Kubernetes

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.

Votre formateur

- Souha Boubaker
  - Docteure en informatique – Télécom SudParis
  - Ingénieure Full Stack/ DevOps
  - Coach de formation

- LinkedIn: https://www.linkedin.com/in/souha-boubaker-phd/

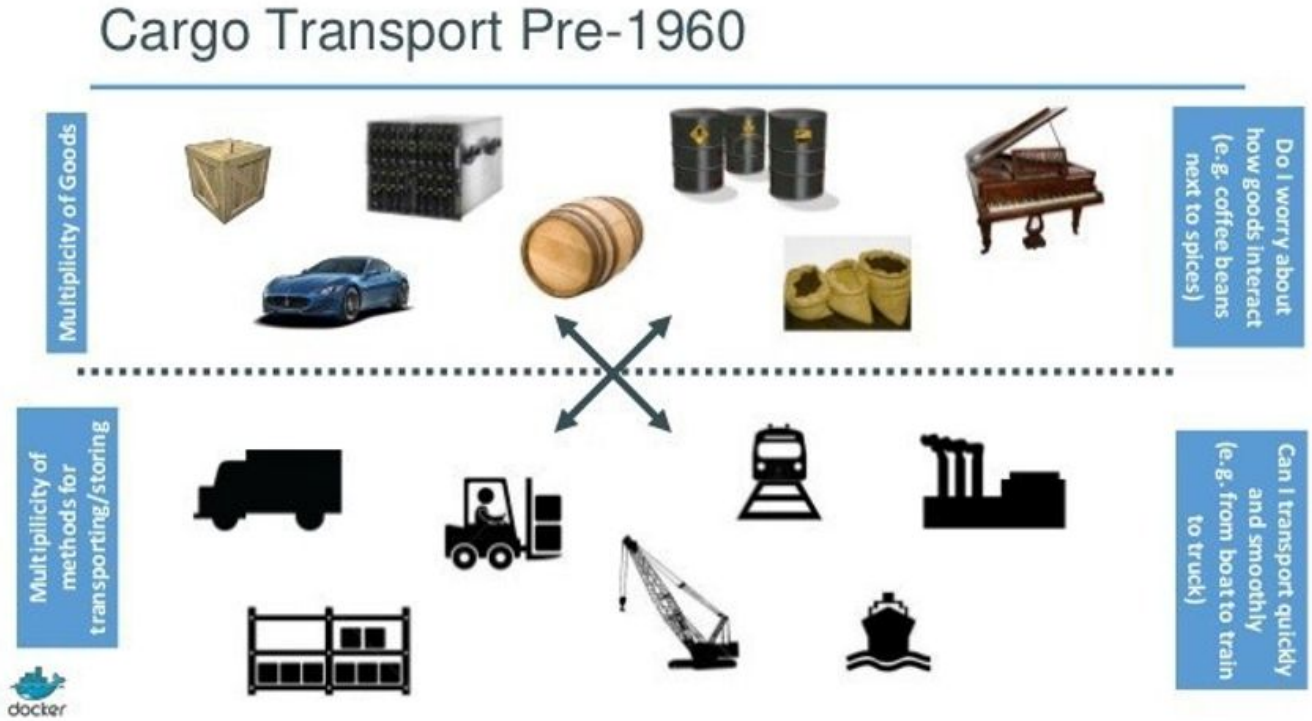
Votre formation - Logistique

- Horaires de la formation
  - 9h00-12h30
  - 14h00-17h30
- Pauses
  - 2 x 15 min

Votre formation - Programme

Introduction à Docker

Origine


Cargo Transport Pre-1960
Multiplicity of Goods
Do I worry about how goods interact (e.g. coffee beans next to spices)
Multiplicity of methods for transporting/storing
Can I transport quickly and smoothly (e.g. from boat to train to truck)
docker

Origine


Solution: Intermodal Shipping Container
Multiplicity of Goods
A standard container that is loaded with virtually any goods, and stays sealed until it reaches final delivery.
Do I worry about how goods interact (e.g. coffee beans next to spices)
...in between, can be loaded and unloaded, stacked, transported efficiently over long distances, and transferred from one mode of transport to another
Multiplicity of methods for transporting/storing
Can I transport quickly and smoothly (e.g. from boat to train to truck)
docker

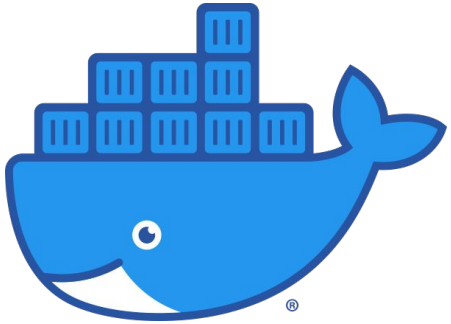
Origine

- Portabilité
- Variété
- Taille standardisée
- Isolation

Des conteneurs de logiciels

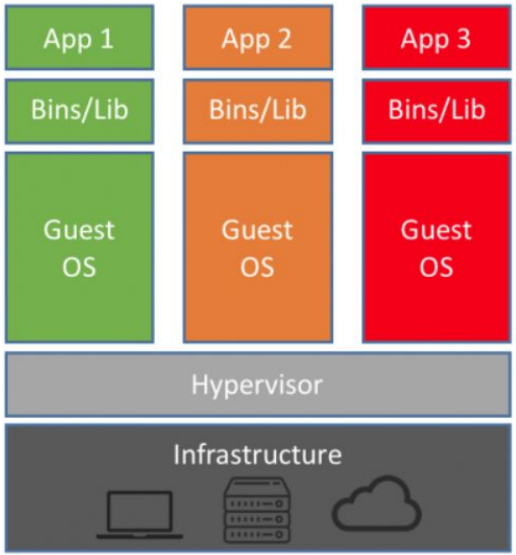
- Solution:
  - Facilement portable sur différents environnements
  - Grande variété de logiciels à packager
  - Création d’un standard pour le format des containers (OCI)
  - Isolation ses applications les unes des autres
  - Flexibilité/modularité des composants
  - Automatisable/Scriptable

Des conteneurs Docker

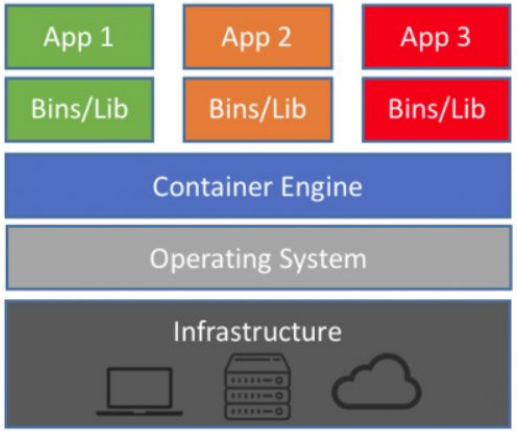


- Une des sociétés à avoir démocratisé l'utilisation des conteneurs Linux est **Docker**
- Créé en 2013, technologie **Open Source**
- Docker n'a pas créé la technologie : c'est un ensemble d'outils et d'API qui ont rendu les conteneurs beaucoup plus facilement gérables.
- L'application est packagée dans un conteneur virtuel exécutable sur n'importe quel type d'environnement
  - Package : une application et ses dépendances

De la virtualisation à Docker



virtualisation



Conteneurs

Build ship and run , any app and anywhere

De la virtualisation à Docker

- **Du point de vue du développeur ..**
  - 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, …)

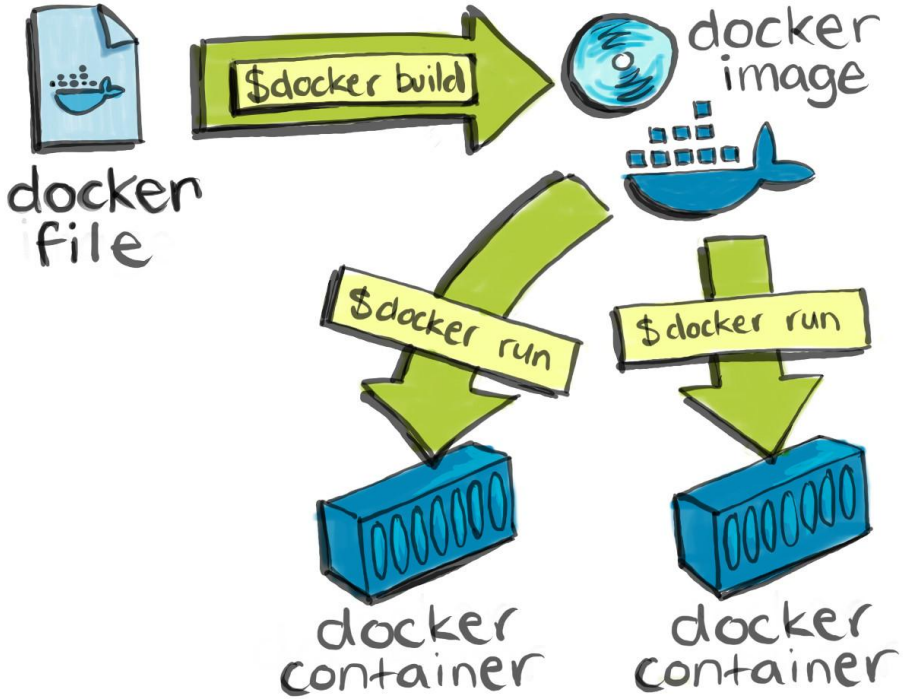
Terminologie et concepts fondamentaux

- Deux concepts centraux :
  - Une **image** : un modèle pour créer un conteneur
  - Un **conteneur** : l'instance qui tourne sur la machine.

- Autres concepts primordiaux :
  - Un **volume** : un espace virtuel pour gérer le stockage d'un conteneur et le partage entre conteneurs.
  - Un **registry** : un serveur où stocker des artifacts docker c'est à dire des images versionnées.
  - Un **orchestrateur** : un outil qui gère automatiquement le cycle de vie des conteneurs (création/suppression).

Terminologie et concepts fondamentaux

- Une image est le **résultat** d'un build



- Un conteneur est une **instance**

en cours de fonctionnement ("vivante")
d'une image

Introduction à Kubernetes

Kubernetes: une nouvelle ère pour les conteneurs ..

- Créé en 2014
- Le projet est développé en Open Source au sein de la **Cloud Native Computing Foundation**
- 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

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.

Qu'est-ce que Kubernetes ?

- Orchestrateur de conteneurs
- Offre la scalabilité
- Offre le self healing
- Offre la configuration déclarative
- Un standard à travers plusieurs vendeurs

Solutions d'installation

- Un bon moyen de commencer à travailler avec Kubernetes, c'est d'utiliser **MicroK8s, Minikube, K3s, Docker ou Kind**.
- MicroK8s sera déployer et utiliser tout au long de 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 haute disponible

Travaux pratiques

- Installer un cluster MicroK8s
- Premiers essais avec des commandes CLI
- Déployer une application nginx en CLI

Notre Cluster K8S

- Notre cluster k8s est plein d'objets divers, organisés entre eux de façon dynamique pour décrire des applications, tâches de calcul, services et droits d'accès.

- Lister les nodes pour récupérer le nom de l'unique node:
  - Kubectl get nodes
- Afficher ses caractéristiques
  - kubectl describe node <nom_node>

- Les commandes **get** et **describe** sont génériques et peuvent être utilisées pour tous les types de ressources.

Notre Cluster K8S

- Pour afficher tous les types de ressources à la fois que l'on utilise
  - Kubectl get all

une seule ressource dans notre cluster: le service d'API Kubernetes, pour qu'on puisse communiquer avec le cluster.

- Afficher les namespaces :
  - kubectl get namespaces
- Pour lister les ressources liées à un namespace :
  - kubectl get all -n kube-system
    - Les **namespaces** sont des groupes qui servent à isoler les ressources de façon logique et en termes de droits (avec le Role-Based Access Control (RBAC) de Kubernetes).

Déploiement d'une application en CLI

Pour créer un déploiement et du pod nginx:

- kubectl create deployment nginx --image=docker.io/library/nginx:latest

Afficher les pods :

- kubectl get pods

Pour créer un service NodePort :

- kubectl expose deployment/nginx --type="NodePort" --port 80

Afficher les services :

- kubectl get svc

Aller sur le navigateur et taper :

- http://localhost:<port_nodeport>

Les fichiers descriptifs

Les fichiers descriptifs

- **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.

- 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.

Syntaxe YAML

- La syntaxe du flux YAML est relativement simple, efficace, moins verbeuse que du XML, moins compacte cependant que du CSV.

- Les commentaires (**#**)
- Alignement ! (indentation de 2 espaces !!)
- Les listes (tirets **-**, suivi d'une espace, à raison d'un élément par ligne)
- Des paires **clé: valeur**

```yaml
- marché:
    lieu: Marché de la Place
    jour: jeudi
    horaire:
      unité: "heure"
      min: 12
      max: 20
    fruits:
      - nom: pomme
        couleur: "verte"
        pesticide: avec

      - nom: poires
        couleur: jaune
        pesticide: sans
    légumes:
      - courgettes
      - salade
      - potiron
```

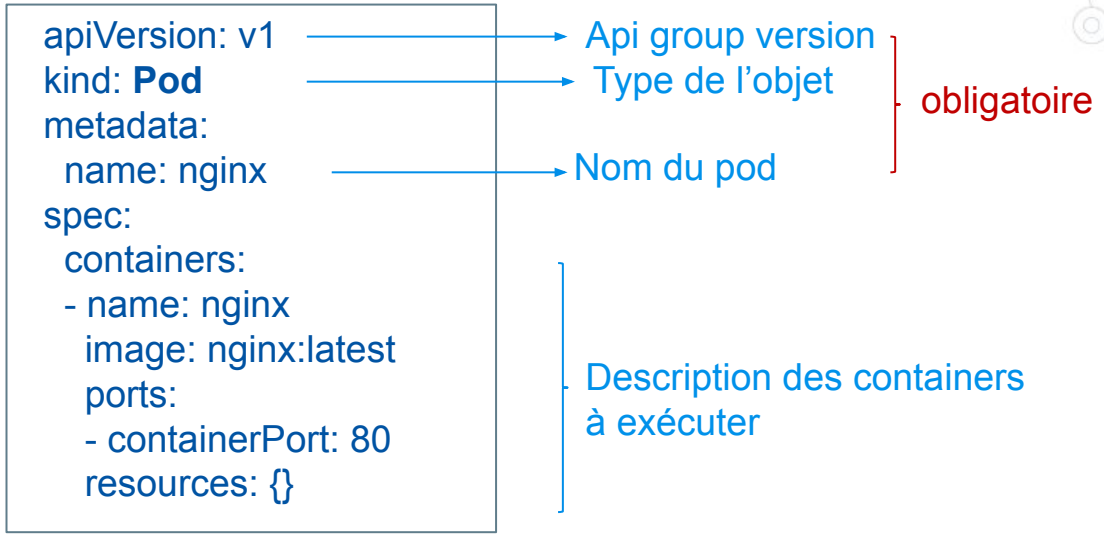
Création d'objets Kubernetes

- On définit des objets Kubernetes généralement via l'interface en ligne de commande et **kubectl** de deux façons :
  - en lançant une commande
    kubectl run <conteneur> ..., kubectl expose ..., kubectl create …, etc.
  - en décrivant un objet dans un fichier **YAML** ou JSON et en lançant la commande
    kubectl apply -f objet.yml

    - il est recommandé d'utiliser une description YAML et versionnée des objets et des configurations Kubernetes plutôt que la CLI

Description des objets Kubernetes

- **Pod**:



Scalabilité d'un déploiement

- Pour augmenter le nombre de répliques du déploiement en CLI

  kubectl scale deployments <nom_deploiement> --replicas=4

- avec YAML

  kubectl scale –f deployment.yaml –replicas=4

Ou bien modifier le fichier YAML

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 4
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
          - containerPort: 80
…
```

Suppression d’un objet

- Pour supprimer un objet en CLI

  kubectl delete <nom_objet>

- Supprimer un objet en YAML

  kubectl delete -f objet.yml

Travaux pratiques

- Créer un pod à l’aide d’un fichier yaml

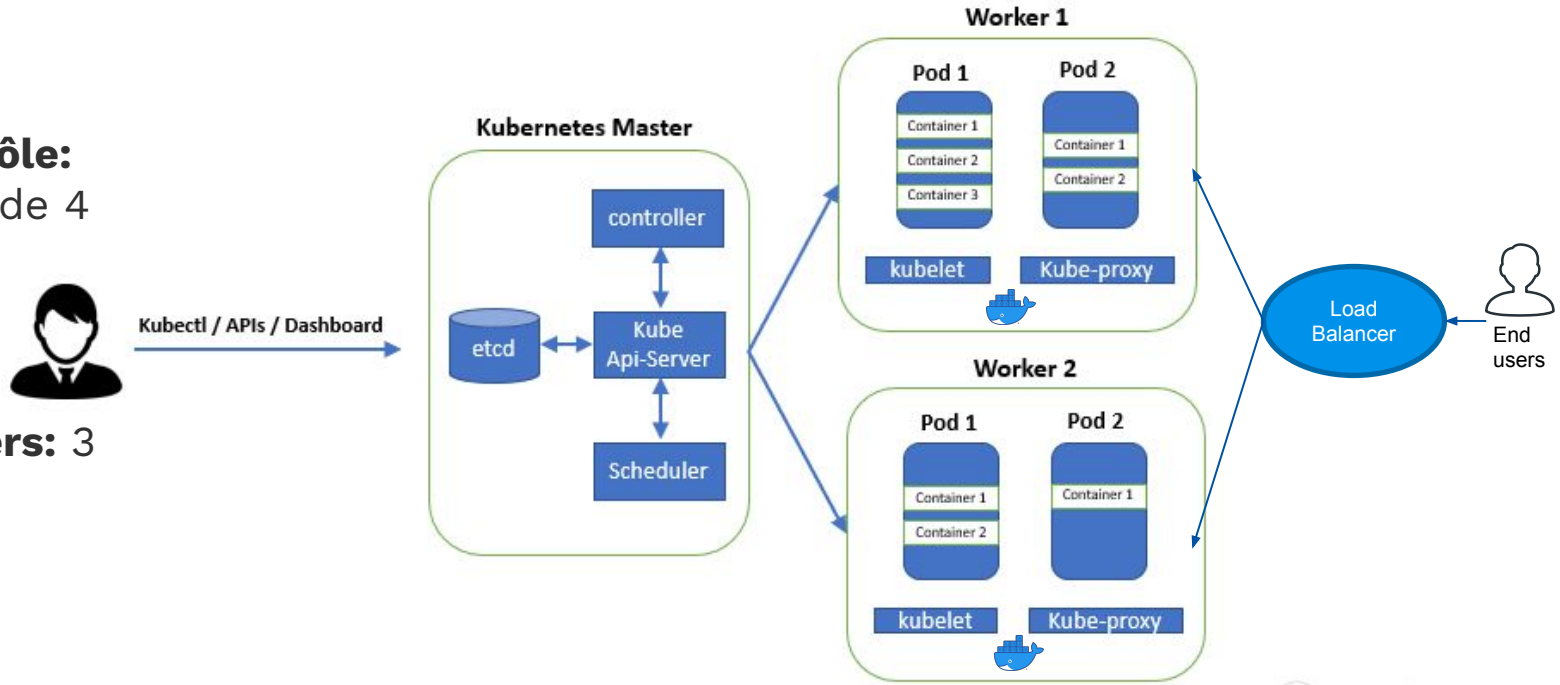
Architecture de Kubernetes

Architecture de Kubernetes

Mater/slave

Plan de contrôle: nœud maître de 4 composants

Nœuds workers: 3 composants



Architecture de Kubernetes

- Le plan de contrôle: **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 c'est lui qui reçoit tous les appels et demandes, externes ou internes.
  - 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 méthode par défaut pour interagir avec le cluster Kubernetes depuis un ordinateur local.
  - Il est responsable de s'assurer que le stockage etcd et les services actifs sont synchrones.

Architecture de Kubernetes

- Le plan de contrôle: **etcd**
  - Une solution de stockage clé/valeurs légère et distribuée .
  - stocke les données de configuration accessibles par chacun des nœuds du cluster.
  - peut être configuré sur un seul serveur maître ou, dans les scénarios de production, réparti sur plusieurs nœuds.
  - fourni une API HTTP/JSON.

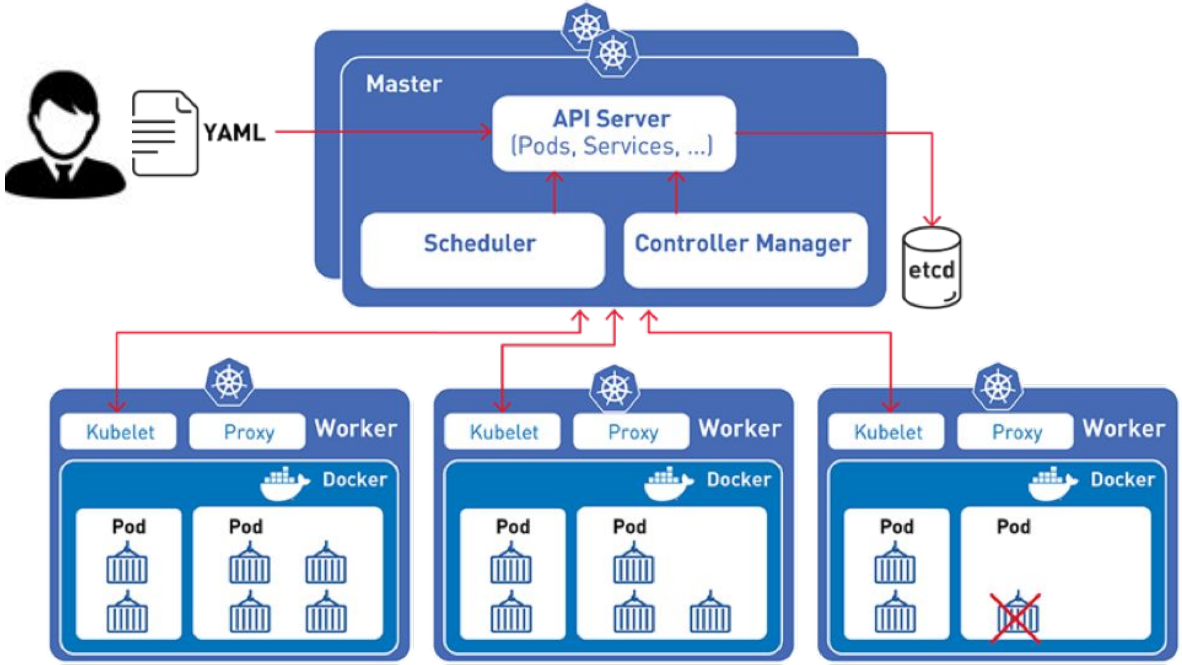
Architecture de Kubernetes

- Le plan de contrôle: **kube-controller-manager**
  - 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 de Kubernetes

- Le plan de contrôle: **kube-scheduler**
  - **l'ordonnanceur** : Le processus qui dispatche les charges de travail à des nœuds spécifiques du cluster
  - 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.
    - lit les exigences opérationnelles d'une charge de travail,
    - analyse l'environnement d'infrastructure actuel,
    - place le travail sur un ou plusieurs nœuds acceptables.

Architecture de Kubernetes



Architecture de Kubernetes

- Nœud de travail: **kubelet**
  - Point de contact principal pour chaque nœud avec le cluster.
  - 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.
  - Communique avec les composants maîtres pour s'authentifier au cluster et recevoir les commandes et le travail à effectuer.
  - Examine les spécifications de chaque pod et fait en sorte que les containers définis dans ces PodSpecs tournent et sont en bonne santé.

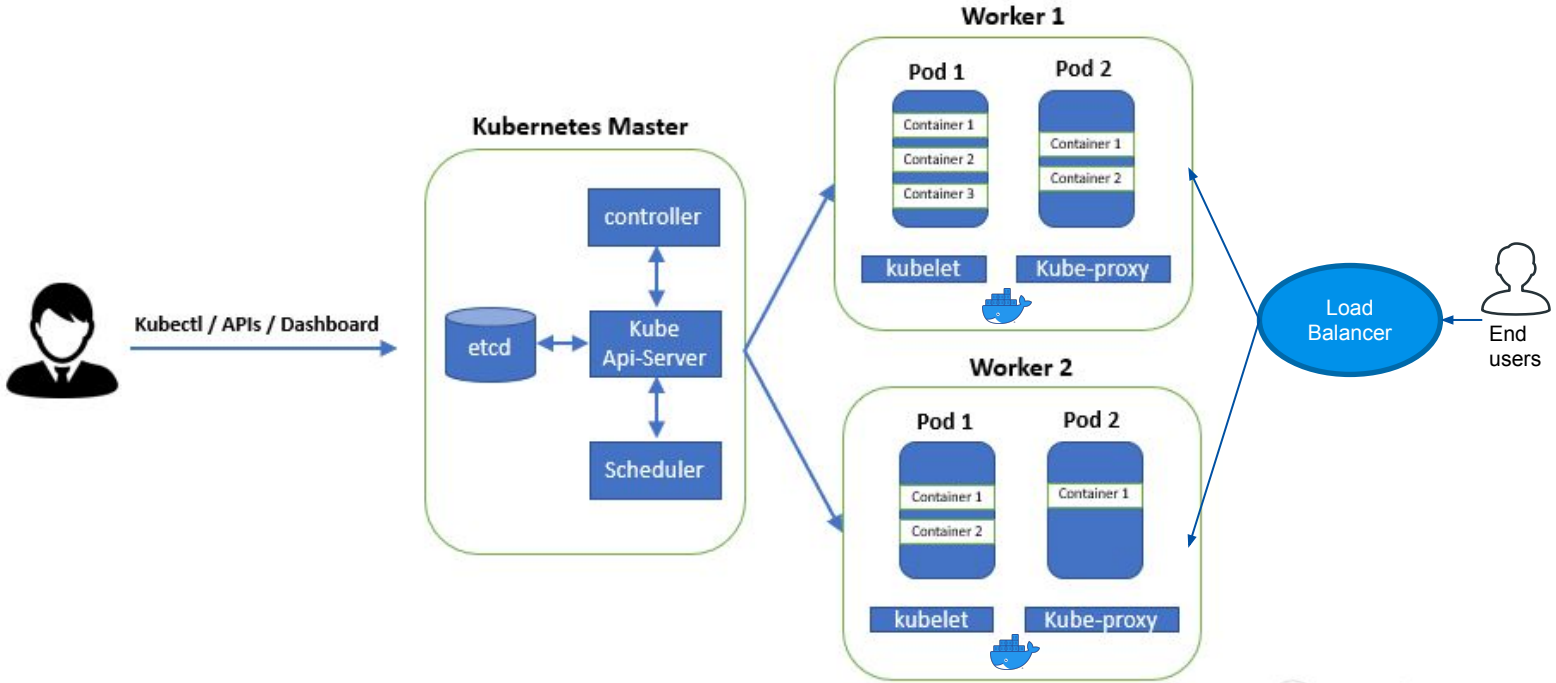
Architecture de Kubernetes

- Nœud de travail: **kube-proxy**
  - maintient les règles réseau sur les nœuds. Ces règles permettent la communication vers les pods depuis l'intérieur ou l'extérieur de votre cluster.
  - kube-proxy utilise la couche de filtrage des paquets du système d'exploitation, sinon il le fait lui-même.

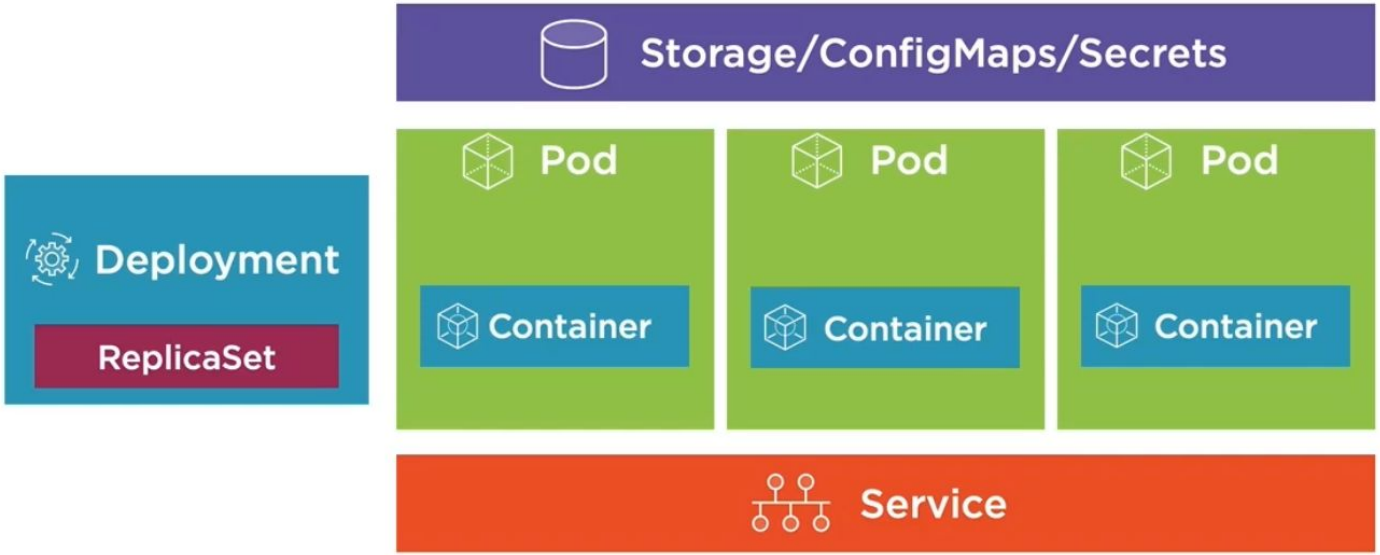
Architecture de Kubernetes

- Nœud de travail: **container runtime engine**
  - La base de chaque nœud est un runtime de conteneur.
  - Le runtime est responsable du démarrage et de la gestion des conteneurs, applications encapsulées dans un environnement d'exploitation relativement isolé mais léger.
  - Généralement, cette exigence est satisfaite en installant et en exécutant Docker, mais des alternatives comme rkt et runc sont également disponibles.

Architecture de Kubernetes



Objets de base Kubernetes

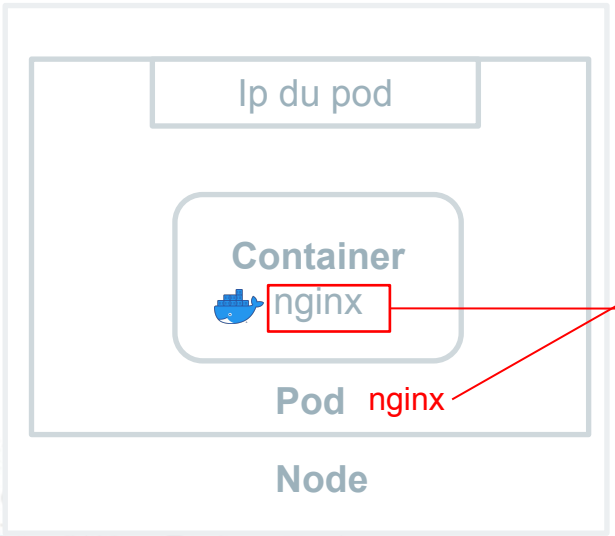


Objets de base Kubernetes

- **Pod**:
  - Un pod est l'unité la plus basique avec lequel Kubernetes fonctionne.
  - 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 programmés sur le même nœud.
  - Ils sont gérés comme une unité et partagent leur environnement, leurs volumes et leur espace IP.

Description des objets Kubernetes

- **Pod:**




```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:latest
    ports:
    - containerPort: 80
    ressources: {}
```

Api group version

Type de l'objet

obligatoire

Nom du pod

Description des containers à exécuter

Objets de base Kubernetes

- **Pod** lifecycle:
  - Pending
  - Containercreating
  - Running
  - Error
  - Crashloopbackoff
  - Succeded

- Les **pods** sont éphémères, leurs IPs changent

Objets de base Kubernetes

- **Service**:
  - Désigne un ensemble de pods (grâce à des tags) généralement géré par un déploiement.
  - Fournit un endpoint réseau pour les requêtes à destination de ces pods.
  - Configure une politique permettant d'y accéder depuis l'intérieur ou l'extérieur du cluster.

- Les **Services** ne sont pas éphémères,

Objets de base Kubernetes

- **Service**: 4 types
  - **ClusterIP:** expose le service sur **une IP interne** au cluster (type par défaut). Les autres pods peuvent alors accéder au service de l'intérieur du cluster, mais pas l'extérieur
  - **NodePort:** expose le service depuis **l'IP de chacun des noeuds** du cluster en ouvrant un port directement sur le nœud (entre 30000 et 32767)
  - **LoadBalancer:** expose le service en externe à l'aide d'un Loadbalancer de fournisseur de cloud. Les services NodePort et ClusterIP, vers lesquels le Loadbalancer est dirigé sont automatiquement créés.
  - **ExternalName:** mappe un service à un nom DNS

Objets de base Kubernetes

Service :

Service sera appliqué à tous les pods ayant label **app: nginx**

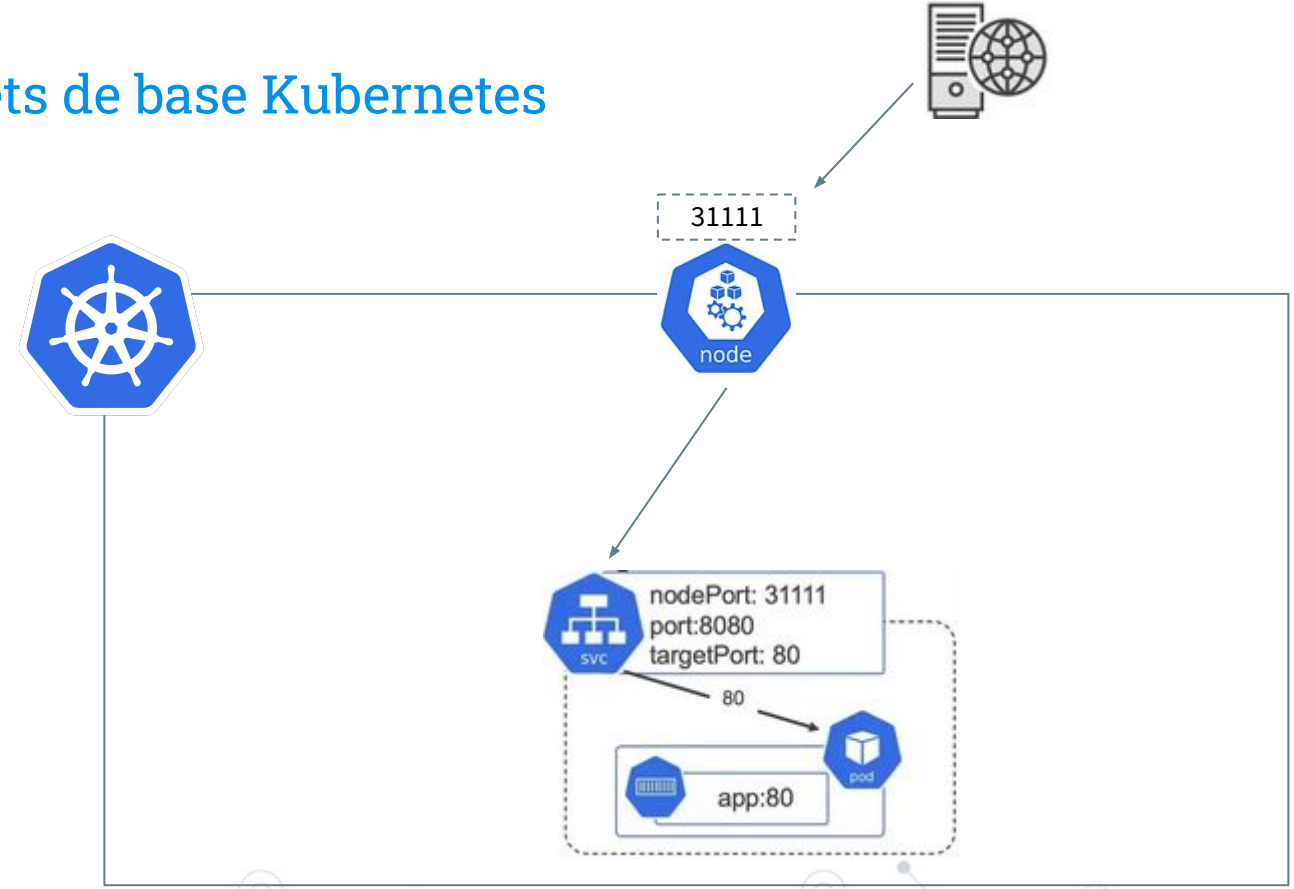
Un pod peut accéder à un service en utilisant **name: port → nginx-svc:80**

On n'a plus besoin de chercher les IPs

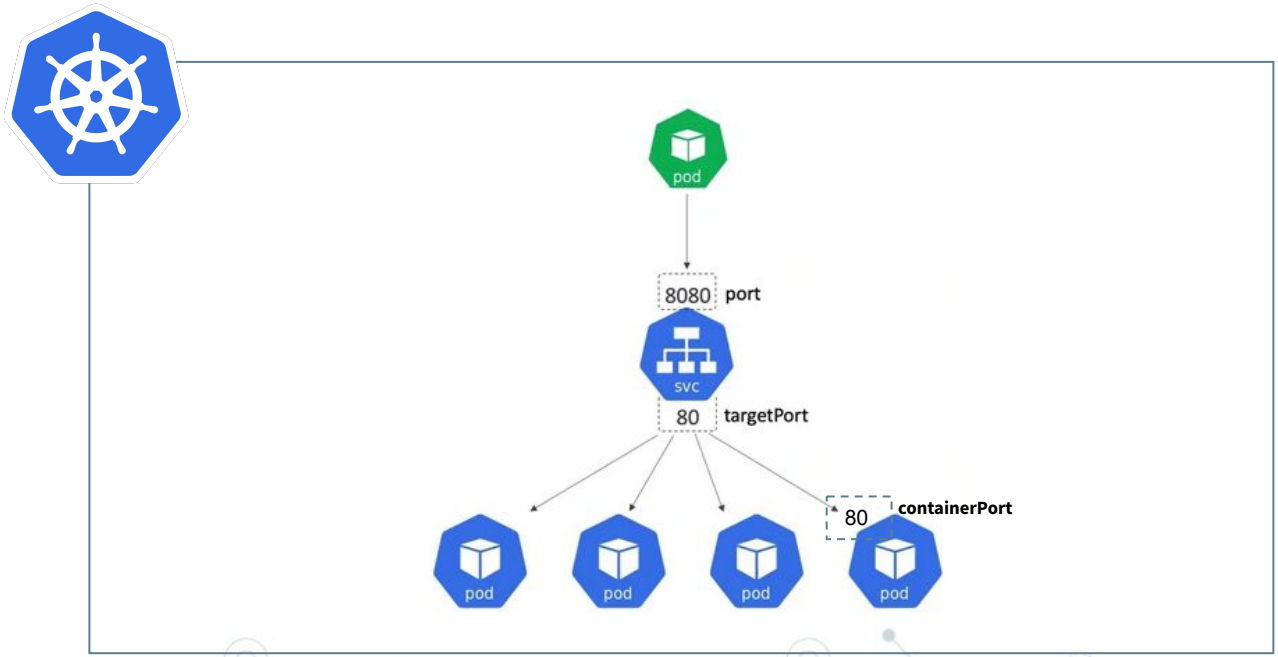
```
apiVersion: v1
kind: Service
metadata:
  name: nginx-svc
  labels:
    app: nginx
spec:
  type: NodePort
  selector:
    app: nginx
  ports:
  - name: http
    port : 80
    targetPort: 80
    nodePort: 31000
```

Optionnel:
entre 30000 et 32767

Objets de base Kubernetes

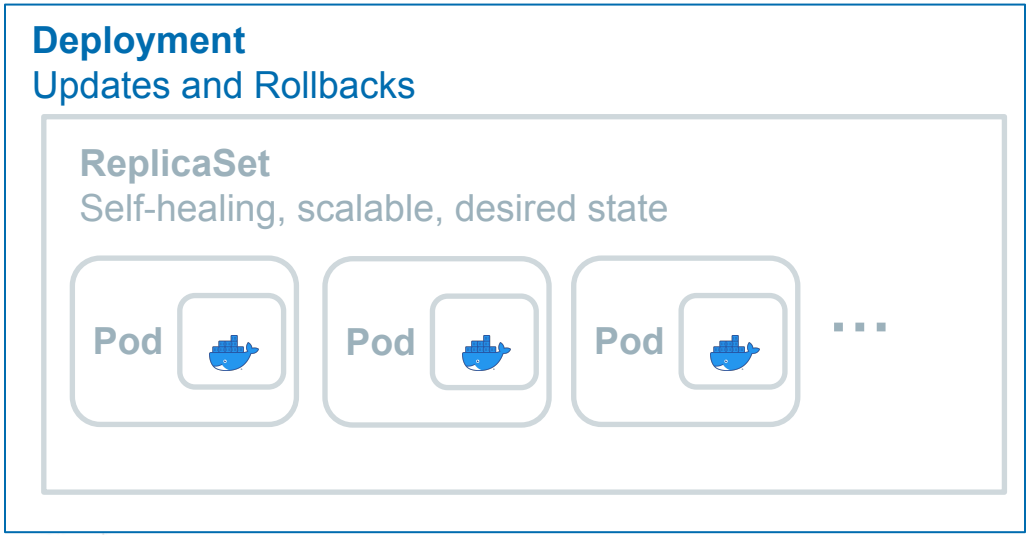


Objets de base Kubernetes



Objets de base Kubernetes

- **Deployment** et **replicaSet**:



Objets de base Kubernetes

- **Deployment** et **replicaSet**:
  - Le **deployment** gère la coexistence et le **tracking de versions** multiples d'une application et d'effectuer des montées de version automatiques en haute disponibilité en suivant une **RolloutStrategy**
  - Lors des changements de version, un seul **deployment** gère automatiquement de multiples **replicasets** contenant chacun **une version** de l'application
  - Les **ReplicaSet**s gèrent :
    - la réplication : avoir le bon nombre d'instances et le scaling
    - la santé et le redémarrage automatique des pods de l'application (Self-Healing)

Objets de base Kubernetes

- **Deployment** :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

Objets de base Kubernetes

- En plus du déploiement d'un application, Il existe pleins d'autre raisons de créer un ensemble de Pods :
  - Le **DaemonSet**: Faire tourner un agent ou démon sur chaque nœud, par exemple pour des besoins de monitoring, ou pour configurer le réseau sur chacun des nœuds.
  - Le **Job** : Effectuer une tache unique de durée limitée et ponctuelle, par exemple de nettoyage d'un volume ou la préparation initiale d'une application, etc.
  - Le **CronJob** : Effectuer une tache unique de durée limitée et récurrente, par exemple de backup ou de régénération de certificat, etc.
  - Le **StatefulSet** : plus adapté pour des applications statefull comme les bases de données de toutes sortes qui ont besoin de persister des données critiques

Objets de base Kubernetes

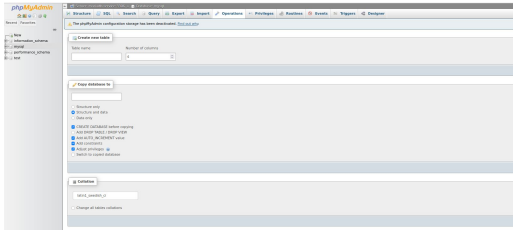
- Les **Deployments** (liés à des **ReplicaSets**) doivent être utilisés :
  - lorsque votre application est complètement découplée du nœud
  - que vous pouvez en exécuter plusieurs copies sur un nœud donné sans considération particulière
  - que l'ordre de création des replicas et le nom des pods n'est pas important
  - lorsqu'on fait des opérations stateless
- Les **DaemonSets** doivent être utilisés :
  - lorsqu'au moins une copie de votre application doit être exécutée sur tous les nœuds du cluster (ou sur un sous-ensemble de ces nœuds).
- Les **StatefulSets** doivent être utilisés :
  - lorsque l'ordre de création des replicas et le nom des pods est important
  - lorsqu'on fait des opérations stateful (écrire dans une base de données)

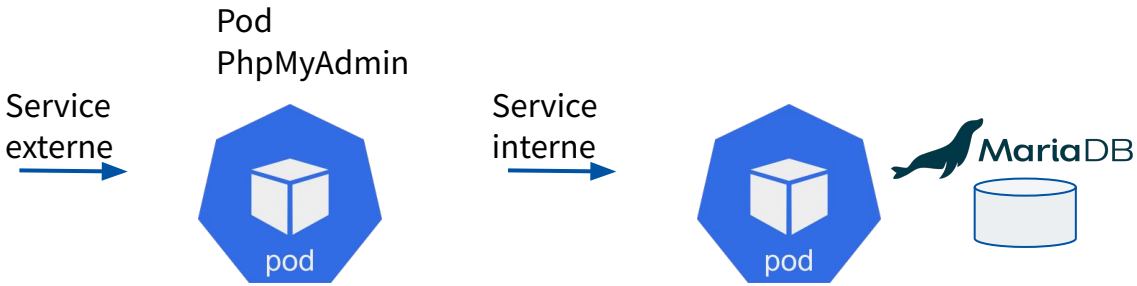
Objets de base Kubernetes

- **DaemonSet** :

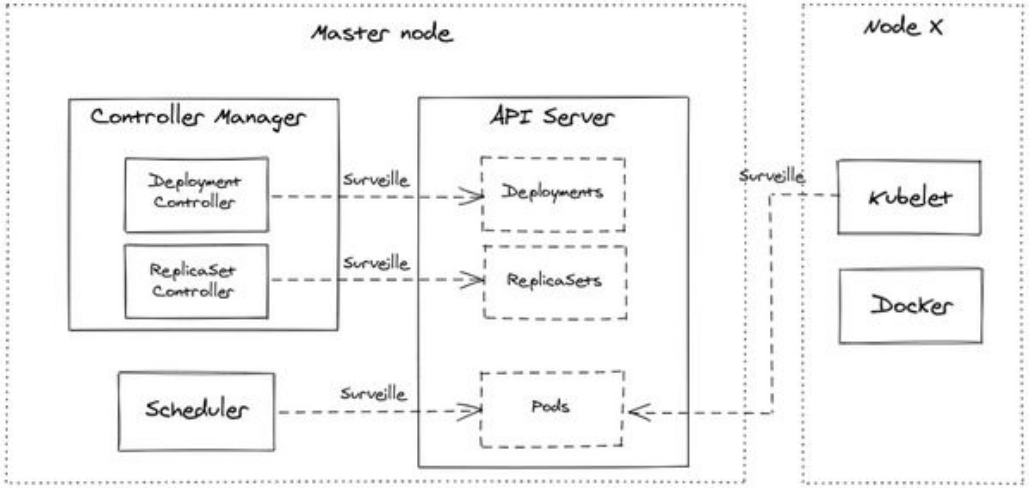
```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  labels:
    app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
      - key: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

Travaux pratiques

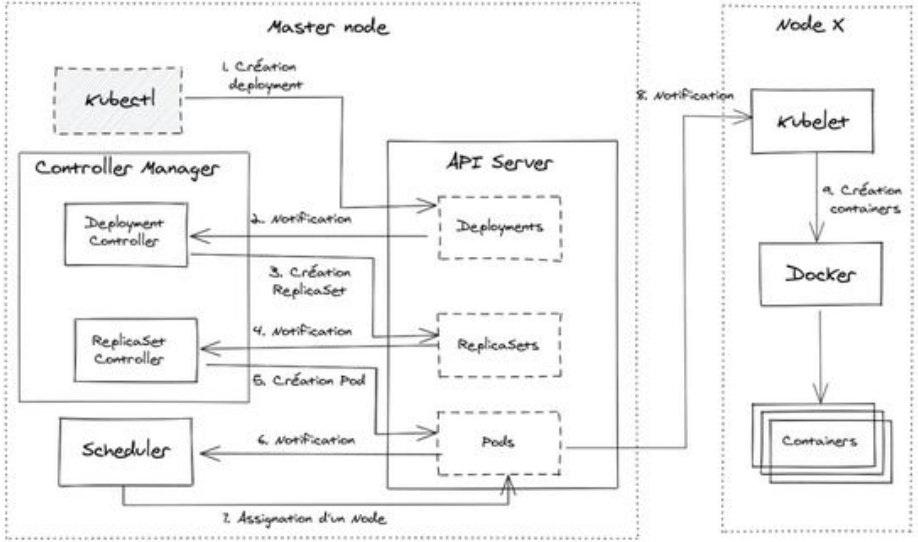




Architecture de Kubernetes



Architecture de Kubernetes



Exploiter Kubernetes

Les Volumes Kubernetes

- Kubernetes fournit la possibilité de monter des volumes virtuels dans les conteneurs de nos pods.

- On liste séparément les volumes de notre pod puis on monte un ou plusieurs dans les différents conteneurs

Les Volumes Kubernetes

HostPath

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

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

```
apiVersion: v1
kind: Pod
metadata:
…
spec:
  containers:
   - name: nginx
     image: nginx:latest
     ports:
     - containerPort: 80
     volumeMounts:
     - name: html
       mountPath: /usr/share/nginx/html
  volumes:
  - name: html
    hostPath:
      path: /chemin/dossier/sur/host
      type: <type>
```

Le point de montage du volume dans le conteneur

Le point de montage du volume dans le pod

Les Volumes Kubernetes

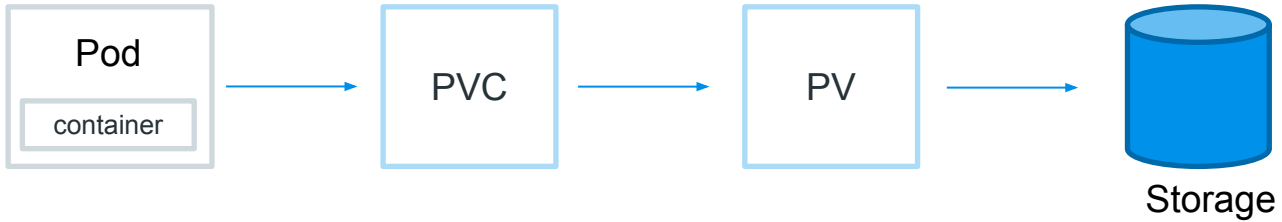
- Les Cloud providers proposent plusieurs types de volume:
  - Azure: Azure Disk an Azure File
  - AWS: Elastic Block Store
  - GCP: GCE Persistent Disk
  - Etc.

```
apiVersion: v1
kind: Pod
metadata:
…
spec:
  containers:
   - name: mon_container
     image: mon_image
     ports:
     - containerPort: 80
     volumeMounts:
     - name: data
       mountPath: /data/storage
  volumes:
  - name: data
    azureFile:
      secretName: <azure_secret>
      shareName: <share_name>
      readOnly: false
```

Les Volumes Kubernetes

PVC, PV et SC

- ***PersistentVolumeClaim*** (PVC) : une demande de ***volume persistant*** (PV)
- 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***.
- PV et PVC sont indépendants du cycle de vie des Pods et préservent les données en redémarrant, reprogrammant et même en supprimant les Pods.



Les Volumes Kubernetes

PVC et PV

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: data-pv
  labels:
    type: local
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /test/mysqldata
```

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: data-pvc
spec:
  storageClassName: ""
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

```
apiVersion: apps/v1
kind: Deployment
…
spec:
…
  template:
…
  spec:
    containers:
…
      volumeMounts:
      - name: data-mysql
        mountPath: /var/lib/mysql
    volumes:
    - name: data-mysql
      persistentVolumeClaim:
        claimName: data-pvc
```

Le point de montage du volume dans le conteneur

Les Volumes Kubernetes

PVC et PV

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: data-pv
  labels:
    type: local
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  azureFile:
   secretName: <azure_secret>
   shareName: <share_name>
   readOnly: false
```

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: data-pvc
spec:
  storageClassName: ""
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

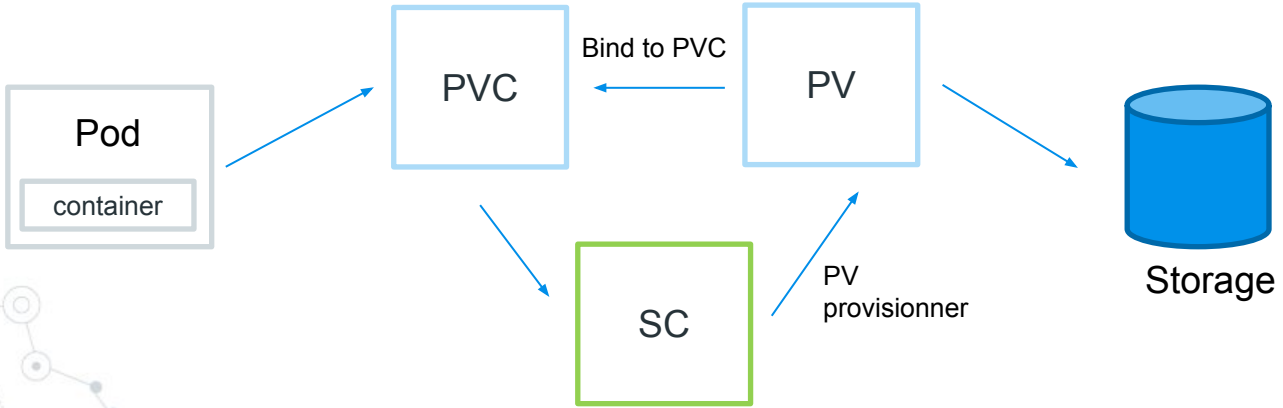
```
apiVersion: apps/v1
kind: Deployment
…
spec:
…
 template:
…
 spec:
   containers:
…
     volumeMounts:
     - name: data-mysql
       mountPath: /var/lib/mysql
   volumes:
   - name: data-mysql
     persistentVolumeClaim:
       claimName: data-pvc
```

Le point de montage du volume dans le conteneur

Les Volumes Kubernetes

PVC, PV et SC

- les ***StorageClasses*** fournissent du stockage
- les conteneurs demandent du volume avec les ***PersistentVolumeClaims***
- les ***StorageClasses*** répondent aux ***PersistentVolumeClaims*** en créant des objets ***PersistentVolumes*** : le conteneur peut accéder à son volume.



Les Volumes Kubernetes

PVC, PV et SC

- **`microk8s enable hostpath-storage:`** active la storageClass par défaut de microk8s

➢ `kubectl get pv,pvc,sc`

➢ `microk8s kubectl describe pv <pv_name>`

→ `le path du volume par défaut:`

/var/snap/microk8s/common/default-storage/default-data-pvc-pvc-1d9205c2- ….

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: data-pvc
spec:
  storageClassName: microk8s-hostpath
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

← Optionnel

Les Volumes Kubernetes

- **PVC, PV et SC:** Personnaliser le path utilisé par le PV

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
   name: custom-sc
provisioner: microk8s.io/hostpath
reclaimPolicy: Delete
parameters:
   pvDir: /test/mysqlData
volumeBindingMode: WaitForFirstConsumer
```

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: data-pvc
spec:
  storageClassName: custom-sc
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Paramétrer les Pods

Secrets et **configMap**

- Il est souvent nécessaire de référencer des données "spéciales", tels que des clés API, des jetons et d'autres secrets. L'application peut être paramétrable 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**.

Paramétrer les Pods

Secrets et **configMap**

- Les **Secrets** se manipulent comme des objets **ConfigMaps**, mais sont faits pour stocker des mots de passe, des clés privées, des certificats, des tokens, ou tout autre élément de config dont la confidentialité doit être préservée.
- 2 utilisations:
  - un fichier que l'on monte en tant que volume dans un conteneur (pas nécessairement disponible à l'ensemble du pod)
  - une variable d'environnement du conteneur.

Paramétrer les Pods

Secrets et **configMap**

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password:
MWYyZDFlMmU2N2Rm
```

```
apiVersion: apps/v1
kind: Deployment
…
spec:
…
 template:
…
 spec:
  containers:
…
     envFrom:
     -  secretRef:
           name: mysecret
```

```
apiVersion: apps/v1
kind: Deployment
…
spec:
…
 template:
…
 spec:
  containers:
…
     env:
     - name: PASSWORD
       valueFrom:
         secretKeyRef:
           name: mysecret
           key: password
```

Paramétrer les Pods

Secrets et **configMap**

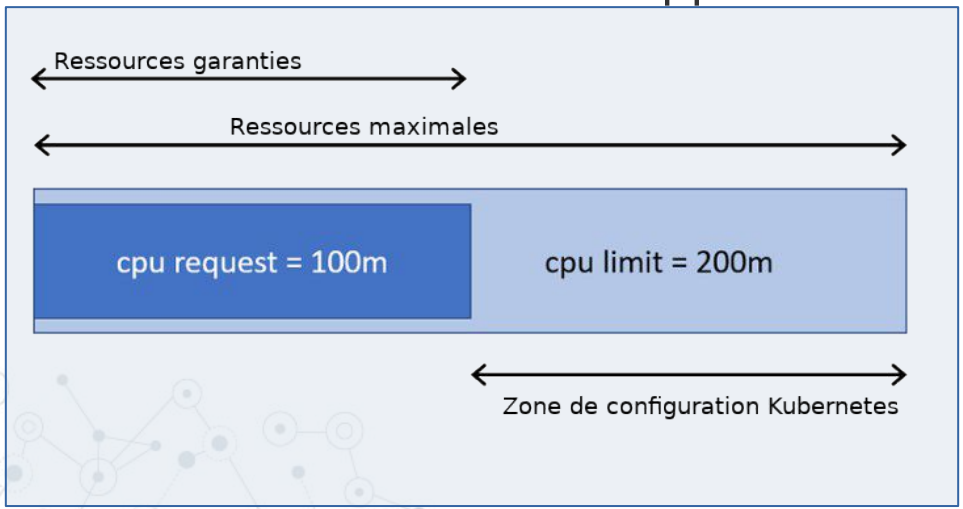
```
apiVersion:v1
kind: ConfigMap
metadata:
  name: myconfig1
data:
  database_url: 10.20.18.63
```

```
apiVersion:v1
kind: ConfigMap
metadata:
  name: myconfig2
data:
  config.properties:  |
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
```

```
apiVersion: apps/v1
kind: Deployment
…
spec:
…
  template:
…
  spec:
    containers:
      envFrom:
      -  secretRef:
           name: mysecret
      env:
      - name: PMA_HOST
        valueFrom:
          configMapKeyRef:
            name: myconfig1
            key: databas_url
      volumeMounts:
      - name: data-config
        mountPath: /config
    volumes:
    - name: data-config
      configMap:
        name: myconfig2
        items:
        - key: config.properties
          path: config.properties
```

Limiter les ressources

- Limitation par container
- En production, il est nécessaire de contrôler la consommation de ressource d'une application




```
apiVersion: extensions/v1beta1
kind: Deployment
...
  Containers:
  - name:
    resources:
      limits:
        cpu: 200m
        memory: 512Mi
      requests:
        cpu: 100m
        memory: 512Mi
```

200m CPU= 0.2 CPU, 20% d'un CPU

512Mi = 512Mo

HealthCheck

- Fournir à l’application une façon d’indiquer qu’elle est disponible, c’est-à-dire :
  - qu’elle est démarrée (liveness)
  - qu’elle peut répondre aux requêtes (readiness)
    - 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).

HealthCheck

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

```
apiVersion: extensions/v1beta1
kind: Deployment
...
  Containers:
  - name:
    livenessProbe:
      initialDelaySeconds: 2
      periodSeconds: 5
      httpGet:
        path: /health
        port: 9876
```

→ Kubernetes commencera à vérifier le path **/health** toutes les **5 secondes** après avoir attendu **2 secondes** pour le premier contrôle.

nodeSelector

- Le **nodeSelector** définit de façon très simple l'assignation du pod à un node possédant un label ou un tag spécifique

```
apiVersion: apps/v1
kind: Deployment
….
spec:
  …..
    spec:
      nodeSelector:
        datacenter: alpha
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

Affinité et anti-affinité

- Les règles d'affinité permettent de définir le scheduling pour un groupe de pod. Elles permettent de placer des pods sur le même node ou, au contraire, de les forcer sur des nodes différents.

Affinité et anti-affinité

```
apiVersion: apps/v1
kind: Deployment
….
spec:
  …..
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: datacenter
                operator: In
                values:
                - alpha
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

```
apiVersion: apps/v1
kind: Deployment
….
spec:
  …..
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: app
                operator: In
                values:
                - alpha
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

Gestion avancée des conteneurs

Dockerfile

- Les **Dockerfile**s sont des fichiers qui permettent de construire une image Docker adaptée à nos besoins, étape par étape
- Généralement, le fichier s'appelle Dockerfile et est placé à la racine du projet.

Dockerfile

Dockerfile:

```
FROM alpine
RUN apk update && apk add nodejs
COPY . /app
WORKDIR /app
CMD ["node","index.js"]
```

Dockerfile

- **FROM**: l'image de base à partir de laquelle est construite l'image actuelle.
- **RUN**: lance une commande shell (installation, configuration)
- **ADD**: ajoute des fichiers depuis le contexte de build à l'intérieur du conteneur
- **COPY**: copie des fichiers/dossiers dans l'image
- **ENV**: définit une variable d'environnement
- **EXPOSE**: définit les ports (UDP/TCP) écoutés au sein du container
- **USER**: utilisateur au sein du process
- **WORKDIR**: répertoire courant du process

Dockerfile

- **VOLUME**: espace de données persistent partagé entre l'hôte et le conteneur
- **CMD**: définit la commande par défaut lancée à la création d'une instance du conteneur avec ***docker run***. on l'utilise avec une liste de paramètres
  - ***CMD ["executable","param1","param2"]***
- **ENTRYPOINT**: précise le programme de base avec lequel sera lancé la commande
  - ***ENTRYPOINT ["executable"]***

Dockerfile

À ne pas confondre **RUN** et **CMD**

- **RUN**: exécute des commande et créer des commit et nouvelles images

- **CMD**: n'exécute rien au moment du build

(ça définit dans l'image la commande à exécuter par un container)

ENTRYPOINT vs. **CMD**

Un Dockerfile doit spécifier au moins un des deux :

- **ENTRYPOINT** doit être utilisé quand on utilise un container 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 container

- CMD sera surchargé en lançant un container avec des arguments

Dockerfile

- **ONBUILD:** permet d'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:
  - 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.
  - Au build de l'image de base des triggers sont ajoutés, mais les instructions ne sont pas exécutés.

```
ONBUILD ADD . /app/src
ONBUILD RUN /usr/local/bin/python-build \ --dir /app/src
```

Dockerfile

- **ONBUILD:**
  - Quand l’image de base est utilisée (FROM) pour un nouveau build: le builder vérifie s’il y a des trigger enregistrés et si oui, ils sont exécutés.
  - Les trigger sont nettoyés de l’image résultante : donc les trigger ne sont pas hérités par les “arrières petits-enfants”.

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 (&& \)
- Limiter la taille !

Création d'une image

- Construire une image depuis un Dockerfile:
  - **docker build -t <target> <rep>**

- Exemple:
  - **docker build -t easylinux/apache:2.5 .**

Création d’un container

- Lancer un container à partie de l’image crée :
  - **docker run -d easylinux/apache:2.5**

Travaux pratiques

- Création et automatisation d'images personnalisées:
  - Mise en œuvre d’un registre privé
  - Créer un conteneur Flask

Travaux pratiques

- `microk8s enable registry`
  - Le registry microk8s est sur localhost:32000
- Il suffit de tag l'image au build:
  - `docker build -t localhost:32000/myimage:registry .`
- Puis push :
  - `docker push localhost:32000/myimage:registry`

liste des images sur le registry microk8s: http://localhost:32000/v2/_catalog

Utiliser un registry privé dans Kubernetes

- Créer une secret en ligne de commande:

```
kubectl create secret docker-registry regcred --docker-server=<your-registry-server>
--docker-username=<your-name> --docker-password=<your-pword> --docker-email=<your-email>
```

- Créer un pod qui utilise cette secret

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: <private_image>
  imagePullSecrets:
  - name: regcred
```

Kubernetes en production

Frontal Ingress

- Dans Kubernetes, il existe trois approches générales 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

Frontal Ingress

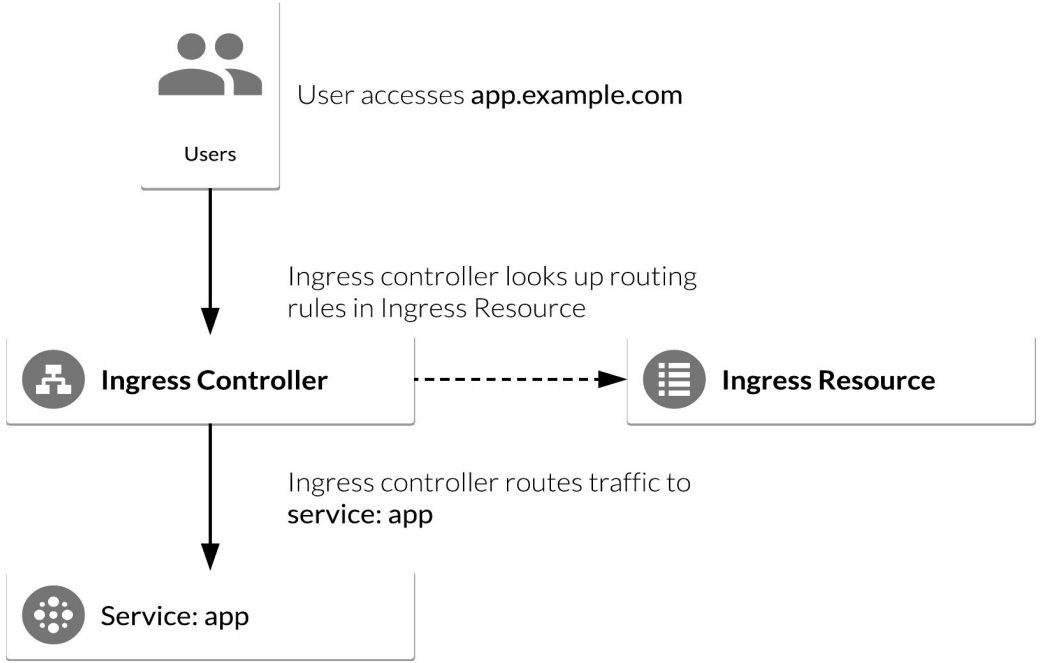
- Pour pouvoir créer des objets **ingress** il est d'abord nécessaire d'installer un **ingress controller** dans le cluster:
  - Il s'agit d'un déploiement conteneurisé d'un logiciel de reverse proxy (comme nginx) et intégré avec l'API de kubernetes
  - Le controlleur agit donc au niveau du protocole HTTP et doit lui-même être exposé (port 80 et 443) à l'extérieur, généralement via un service de type LoadBalancer.
  - Le controlleur redirige ensuite vers différents services (généralement configurés en ClusterIP) qui à leur tour redirigent vers différents ports sur les pods selon l'URL del a requête.

Frontal Ingress

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-wildcard-host
spec:
  rules:
  - host: "domain1.toto.com"
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: service1
            port:
              number: 80
      - pathType: Prefix
        path: "/toto"
        backend:
          service:
            name: service2
            port:
              number: 80
  - host: "domain2.foo.com"
    http:
      paths:
      …..
```

Travaux pratiques

- Mise en œuvre du frontal Ingress



Les namespaces et les quotas

- Pour gérer efficacement Kubernetes, il est conseillé de créer des **namespaces** par usage, puis de donner des *accès* et des *contraintes d'utilisation* par *namespace*.

```
apiVersion: v1
kind: Namespace
metadata:
  name: <nom_namespace>
```

Les namespaces et les quotas

- Limiter les ressources: **Quota**
- *Utilisation :*
  - Des équipes différentes travaillent dans des namespaces différents.
  - L'administrateur crée un quota de ressources pour chaque espace de nommage.
  - Les utilisateurs créent des ressources (pods, services, etc.) dans le namespace, 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.

Les namespaces et les quotas

- 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 aurait été violée.
  - Si le quota est activé dans un namespace 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.

Les namespaces et les quotas

- 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
```

Les namespaces et les quotas

- Limiter les ressources: **Quota**

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

- Dans un cluster d'une capacité de 32 GiBRAM, et 16 cœurs,
  - l'équipe A utilise 20 GiB et 10 cœurs,
  - B utilise 10GiB et 4 cœurs,
  - on garde 2GiB 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. Laisser l'espace de nommage "production" utiliser n'importe quelle quantité.

Les namespaces et les quotas

- Limiter les ressources: **Quota**

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.

Les changements de quotas n'affectent les ressources déjà créées

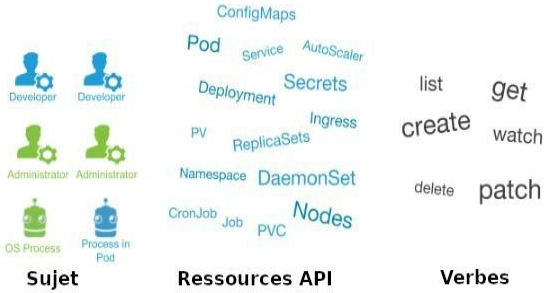
Gestion des ressources et autoscaling

- L'Autoscaler 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'Autoscaler Horizontal Pod 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 deployment monApp --min=2 --max=5 --cpu-percent=80

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, entre autres.
  - **Verbes** : L'ensemble des opérations qui peuvent être exécutées sur les ressources ci-dessus. Différents verbes sont disponibles (exemples : get, watch, create, delete, etc.), mais en fin de compte tous sont des opérations Create, Read, Update or Delete (CRUD).

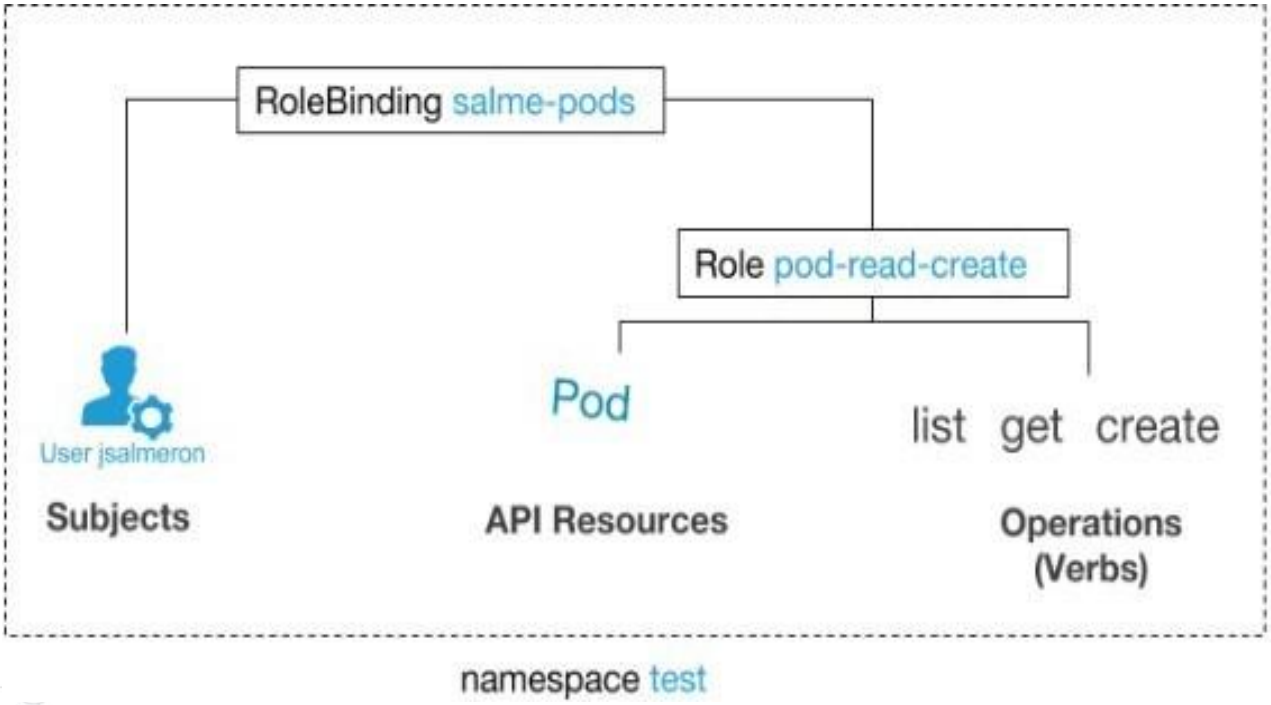


Gestion des accès

- Pour connecter ces trois types d'entités, on utilise les différents objets API RBAC (Le **R**ole-**B**ased **A**ccess **C**ontrol) disponibles dans Kubernetes.
  - **Rôles** : Permet de connecter les ressources API et les verbes. Ceux-ci peuvent être réutilisés pour différents sujets. Ils sont liés à un *namespace* (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 *namespaces* différents).
  - **ClusteRole** : comme le rôle mais pour le cluster entier
  - **RoleBinding** : Connectera 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. Pour l'équivalent de niveau cluster, sans espace de nom, il y a ClusterRoleBindings.

    ClusterRoleBindog : comme le roleBinding mais pour le cluster entier

Gestion des accès



Gestion des accès

- Supposons que nous voulons un utilisateur jdoe qui pourra lancer :

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

- Mais se verra refuser :

`kubectl get pods --namespace kube-system`

`kubectl get pod --namespace test -w` (nécessite watch)

Gestion des accès

```
apiVersion: rbac.authorization.k8s.io/v1
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: “”
roleRef:
  kind: Role
  name pod-read-create
  apiGroup: “”
```

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**

Gestion des accès

les utilisateurs n'ont pas d'objet API Kubernetes associé !

Dès lors :

- `kubectl create serviceaccount test-service`: fonctionne

Mais :

- `kubectl create user jdoe`: échouera

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.

Haute disponibilité

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

Haute disponibilité

- Pour les deux méthodes, vous avez besoin de :
  - ✔ au moins 2 machines pour les maîtres.
  - ✔ au moins 2 machines 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.
  - ✔ microk8s installés sur toutes les machines.
- Pour cluster externe etcd uniquement, vous avez également besoin de :
  - ✔ 2 machines supplémentaires pour les membres etcd

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 copie du pod sera lancée 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.

Mode maintenance

- 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`

    Ceci empêche les nouveaux pods d'être démarrés 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 est lié à 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 s'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

- Création d’un namespace
- Ajout d’un utilisateur à ce namespace
- Gérer les droits d’accès de l’utilisateur

HELM: Package Manager pour Kubernetes

- Besoin de quelque chose de plus puissant:
  - Pour s'adapter à plein de paramétrages différents de notre application
  - Pour éviter la répétition de code
- Helm permet donc de déployer des applications / stacks complètes en utilisant un système de templating et de dépendances, ce qui permet d'éviter la duplication et d'avoir ainsi une arborescence cohérente pour nos fichiers de configuration.
- Helm propose également :
  - la possibilité de mettre les Charts dans un répertoire distant (Git, disque local ou partagé…), et donc de distribuer ces Charts publiquement.
  - un système facilitant les Updates et Rollbacks de vos applications.

HELM: Package Manager pour Kubernetes

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

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

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

pod.yaml

template yaml config

HELM: Package Manager pour Kubernetes

- Les **values** sont définies dans les fichiers yaml ou avec le flag **–set** en ligne de commande

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

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

template yaml config

Values.yaml

HELM: Package Manager pour Kubernetes

- Structure des charts HELM:

MonChart/

- Chart.yaml → info sur le chart
- values.yaml → valeurs pour les fichiers templates
- charts/ → contient les dépendances charts
- templates/ → où les fichiers templates sont stockés
- …

HELM: Package Manager pour Kubernetes

- Créer un chart sur microk8s : `microk8s helm create <nom_chart>`
- Commandes HELM:

Pour installer une release
- `microk8s helm install <nom_release> <nom_chart>`
- `microk8s helm install -- set version:2.1.0 <nom_chart>`
- `microk8s helm install -- values=mes-valeurs.yaml <nom_chart>`

Pour mettre à jour une release
- `microk8s helm upgrade <nom_release> <nom_chart>`

Pour désinstaller une release
- `microk8s helm uninstall <nom_release>`

Déploiement d'un cluster Kubernetes

Mise en place du cluster

- 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
  - Chaque VM Multipass va tourner sur un Ubuntu 18.04 LTS ou Ubuntu 20.04 LTS
  - Installer Docker (optionnel) car Microk8s vient est Containerd (qui est un conteneur runtime)

Mise en place du cluster

- État initial:
  - Afficher les nœuds du cluster

```
sboubaker@sboubaker ~/dev/formation $ microk8s kubectl get no -o wide
NAME        STATUS   ROLES    AGE     VERSION   INTERNAL-IP     EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION       CONTAINER-RUNTIME
sboubaker   Ready    <none>   3d23h   v1.25.2   192.168.1.24    <none>        Ubuntu 20.04.4 LTS   5.15.0-48-generic    containerd://1.6.6
```

  - Vérifier les information du cluster

```
sboubaker@sboubaker ~/dev/formation $ microk8s.kubectl cluster-info
Kubernetes control plane is running at https://127.0.0.1:16443
```

  - Vérifier le status de microk8s

```
sboubaker@sboubaker ~/dev/formation $ microk8s status
microk8s is running
high-availability: no
  datastore master nodes: 127.0.0.1:19001
  datastore standby nodes: none
```

Mise en place du cluster

- Installer multipass:
  - `sudo snap install multipass`
- Créer 2 nœuds worker:
  - `multipass launch --name worker1 -m 2G`
  - `multipass launch --name worker2 -m 2G`
- Se connecter aux 2 nœuds worker et installer microk8s:
  - `multipass shell worker1`

Mise en place du cluster

- Lancer la commande suivante depuis le master:
  - `microk8s add-node`

```
sboubaker@sboubaker ~/dev/formation $ microk8s add-node
From the node you wish to join to this cluster, run the following:
microk8s join 192.168.1.24:25000/799513b1d32d89646c2134c94dfd80d4/c93b8c25829e

Use the '--worker' flag to join a node as a worker not running the control plane, eg:
microk8s join 192.168.1.24:25000/799513b1d32d89646c2134c94dfd80d4/c93b8c25829e --worker

If the node you are adding is not reachable through the default interface you can use one of the following:
microk8s join 192.168.1.24:25000/799513b1d32d89646c2134c94dfd80d4/c93b8c25829e
microk8s join 172.17.0.1:25000/799513b1d32d89646c2134c94dfd80d4/c93b8c25829e
microk8s join 192.168.49.1:25000/799513b1d32d89646c2134c94dfd80d4/c93b8c25829e
microk8s join 2001:861:3889:b480:1058:1376:a6be:23ea:25000/799513b1d32d89646c2134c94dfd80d4/c93b8c25829e
microk8s join 2001:861:3889:b480:1edc:4054:250b:fe1f:25000/799513b1d32d89646c2134c94dfd80d4/c93b8c25829e
```

Mise en place du cluster

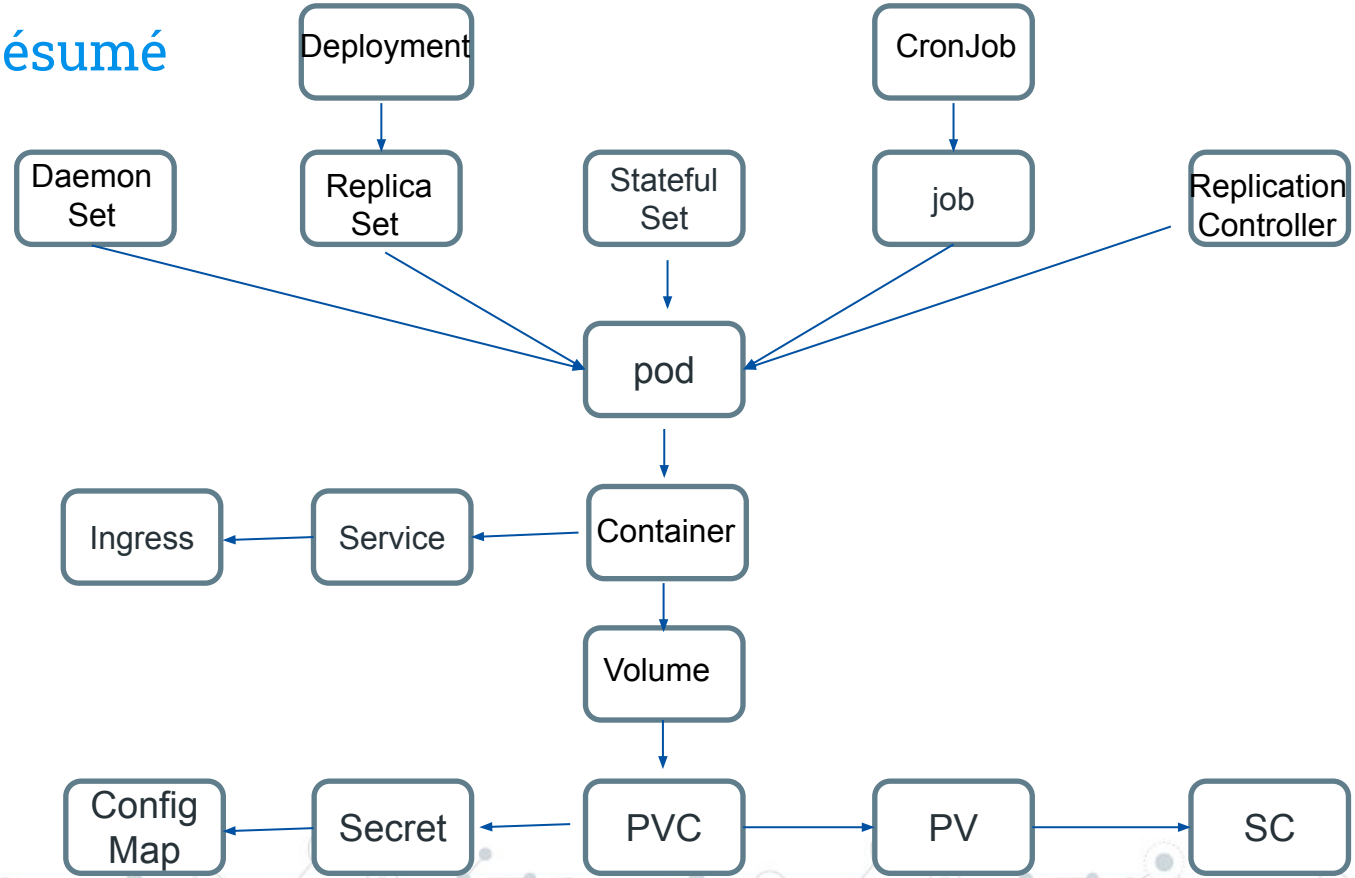
- Lancer la commande **join** depuis le worker:

```
microk8s join
192.168.1.24:25000/799513b1d32d89646c2134c94dfd80d4/c93b8c25829e
```

Travaux pratiques

- Mise en place cluster à 3 noeuds

Résumé


Deployment
CronJob
Daemon Set
Replica Set
Stateful Set
job
Replication Controller
pod
Ingress
Service
Container
Volume
Config Map
Secret
PVC
PV
SC