut rejeter la création de pods.
  Astuce : utilisez le contrôleur d'admission LimitRange pour forcer les valeurs par défaut pour les pods qui ne nécessitent aucun calcul de ressources. Voir la marche à suivre pour un exemple de la façon d'éviter ce problème.

Exploiter Kubernetes

Limiter les ressources – Quota

Voici des exemples de stratégies qui pourraient être définies à l'aide d'espaces de noms et de quotas :

- Dans un cluster d'une capacité de 32 Go RAM et 16 cœurs,
  - l'équipe A utilise 20 Go et 10 cœurs,
  - l'équipe B utilise 10 Go et 4 cœurs,
  - on garde 2 Go et 2 cœurs en réserve pour allocation future.
- Limiter l'espace de nommage "testing" à l'utilisation d'un noyau et de 1 Go de RAM.
  Ne pas restreindre l'espace de nommage "production".

Dans le cas où la capacité totale du cluster est inférieure à la somme des quotas des espaces de noms, il peut y avoir conflit pour les ressources. Les demandes sont traitées selon le principe du premier arrivé, premier servi.

Ni les controverses ni les changements de quotas n'affecteront les ressources déjà créées.

Exploiter Kubernetes

Limiter les ressources – Quota

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: mem-cpu-demo
  namespace: dev
spec:
  hard:
    requests.cpu: "1"
    limits.cpu: "2"
    requests.memory: 1Gi
    limits.memory: 2Gi
```

Exploiter Kubernetes

nodeSelector

Première étape : Fixer un label au nœud

Exécuter $ kubectl get nodes pour obtenir la liste des noeuds de votre cluster.

Choisir celui auquel vous voulez ajouter une étiquette, lancer

$ kubectl label node <node-name> <label-key>=<label-value>

Exemple : $ kubectl label nodes kubernetes disktype=ssd

Vérifier que cela a fonctionné avec :

$ kubectl get nodes --show-labels

Exploiter Kubernetes

Seconde étape : Ajouter un sélecteur à votre définition

```
apiVersion: v1
kind: Pod
metadata:
    name: nginx
    labels:
        env: test
spec:
    containers:
      - name: nginx
        image: nginx
        imagePullPolicy:  IfNotPresent
    nodeSelector:
        disktype: ssd
```

Lancer $ kubectl create -f pod-nginx.yaml, le pod sera affecté au nœud avec le label.

Pour vérifier :
$ kubectl get pods -o wide

Exploiter Kubernetes

Affinité et anti-affinité

nodeSelector fournit un moyen très simple de contraindre les pods à des nœuds avec des étiquettes particulières.

La fonction affinité/anti-affinité, **actuellement en version bêta**, étend considérablement les types de contraintes que vous pouvez exprimer. Les principales améliorations sont les suivantes :

- le langage est plus expressif (et pas seulement "de correspondance exacte").
- Vous pouvez indiquer que la règle est "soft"/"preference" plutôt qu'une exigence dure, donc si le planificateur ne peut pas la satisfaire, le pod sera toujours programmé.
- Vous pouvez vous opposer à des étiquettes sur d'autres pods fonctionnant sur le nœud (ou un autre domaine topologique), plutôt qu'à des étiquettes sur le nœud lui-même, ce qui permet des règles sur les pods qui peuvent et ne peuvent pas être co-localisés.

Exploiter Kubernetes

Health check

Afin de déterminer si un conteneur dans un pod est sain et prêt à servir, Kubernetes met en place une série de mécanismes de vérification.
Les HealthCheck, ou probes comme on les appelle dans Kubernetes, sont effectués par kubelet pour déterminer quand redémarrer un conteneur (pour LivenessProbe) et par les services pour déterminer si un pod peut recevoir du trafic ou non (pour ReadinessProbe).

NB :

Il est de la responsabilité du développeur de l'application d'exposer une URL que le kubelet peut utiliser pour déterminer si le conteneur est sain (et potentiellement prêt).

Exploiter Kubernetes

Créer un pod qui expose un endpoint/health, en répondant avec un code d'état HTTP 200 :

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

Le code ci-dessus signifie que Kubernetes testera /health toutes les 5 secondes après avoir attendu 2 secondes pour le contrôle initial.

Exploiter Kubernetes

Pour visualiser l'état de santé du pod :

```
$ kubectl describe pod hc
Name:                   hc
Namespace:              default
Security Policy:        anyuid
Node:                   192.168.99.100/192.168.99.100
Start Time:             Tue, 25 Apr 2017 16:21:11 +0100
Labels:                 <none>
Status:                 Running
...
Events:
    FirstSeen           LastSeen            Count       From
...
```

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

Travaux pratiques

Déploiement d'une application

- Mise en place d’affinité sur un nœud
- Contrainte sur une machine
- Mise en œuvre de healthcheck

Création de conteneur personnalisé

Qu'est-ce qu'un Dockerfile ?

Les 'Dockerfile' sont des fichiers qui permettent de construire une image Docker adaptée à nos besoins, étape par étape.

La convention est de nommer le fichier 'Dockerfile' et il doit se trouver à la racine du dossier projet.

La première chose à faire dans un Dockerfile est de spéficier quelle image va faire office de socle.

```
FROM alpine:3.9
```

FROM permet de définir notre image de base.

Création de conteneur personnalisé

Qu'est-ce qu'un Dockerfile ?

Exemple :

```
FROM python:3.5
RUN apt-get update && apt-get install -y libffi-dev
RUN mkdir -p /usr/src/app
RUN groupadd myappgroup \
        && useradd --create-home --home-dir /home/myappuser \
          --uid 4242 -g myappgroup myappuser
RUN chown -R myappuser:myappgroup /usr/src/app
COPY . /usr/src/app
WORKDIR /usr/src/app
RUN pip install -r requirements.txt
ENTRYPOINT docker-entrypoint.sh
EXPOSE 5000
```

Création de conteneur personnalisé

FROM: importe récursivement la définition d'images parentes

LABEL: définit une valeur (author)

RUN: lance une commande via /bin/sh

CMD: définit la commande par défaut du conteneur

EXPOSE: définit les ports (UDP/TCP) écoutés au sein du conteneur

ENV: définit une variable d'environnement

COPY: copie des fichiers/dossiers dans l'image

Création de conteneur personnalisé

ENTRYPOINT:	définit une commande exécutée au lancement du conteneur
VOLUME:	espace de données persistant partagé entre l'hôte et le conteneur
USER:	utilisateur au sein du conteneur
WORKDIR:	répertoire courant dans le conteneur

Référence : https://docs.docker.com/engine/reference/builder/

Création de conteneur personnalisé

Attention à ne pas confondre RUN et CMD :

RUN exécute des commandes et produit des commit et nouvelles images

CMD n'exécute rien au moment du build (ça établit dans l'image la commande à exécuter à l'instanciation du conteneur)

Création de conteneur personnalisé

ENTRYPOINT vs. CMD

Un Dockerfile doit spécifier au moins un des deux :

- **ENTRYPOINT** doit être utilisé quand on utilise un conteneur comme un exécutable.

- **CMD** doit être utilisé pour définir des arguments par défaut pour un ENTRYPOINT ou pour exécuter une commande ad-hoc dans le conteneur.

NB :

CMD sera surchargé en démarrant un conteneur avec des arguments

Création de conteneur personnalisé

ONBUILD

Ajouter à l'image un déclencheur (trigger) qui sera déclenché plus tard : lorsque cette image sera utilisée comme base image (FROM) pour la construction (build) d'une nouvelle image.

Cas d'utilisation : une image réutilisable.

Création de conteneur personnalisé

ONBUILD

Lors de la première construction, on ne peut pas ajouter des sources avec ADD car à ce moment nous n'avons pas encore d'application.

...

ONBUILD ADD . /app/src

**ONBUILD RUN /usr/local/bin/python-build **

--dir /app/src

...

Au build de l'image de base des triggers sont ajoutés, mais les instructions correspondantes ne sont pas exécutées.

Création de conteneur personnalisé

ONBUILD

Quand l'image de base est utilisée (FROM) pour un nouveau build.

Le builder vérifie s'il y a des triggers définis et si c'est le cas ceux-ci sont exécutés.

Les triggers sont nettoyés de l'image résultante : donc les triggers ne sont pas hérités par les "petits-enfants".

Création de conteneur personnalisé

Bonnes pratiques :

- ➔ Un conteneur doit être éphémère
- ➔ Ne pas installer des paquets inutiles
- ➔ Un seul but / application par conteneur
- ➔ Limiter le nombre de couches
- ➔ Trier les lignes multiples (`&& \`)

Création de conteneur personnalisé

Dockerfile : spécificités Windows

WORKDIR C:\\App

VOLUME C:\\Data

Possibilité d'utiliser ‘ à la place de #

ADD web.config C:/App

SHELL [“PowerShell”]

Par défaut : Cmd.exe

NB:

Pas de **FROM SCRATCH**

Création de conteneur personnalisé

Construire une image depuis un Dockerfile :

docker build [-t <repository:tag>] <dossier>

Exemple :

docker build -t easylinux/apache:2.5 .

Création de conteneur personnalisé

Une fois l'image créée, nous pouvons instance un conteneur avec docker run :

docker run -d easylinux/apache:2.5

Création de conteneur personnalisé

Les bonnes pratiques imposent de séparer les services dans des conteneurs distincts. Mais dans certains cas (Nginx/PHP), il peut être lourd d'avoir 2 conteneurs.

Une solution : **supervisor**

Supervisor est un système qui permet de contrôler un ensemble de processus évoluant dans un environnement UNIX. Pour faire simple Supervisor lance des services et les redémarre s'ils échouent.

Création de conteneur personnalisé

Supervisor lit un fichier de configuration situé dans /etc/supervisor.

Chaque fichier finissant en .conf et situé dans le chemin /etc/supervisor/conf.d représente un processus surveillé par Supervisor.

[program:<nom>]

command = <programme>

user = <utilisateur>

directory = <chemin>

autostart = true

autorestart = true

Travaux pratiques

Création et automatisation d'images personnalisées.

- Mise en œuvre d’un registre privé
- Créer un conteneur MariaDB
- Créer un conteneur JBoss

Kubernetes en production

Frontal Ingress.

- Kubernetes Ingress est un ensemble de règles de routage qui régit la manière dont les utilisateurs externes accèdent aux services s'exécutant dans un cluster Kubernetes.
- Dans les déploiements réels de Kubernetes il y a souvent d'autres considérations que le routage pour la gestion d’Ingress.

Kubernetes en production

Ingress dans Kubernetes

Dans Kubernetes, il existe trois principales approches pour exposer votre application.

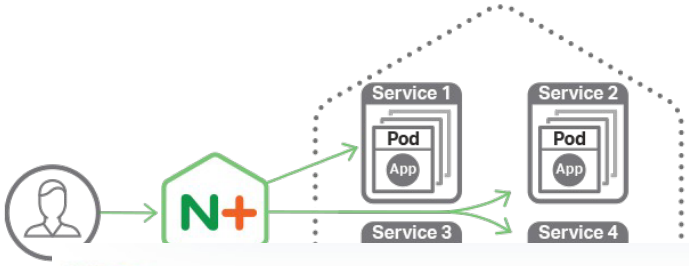
- Utiliser un service Kubernetes de type NodePort qui expose l'application sur un port à travers chacun de vos nœuds.
- Utiliser un service Kubernetes de type LoadBalancer qui crée un équilibreur de charge externe qui pointe vers un service Kubernetes dans votre cluster.
- Utiliser une ressource Ingress.

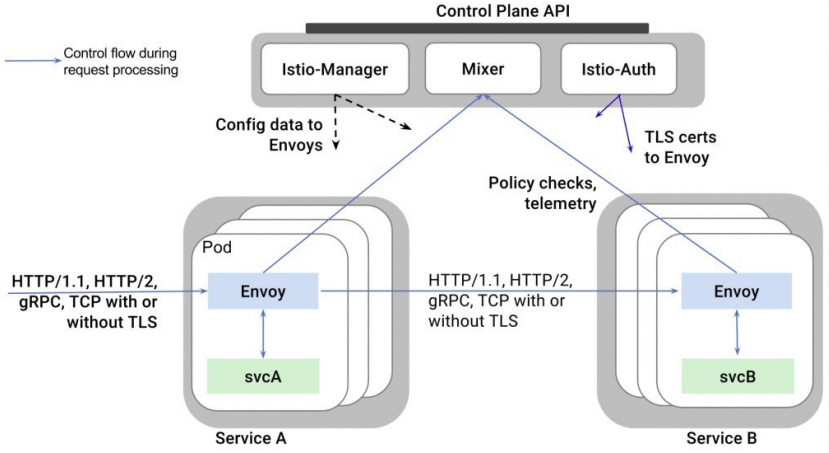
Kubernetes en production

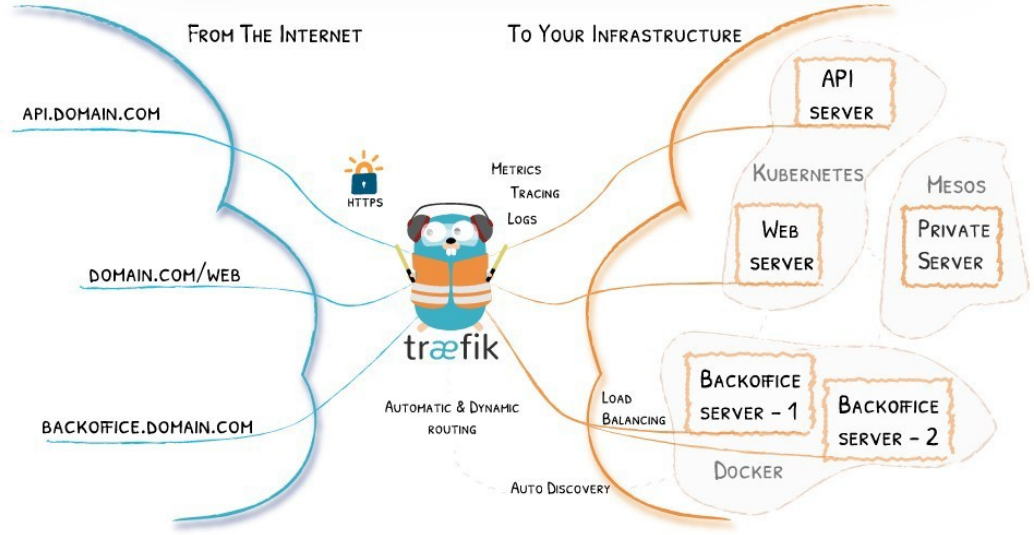
Contrôleur Ingress et ressource Ingress

- Une entrée est un concept de base (en bêta) de Kubernetes, mais elle est toujours implémentée par un **proxy tiers**.
- Le contrôleur Ingress est responsable de lire l'information sur les ressources d'entrée et de traiter ces données en conséquence. Différents contrôleurs Ingress ont étendu la spécification de différentes manières pour prendre en charge des cas d'utilisation supplémentaires.
- Ingress est étroitement intégré à Kubernetes, ce qui signifie que vos workflows existants autour de kubectl s'étendront probablement bien à la gestion des entrées.
- Un contrôleur Ingress n'élimine généralement pas le besoin d'un équilibreur de charge externe - le contrôleur Ingress ajoute simplement une couche supplémentaire de routage et de contrôle derrière l'équilibreur de charge.

Kubernetes en production







Kubernetes en production

Gestion des ressources et autoscaling.

- L'Horizontal Pod Autoscaler met automatiquement à l'échelle le nombre de pods dans un contrôleur de réplication, un déploiement ou un ensemble de répliques en fonction de l'utilisation observée du CPU (ou, avec le support de métriques personnalisées, sur certaines autres métriques fournies par l'application).
  Notez que la mise à l'échelle horizontale ne s'applique pas aux objets qui ne peuvent pas être mis à l'échelle, par exemple les DaemonSets.
- L'Horizontal Pod Autoscaler est implémenté en tant que ressource API Kubernetes et contrôleur. La ressource détermine le comportement du régulateur. Le contrôleur ajuste périodiquement le nombre de répliques d'un contrôleur de réplication ou d'un déploiement afin de faire correspondre l'utilisation moyenne observée du CPU à la cible spécifiée par l'utilisateur.

```
$ kubectl autoscale rs monApp --min=2 --max=5 --cpu-percent=80
```

Kubernetes en production

Service Discovery (env, DNS).

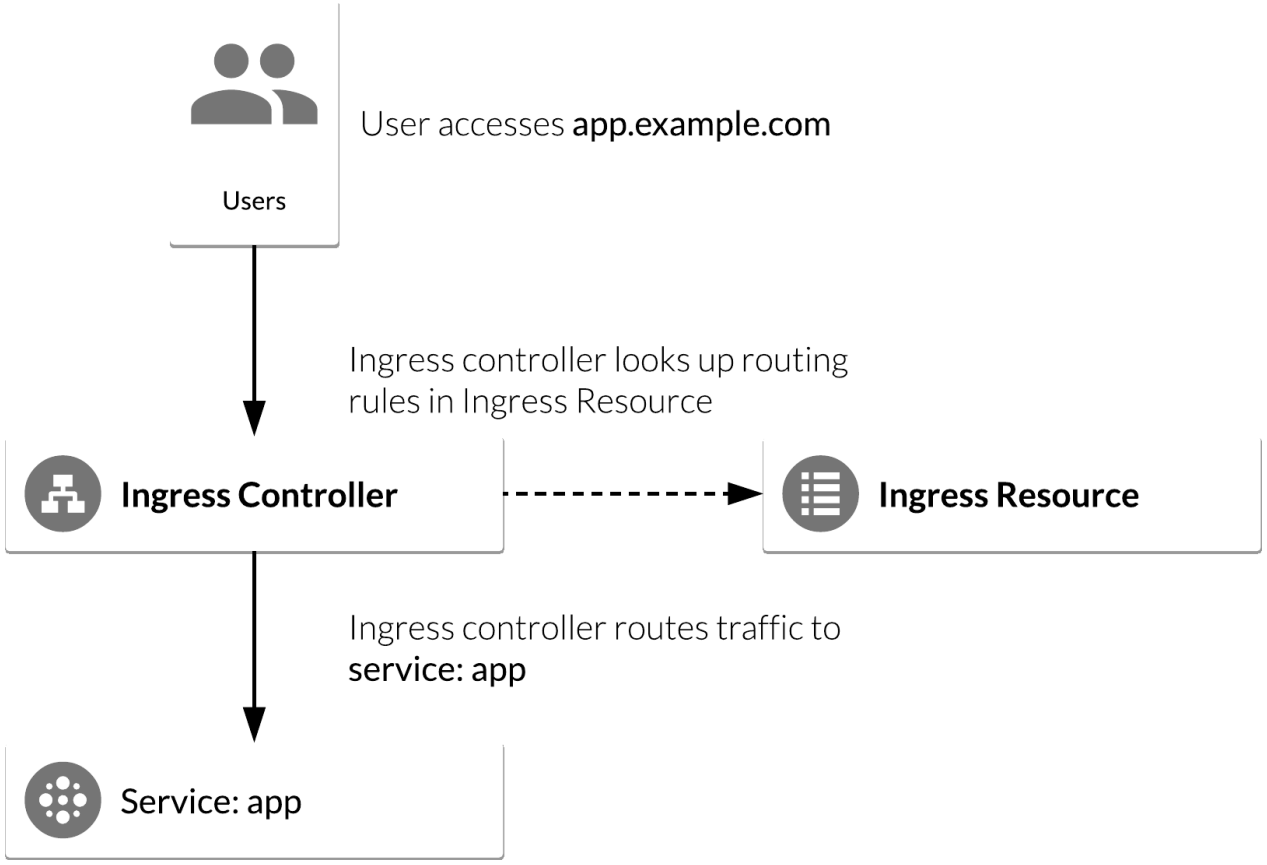
Le service Discovery est un processus qui consiste à déterminer comment se connecter à un service. Bien qu'il existe une option de découverte de service basée sur les variables d'environnement, la découverte de service basée sur DNS est préférable. Notez que le DNS est un add-on cluster, donc assurez-vous que votre distribution Kubernetes en fournit un ou installez-le vous-même.

Travaux pratiques

- Mise en œuvre du frontal Ingress
- Autoscaling du frontal Web

Travaux pratiques

Mise en œuvre du frontal Ingress

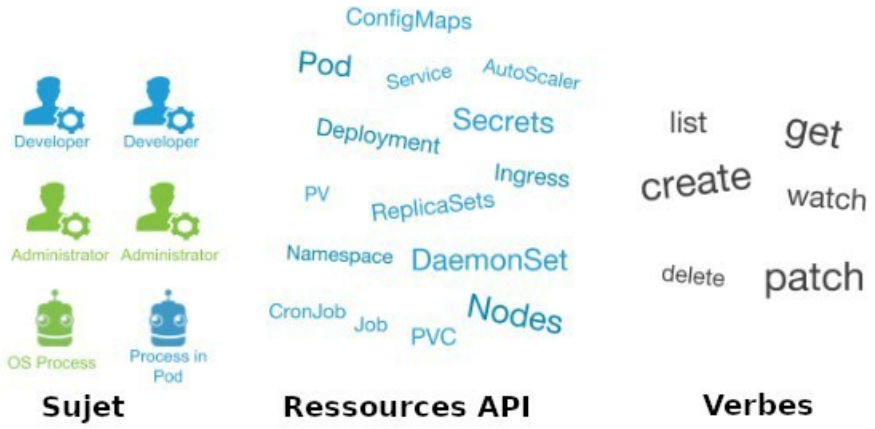


Kubernetes en production

Gestion des accès.

Il y a trois éléments en jeu :

- **Sujets** : L'ensemble des utilisateurs et des processus qui souhaitent accéder à l'API Kubernetes.
- **Ressources** : L'ensemble des objets API Kubernetes disponibles dans le cluster. Exemples : Pods, Déploiements, Services, Nœuds et PersistentVolumes, …
- **Verbes** : L'ensemble des opérations qui peuvent être exécutées sur les ressources ci-dessus. Exemples : get, watch, create, delete, … Qui correspondent en fait au quatuor Create, Read, Update et Delete (CRUD).

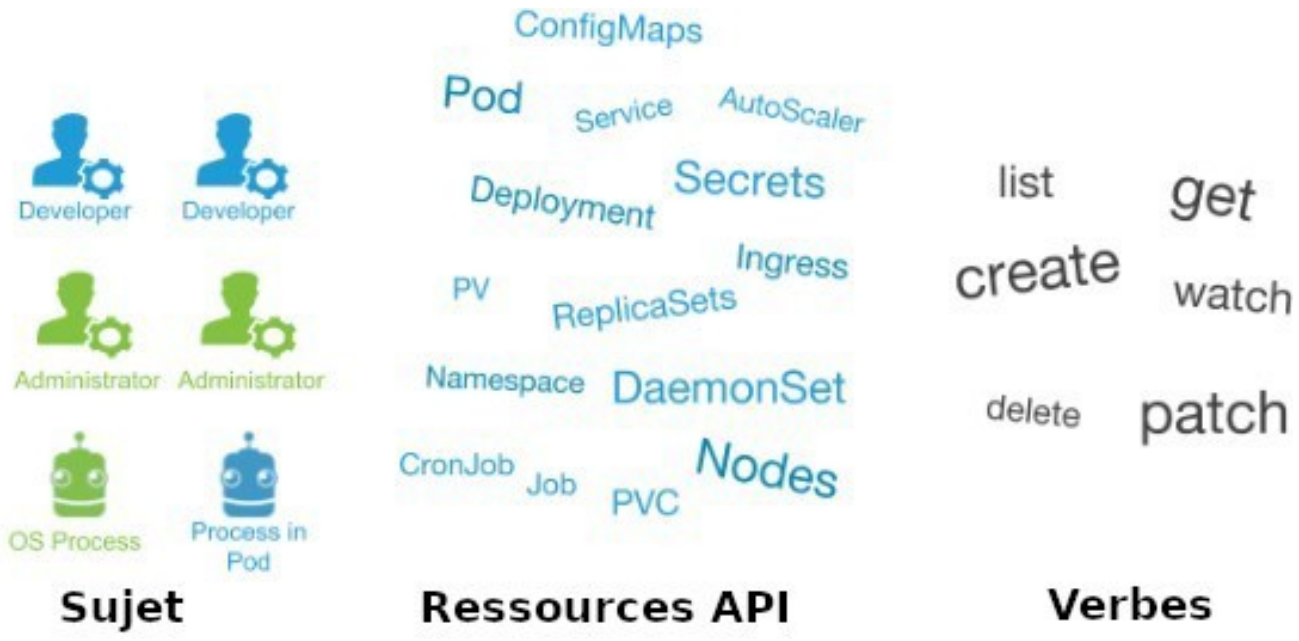


Kubernetes en production

Gestion des accès.

Les trois éléments en jeu :

- **Sujets**
- **Ressources**
- **Verbes**



Kubernetes en production

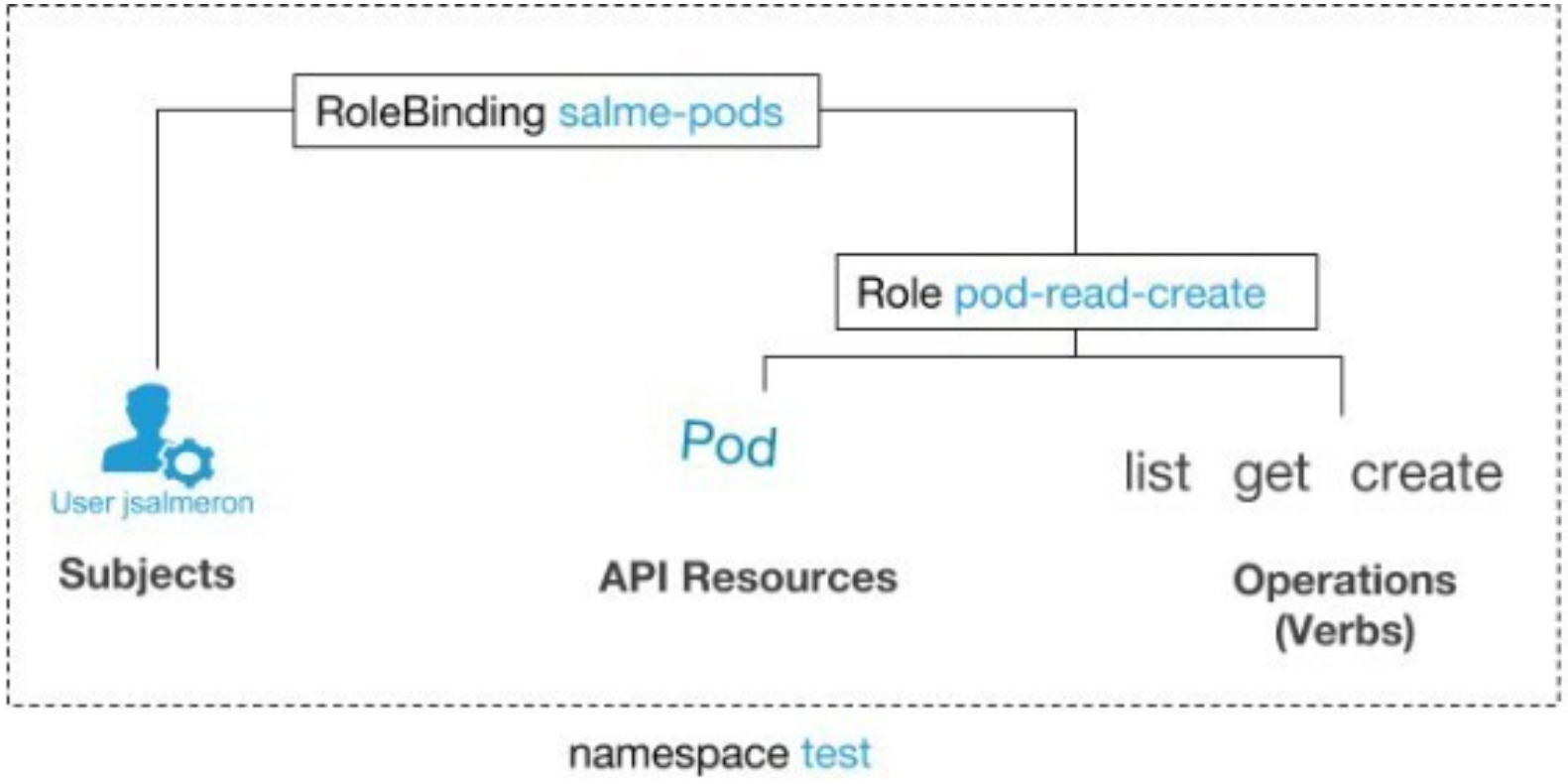
Gestion des accès.

Pour lier ces trois types d'entités, on utilise les différents objets API RBAC disponibles dans Kubernetes :

- **Role** : permet de relier les ressources API et les verbes. Ceux-ci peuvent être réutilisés pour différents sujets.
  Ils sont liés à un espace de nommage (nous ne pouvons pas utiliser de caractères génériques pour en représenter plus d'un, mais nous pouvons déployer le même objet de rôle dans des espaces de nommage différents).
- **ClusterRole** : comme le rôle mais pour le cluster entier.
- **RoleBinding** : relie les entités-sujets restants. Étant donné le rôle, qui lie déjà les objets API et les verbes, nous allons établir quels sujets peuvent l'utiliser.
- **ClusterRoleBinding** : comme le roleBinding mais pour le cluster entier.

Kubernetes en production

Gestion des accès.



Kubernetes en production

Gestion des accès.

Supposons que nous voulons un utilisateur jdoe qui puisse exécuter :

```
kubectl get pods --namespace test
kubectl describe pod --namespace test pod-name
kubectl create --namespace test -f pod.yaml
```

Mais se verra refuser :

```
kubectl get pods --namespace kube-system
kubectl get pod --namespace test -w
```

Kubernetes en production

Gestion des accès.

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: Role
metadata:
  name: pod-read-create
  namespace: test
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["get","list","create"]
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: jdoe-pods
  namespace: test
subjects:
  - kind: user
    name: jdoe
    apiGroup: rbac.authorization.k8s.io/v1beta1
roleRef:
  kind: Role
  name: ns-admin
  apiGroup: rbac.authorization.k8s.io/v1beta1
```

Kubernetes en production

Gestion des accès

Question : maintenant que l'utilisateur peut créer des pods, pouvons-nous en limiter le nombre ?

- Pour ce faire, d'autres objets, non directement liés à la spécification RBAC, permettent de configurer la quantité de ressources : Quota de ressources et limites.
- Il faut les vérifier pour configurer un aspect aussi vital du cluster.

Kubernetes en production

Gestion des accès

Quelle est la différence entre les utilisateurs réguliers et les ServiceAccounts.

- **Utilisateurs** : ceux-ci sont globaux et s'adressent aux êtres humains ou aux processus vivant en dehors du cluster.
- **ServiceAccounts** : destinés aux processus intra-cluster qui se déroulent à l'intérieur des pods.

Tous deux ont en commun de devoir s'authentifier par rapport à l'API pour effectuer un ensemble d'opérations sur un ensemble de ressources.

Leurs domaines semblent être clairement définis. Ils peuvent aussi appartenir à ce qu'on appelle des groupes

Kubernetes en production

Gestion des accès

un RoleBinding peut lier plus d'un sujet

ServiceAccounts ne peut appartenir qu'au groupe "system:serviceaccounts".

NB : les utilisateurs n'ont pas d'objet API Kubernetes associé.

Dès lors :

`kubectl create serviceaccount test-service` fonctionne mais

`kubectl create user jdoe` échoue

Kubernetes en production

Gestion des accès

Conséquence directe : si le cluster ne stocke aucune information sur les utilisateurs, l'administrateur devra gérer les identités en dehors du cluster.
Il y a différentes façons de le faire :

- Certificats TLS,
- jetons et
- OAuth2, ...

De plus, il faudra créer des contextes kubectl pour pouvoir accéder au cluster avec ces nouvelles informations d'identification.

Pour créer les fichiers d'identification, il est possible d’ utiliser les commandes de configuration de kubectl (qui ne nécessitent aucun accès à l'API de Kubernetes, donc elles peuvent être exécutées par n'importe quel utilisateur).

Kubernetes en production

Gestion des accès

RBAC et les déploiements

On peut maintenant définir ce qu'un utilisateur peut faire à l'intérieur du cluster. Cependant, qu'en est-il des déploiements qui nécessitent un accès à l'API Kubernetes ?

Prenons l’exemple d’une application d'infrastructure : RabbitMQ.

Ce conteneur accueille un plugin Kubernetes responsable de la détection des autres membres du cluster RabbitMQ.

Le processus à l'intérieur du conteneur nécessite d'accéder à l'API Kubernetes, et il faut configurer un ServiceAccount avec les privilèges RBAC appropriés.

Kubernetes en production

Gestion des accès

En matière de ServiceAccounts, la bonne pratique consiste :

- Créer un ServiceAccount par déploiement avec le minimum de privilèges pour travailler.

Dans le cas d'applications qui nécessitent un accès à l'API Kubernetes, on pourrait être tenté d'avoir un type de "ServiceAccount privilégié" qui pourrait faire presque tout dans le cluster. Mais des opérations non désirées pourraient être réalisées.

En outre, les différents déploiements auront des besoins différents en terme d'accès aux API, il est donc logique d'avoir différents ServiceAccounts pour chaque déploiement.

Quelle doit être la configuration RBAC appropriée pour notre déploiement RabbitMQ.

D’après la documentation, RabbitMQ interroge l'API Kubernetes pour la liste des Endpoints.

Kubernetes en production

Gestion des accès – le service

```
{{- if .Values.rbacEnabled }}
apiVersion: v1
kind: ServiceAccount
metadata:
  name: {{ template "rabbitmq.fullname" . }}
  labels:
    app: {{ template "rabbitmq.name" . }}
    chart: {{ template "rabbitmq.chart" .  }}
    release: "{{ .Release.Name }}"
    heritage: "{{ .Release.Service }}"
{{- end }}
```

Kubernetes en production

Gestion des accès – le role

```
{{- if .Values.rbacEnabled }}
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: {{ template "rabbitmq.fullname" . }}-endpoint-reader
  labels:
    app: {{ template "rabbitmq.name" . }}
    chart: {{ template "rabbitmq.chart" .  }}
    release: "{{ .Release.Name }}"
    heritage: "{{ .Release.Service }}"
rules:
- apiGroups: [""]
  resources: ["endpoints"]
  verbs: ["get"]
{{- end }}
```

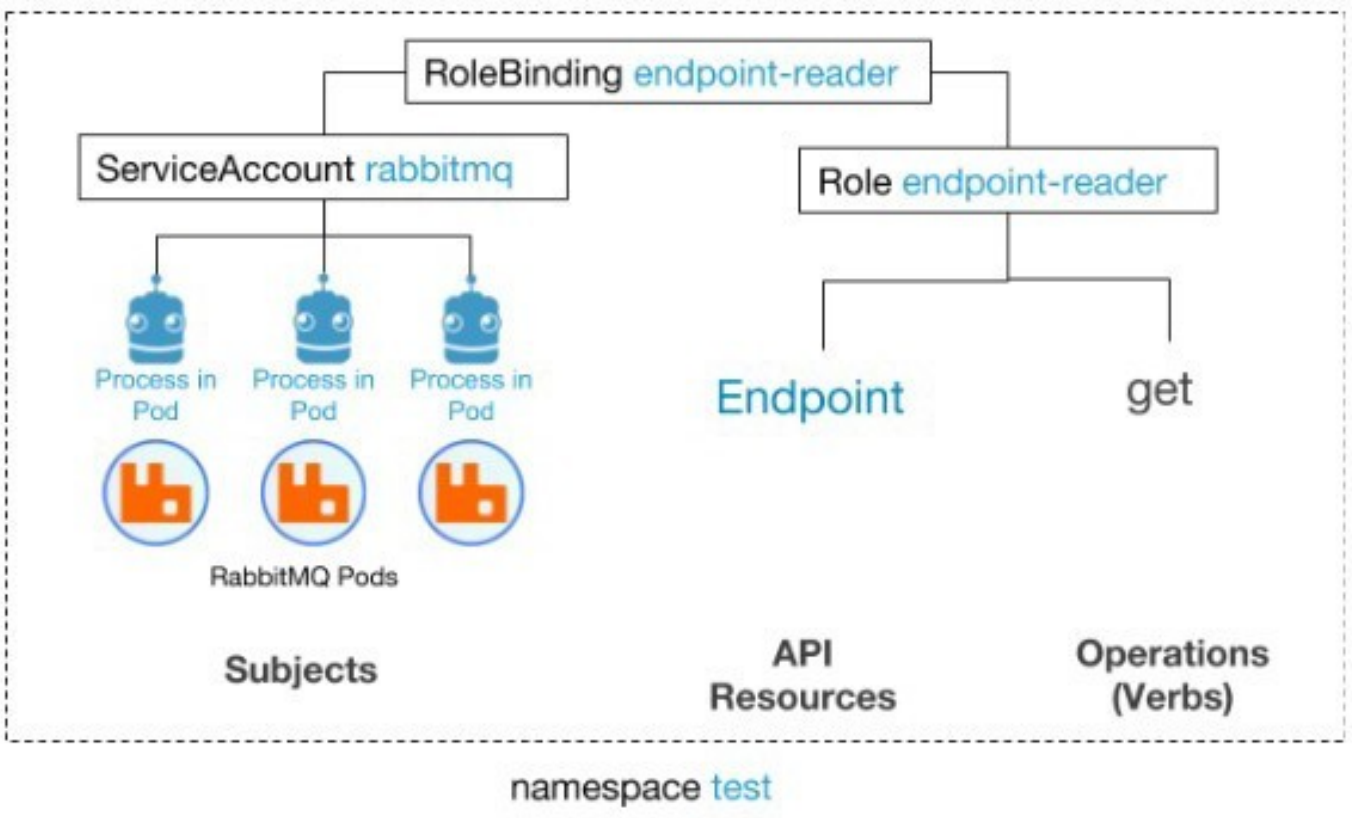
Kubernetes en production

Gestion des accès – le roleBinding

```
{{- if .Values.rbacEnabled }}
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: {{ template "rabbitmq.fullname" . }}-endpoint-reader
  labels:
    app: {{ template "rabbitmq.name" . }}
    chart: {{ template "rabbitmq.chart" .  }}
    release: "{{ .Release.Name }}"
    heritage: "{{ .Release.Service }}"
subjects:
- kind: ServiceAccount
  name: {{ template "rabbitmq.fullname" . }}
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: {{ template "rabbitmq.fullname" . }}-endpoint-reader
{{- end }}
```

Kubernetes en production

Gestion des accès



Travaux pratiques

- Création d'un namespace
- Ajout d'un utilisateur à ce namespace
- Gérer les droits d'accès d'une application

Kubernetes en production

Haute disponibilité.

Il y a deux approches différentes pour mettre en place un cluster Kubernetes hautement disponible avec kubeadm :

- Avec des nœuds master. Cette approche nécessite moins d'infrastructure. Les etcd et les masters sont situés au même niveau.
- Avec un cluster etcd. Cette approche nécessite davantage d'infrastructure. Les nœuds masters et les etcd sont séparés.

Avant de choisir, vous devriez examiner attentivement quelle approche répond le mieux aux besoins de vos applications et de votre environnement.

Vos clusters doivent exécuter Kubernetes version 1.12 ou ultérieure.

Sachez également que la mise en place de clusters HA avec kubeadm est encore expérimentale et sera encore simplifiée dans les versions futures. Vous pourriez rencontrer des problèmes lors de la mise à niveau de vos clusters, par exemple.

Kubernetes en production

Haute disponibilité.

Pour les deux méthodes, vous avez besoin de :

- ✔ Trois machines qui répondent aux exigences minimales de kubeadm pour les maîtres.
- ✔ Trois machines qui répondent aux exigences minimales de kubeadm pour les minions.
- ✔ Connectivité réseau complète entre toutes les machines du cluster (réseau public ou privé).
- ✔ Privilèges sudo sur toutes les machines.
- ✔ Accès SSH à partir d'un seul périphérique vers tous les nœuds du système.
- ✔ kubeadm et kubelet installés sur toutes les machines. kubectl est optionnel.

Pour le cluster externe etcd uniquement, vous avez également besoin de :

- ✔ Trois machines supplémentaires pour les membres etcd.

Kubernetes en production

Mode maintenance.

Si vous avez besoin de redémarrer un nœud (comme pour une mise à niveau du noyau, une mise à niveau de libc, une réparation matérielle, etc), et que le temps d'arrêt est bref, quand le Kubelet redémarre, il va tenter de redémarrer automatiquement les modules.

Si le redémarrage prend plus de temps (le temps par défaut est de 5 minutes, contrôlé par --pod-eviction-timeout sur le contrôleur-manager), alors le node controler les pods liés au noeud non disponible.

S'il existe un jeu de répliques correspondant (ou un contrôleur de réplication), alors une nouvelle instance du pod sera démarrée sur un autre nœud. Ainsi, dans le cas où tous les pods sont répliqués, les mises à niveau peuvent se faire sans action spéciale, en supposant que tous les nœuds ne s'arrêtent pas en même temps.

Si vous voulez plus de contrôle sur le processus de mise à niveau, vous pouvez utiliser le workflow suivant :

Utiliser cette commande kubectl pour rendre le nœud inactif :

```
$ kubectl drain $NODENAME
```

Kubernetes en production

Mode maintenance.

Ceci empêche les nouveaux pod d'être démarré sur ce nœud pendant l'arrêt.

Pour les pods avec un replicaSet, le pod sera remplacé par un nouveau pod qui sera programmé sur un nouveau nœud. De plus, si le pod fait partie d'un service, les clients seront automatiquement redirigés vers le nouveau pod.

Pour les pods sans replicaSet, vous devez lancer une nouvelle copie du pod, et en supposant qu'il ne fait pas partie d'un service, rediriger les clients vers celui-ci.

Effectuez les travaux de maintenance sur le nœud.

Pour rendre le nœud à nouveau utilisable :

```
$ kubectl uncordon $NODENAME
```

Travaux pratiques

Remplacement d’un node

Kubernetes en production

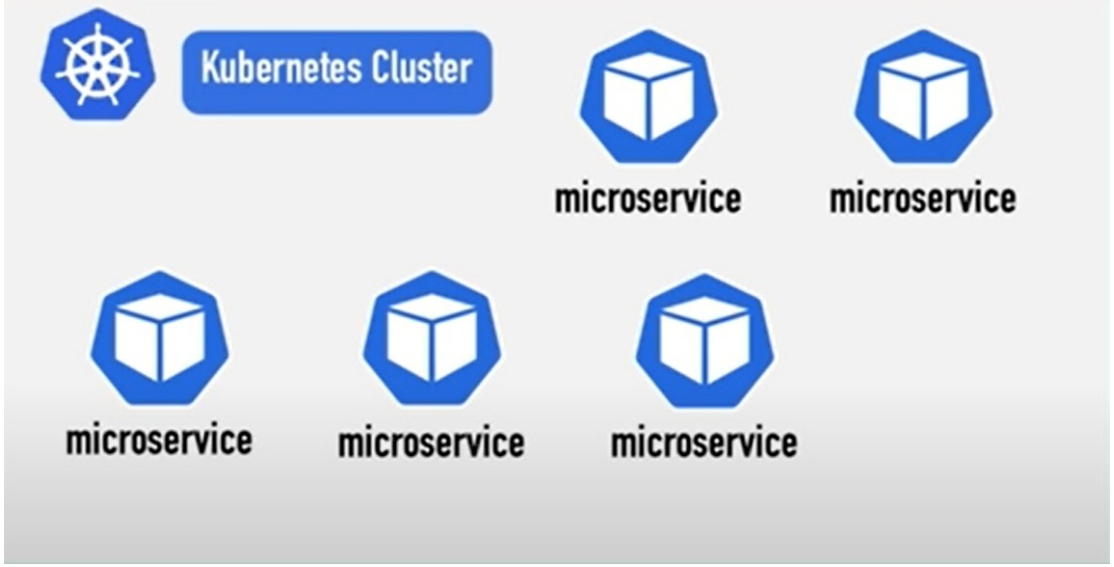
HELM – package manager pour Kubernetes

- Gestion de fichiers YAML
  - Pour les distribuer sur des dépôts publics et privés

- Les fichiers YAML au sein de Helm → Helm Charts
  - Ensemble de Fichiers Yaml
  - Créer ses propres Charts Helm avec Helm
  - Les « pusher » sur le dépôt Helm
  - Télécharger et utiliser les charts helm existants

Kubernetes en production

HELM – package manager

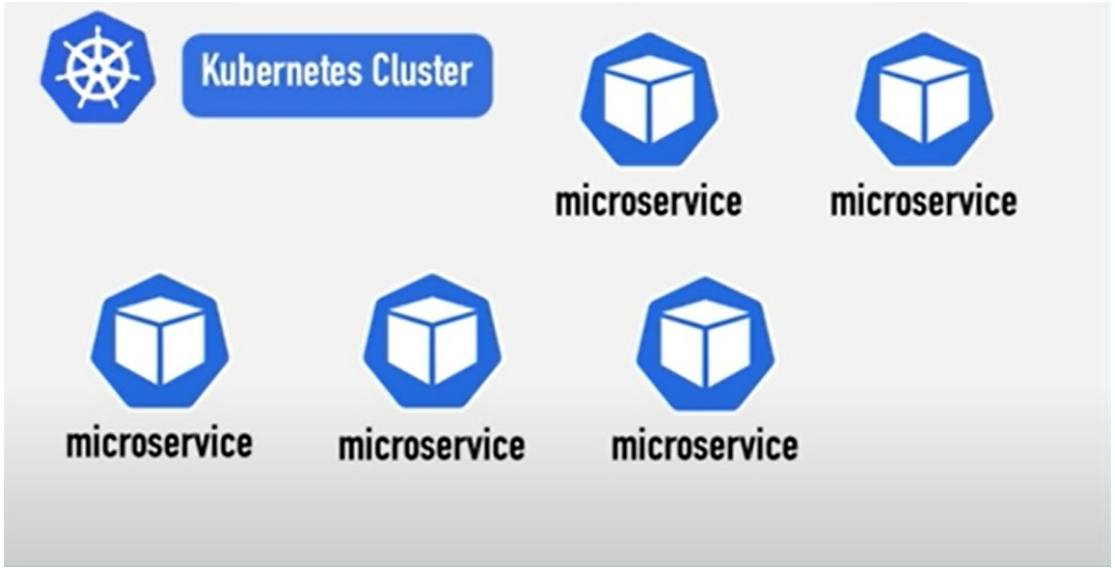



```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
  - name: my-app-container
    image: my-app-image
    port: 9001
```

- Helm permet de définir un modèle commun (« blue print »)

Kubernetes en production

HELM – package manager



template yaml config

```
apiVersion: v1
kind: Pod
metadata:
  name: {{ .Values.name }}
spec:
  containers:
  - name: {{ .Values.container.name }}
    image: {{ .Values.container.image }}
    port: {{ .Values.container.port }}
```

- Helm permet de définir un modèle commun (« blue print »)
- Les valeurs dynamiques sont remplacées par des « espaces réservés »

Kubernetes en production

HELM – package manager

values.yaml

```yaml
name: my-app
container:
  name: my-app-container
  image: my-app-image
  port: 9001
```

template yaml config

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: {{ .Values.name }}
spec:
  containers:
  - name: {{ .Values.container.name }}
    image: {{ .Values.container.image }}
    port: {{ .Values.container.port }}
```

- Les « values » sont définies dans les fichiers yaml ou avec le flag –set en ligne de commande
- Très pratique pour les pipelines CI/CD mis en place
  - Avant déploiement, dans le « build », remplacer les valeurs à la volée

Kubernetes en production

HELM – structure de chart helm

- mon-chart/
  - Chart.yaml meta → info sur le chart
  - values.yaml → valeurs pour les fichiers templates
  - charts/ → contient les dépendances charts
  - templates/ → emplacement des fichiers templates
  - …

- helm install « nom-du-chart »
- helm install - - set version:2.1.0 « nom-du-chart »
- helm install - - values=mes-valeurs.yaml « nom-du-chart »

Questions ?