Inria

OVHcloud

kubernetes

Lyon 1

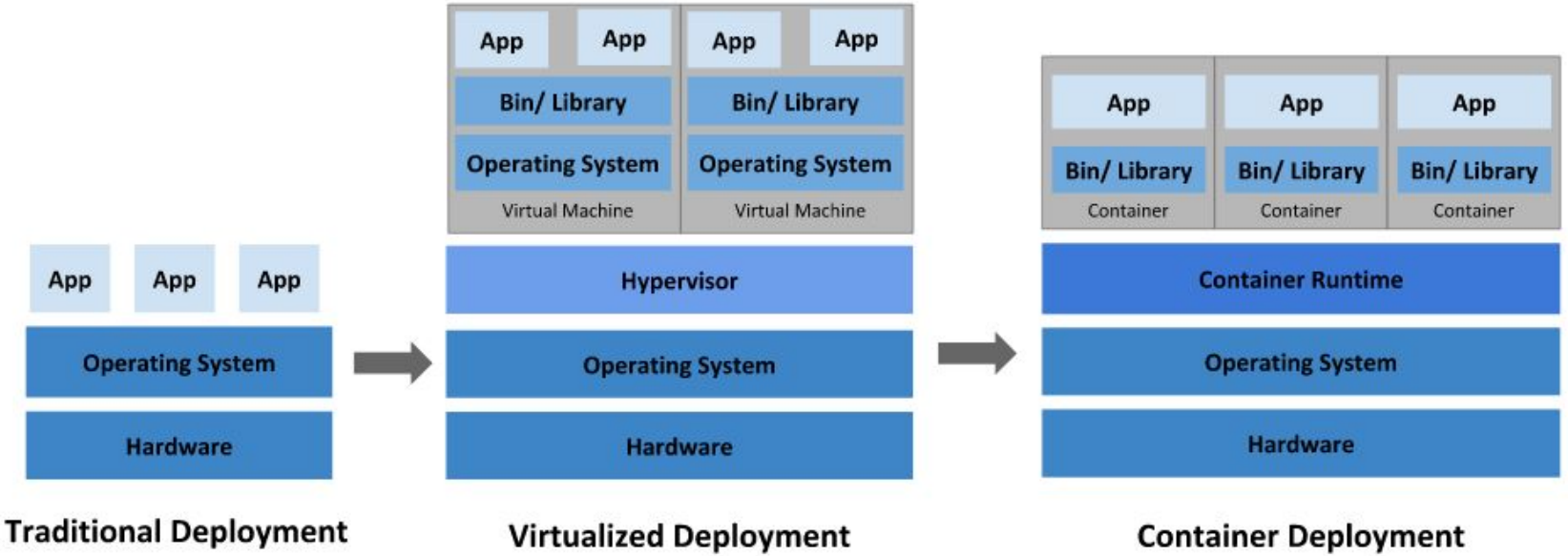
ENS DE LYON

Qui je suis?

- Doctorant
  - Inria - ENS de Lyon - Équipe Avalon
- Efficacité énergétique du cloud
  - Dans le cadre de défi Inria/OVHCloud
- Précédemment: Ingénieur Système &| Développeur open-source
- Page perso
  - https://vladost.com
- Mail
  - vladimir.ostapenco@univ-lyon1.fr

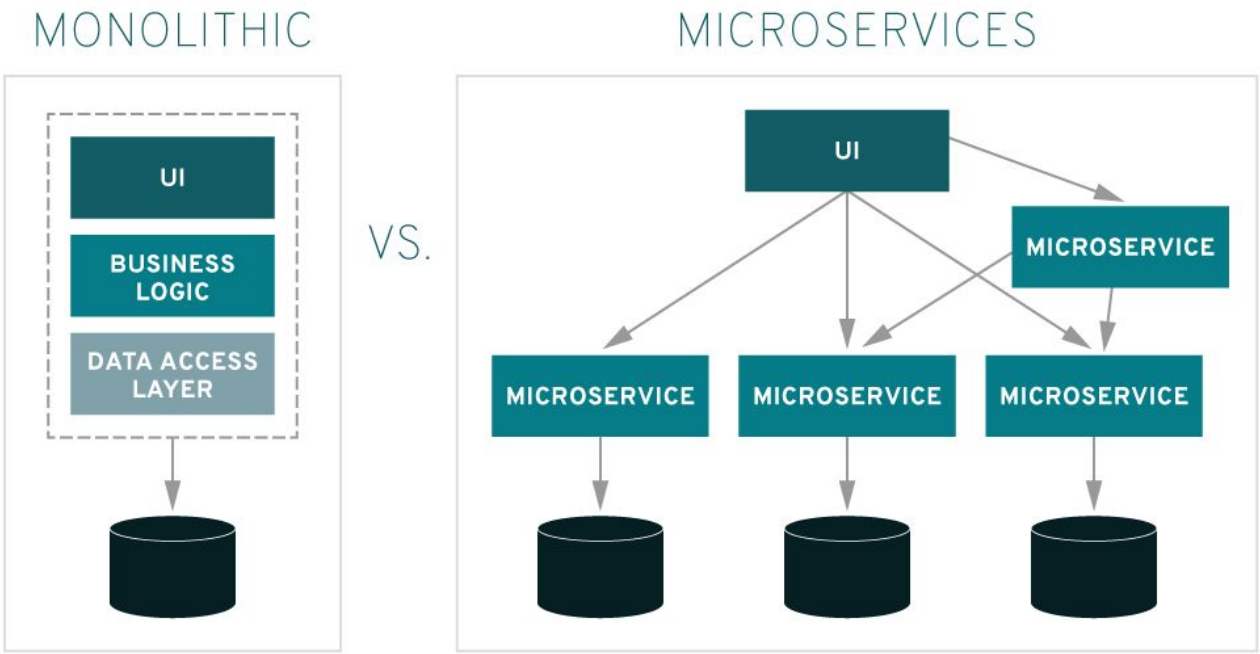
Vladimir Ostapenco

Infrastructure



Souce: kubernetes.io

Microservices



Souce: redhat.com

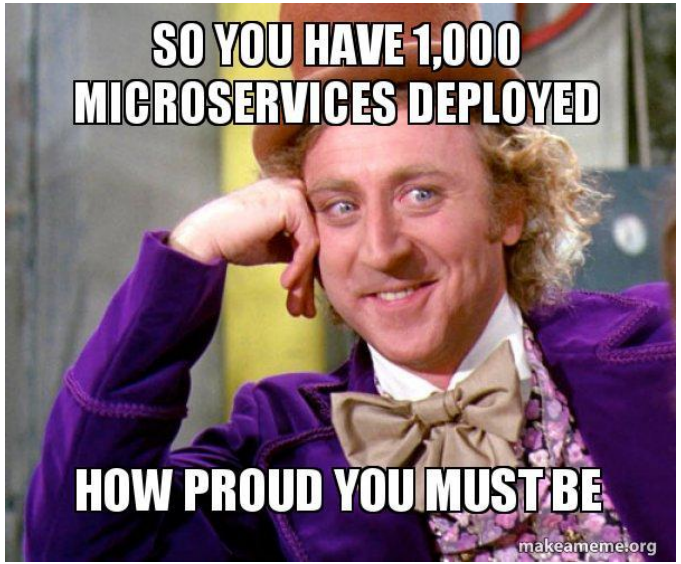
Microservices - Pourquoi?

- Création et maintenance des applications plus simples
- Productivité et vitesse améliorées
- Équipes autonomes et multifonctionnelles
- Évolutivité simple et mise à l'échelle dynamique
- Flexibilité dans l'utilisation des technologies

Microservices sans orchestration = enfer

«Sans un framework d'orchestration quelconque, vous avez juste des services qui s'exécutent “quelque part” où vous les configurez pour s'exécuter manuellement - et si vous perdez un nœud ou quelque chose tombe en panne, il faut le réparer manuellement. Avec un framework d'orchestration, vous déclarez à quoi vous voulez que votre environnement ressemble, et le framework le fait ressembler à cela.» ***Sean Suchter, co-fondateur et CTO de Pepperdata.***

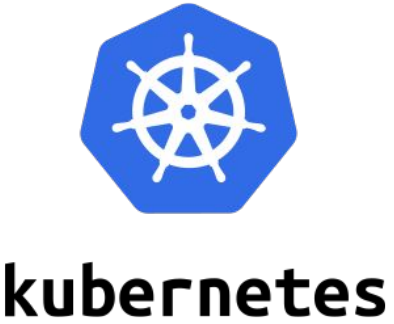
- Adidas (plusieurs releases par jour, 4000 conteneurs, 200 noeuds)
- Google (des milliards des conteneurs par semaine)



Kubernetes (K8s) vient à la rescousse !

Kubernetes (K8s) est un système open-source pour la gestion des charges de travail et des services conteneurisés, qui facilite à la fois la configuration déclarative et l'automatisation.

Kubernetes s’appuie sur 15 années d’expérience dans la gestion de charges de travail de production (workloads) chez Google, associé aux meilleures idées et pratiques de la communauté.



Souce: kubernetes.io

Kubernetes (K8s)

- Conçu pour le déploiement
- Est flexible (Run anywhere)
  - Environments hybrides sont possibles
- Est capable de gérer des millions des conteneurs
- Optimise l'utilisation des ressources
- Permet zero downtime
- Est DevOps friendly
- Est open-source
  - Soutenu par une communauté active (~ 3200 contributeurs sur GitHub)

Kubernetes (K8s)

- Kubernetes propose
  - Découverte de service et équilibrage de charge
  - Orchestration du stockage
  - Déploiements et rollbacks automatisés
  - Bin packing automatique
  - Self-healing
  - Gestion des secrets et des configurations

“Kubernetes is the new Linux OS of the Cloud.”

Souce: sumlogic.com

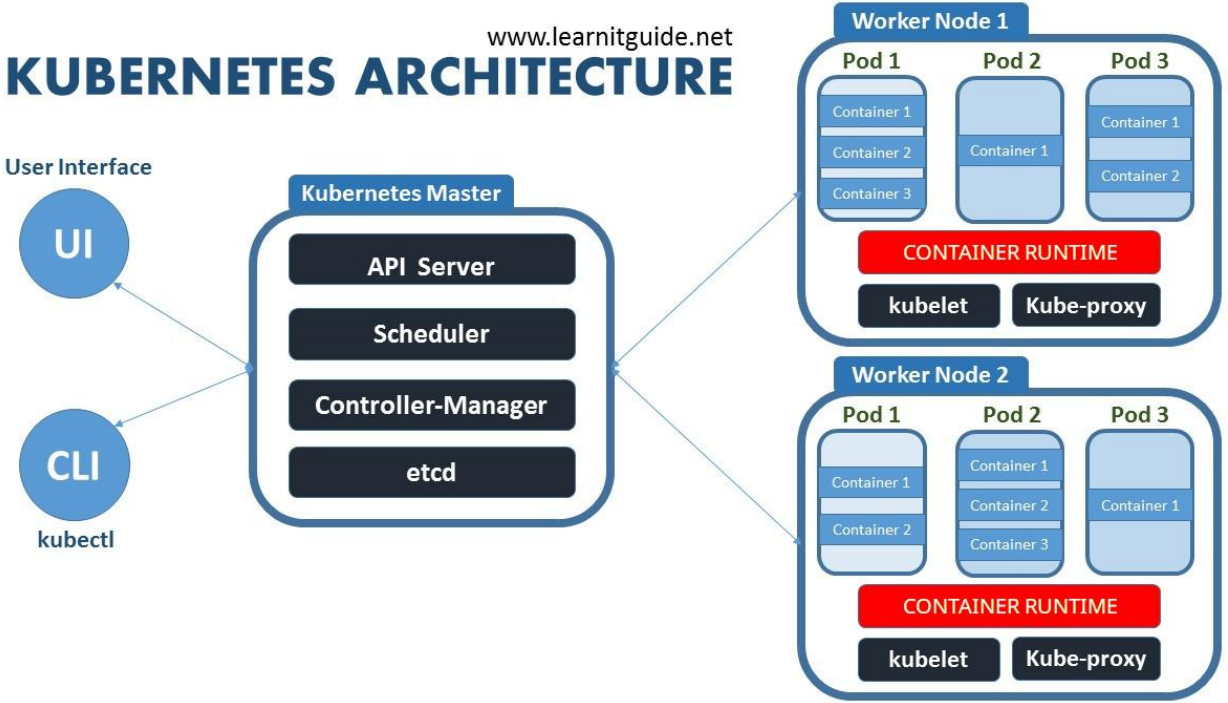
Kubernetes (K8s) - Why ?

- Pour le business
  - Délais de commercialisation plus rapides
  - Optimisation des coûts IT
  - Amélioration de l'évolutivité et de la disponibilité des services
  - Flexibilité multi cloud
  - Migration transparente vers le cloud
- Quelques stats
  - Gartner prédit qu'en 2022, plus de 75 % des organisations mondiales exécuteront des applications conteneurisées en production, contre moins de 30 % en 2020 [1]
  - RedHat dans leur étude montre que 70% des responsables informatiques travaillent pour des organisations utilisant Kubernetes [2]

1 - https://www.gartner.com/en/newsroom/press-releases/2020-06-25-gartner-forecasts-strong-revenue-growth-for-global-co
2 - https://www.redhat.com/en/resources/state-of-enterprise-open-source-report-2022

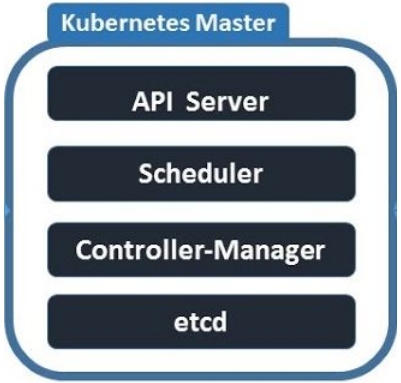
Kubernetes (K8s) - Sauve des entreprises!

Architecture



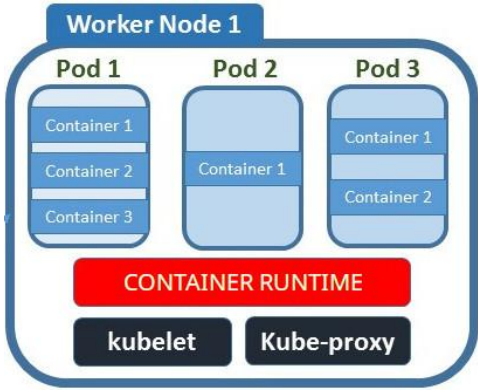
Architecture: Master Node (Control Plane)

- Composants du Control Plane
  - **API Server** - front end pour la Control Plane
    - Expose Kubernetes API
    - Hub de communication pour les composants K8S
  - **Etcd** - base de données clé/valeur pour toutes les données du cluster
  - **Scheduler** - assigne votre application à un Worker Node
    - Détecte les exigences de l'application et la place sur un nœud qui répond à ces exigences
  - **Controller manager** - maintient le cluster en état opérationnel, gère les pannes de nœuds, réplique les composants, maintient le bon nombre d'instances d'applications...
- Par défaut, ne lance aucun pod et il peut être répliqué pour une haute disponibilité
  - Options for Highly Available Topology



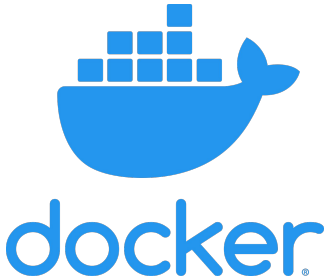
Architecture: Worker Node

- Responsable de la maintenance des pods en cours d'exécution et de la fourniture de l'environnement d'exécution Kubernetes
- Composants de Worker Node
  - **kubelet** - gère les conteneurs sur le nœud et communique avec l'API Kubernetes
  - **kube-proxy** (service proxy) - gère les règles réseau sur les nœuds
    - Ces règles permettent la communication avec vos conteneurs
  - **Container runtime** - exécute vos conteneurs (containerd, CRI-O, docker?..)
    - Tout environnement d'exécution qui implémente Kubernetes CRI



Kubernetes (K8s) - Abandon de Docker

- Kubernetes utilise CRI (Container Runtime Interface) comme l'API d'exécution des conteneurs
- Docker ne supporte pas CRI
  - **dockershim** a été utilisé par K8s comme un pont entre API docker et CRI
- Kubernetes n'utilise pas les fonctionnalités de networking et de volume livrés avec docker par défaut
  - Ces fonctionnalités inutilisées peuvent entraîner des risques de sécurité
- **Dockershim** à été supprimé dans la version v1.24
- Alternatives
  - **Containerd** (Fait partie et compatible avec Docker)
    - Container Runtime par défaut à partir de K8s v1.24
  - **CRI-O** (Approche plus minimaliste développée par RedHat et utilisé avec OpenShift)



Moving Forward

- Namespaces
- API Server et Objets API
- Pods
- Workloads
- Labels, selectors et Annotations
- Services
- Scheduling
- Networking
- Stockage
- initContainers
- Variables d'environnement
- Secrets et ConfigMaps
- Surveillance des applications
- Cloud controller
- Déployer un cluster Kubernetes

Namespaces

- Séparation virtuelle de différents environnements (Clusters virtuels)
- Moyen de diviser les ressources du cluster entre plusieurs utilisateurs
  - Séparation du cluster entre différentes équipes ou projets
- Ne sont pas applicables aux objets cluster-wide (e.g. StorageClass, Nodes, PersistentVolumes, etc.)
- Namespaces initiés à la création du cluster
  - **default** - namespace par défaut pour les objets sans autre namespace
  - **kube-system -** namespace pour les objets système Kubernetes
  - **kube-public** - cet namespace est créé automatiquement et est lisible par tous les utilisateurs (Utilisé par le cluster)
  - **kube-node-lease** - namespace dédié aux heartbeats des nœuds afin améliorer leur performances
- **NB**: Les noms des ressources doivent être uniques dans un namespace

API Server

- Composant principal du Control Plane qui expose l'API Kubernetes
- Hub de communication
  - Permet aux utilisateurs finaux et aux différentes parties de votre cluster de communiquer entre eux
- Chaque composant Kubernetes communique exclusivement via le API server
  - Ils ne se parlent pas directement
- Seul composant qui communique directement avec le **etcd**
- Permet d'interroger, manipuler et valider les objets API (les services, les contrôleurs de réplication, etc)
- **Kubectl -** l'outil CLI qui permet d'utiliser l'API server pour gérer les objets API
  - **kubectl get nodes** - afficher l'état des noeuds du cluster
- API server peut aussi être utilisé
  - À partir de vos applications via une lib Go ou Python
  - Directement via les appels REST

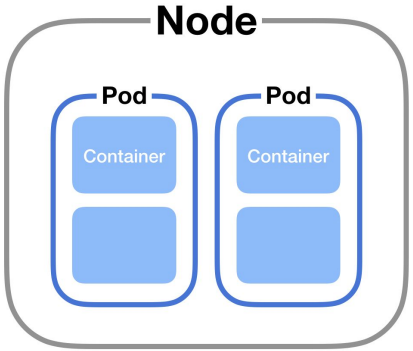
Objet API

- Tout sur la plateforme K8s est traité comme un objet API
  - Les services, les volumes...
- Peut être défini au format YAML
  - langage de sérialisation de données simple et lisible
- Une fois créé, K8s s'assure que l'objet existe et est fonctionnel
- Son **nom** doit être **unique** pour un type d'objet dans un **namespace**
- Doit contenir les champs suivants
  - apiVersion - version de l'API Kubernetes utilisé pour créer cet objet (v1alpha1, v3beta2, v1)
  - kind - type d'objet
  - metadata - données qui aident à identifier votre objet (nom, label, UID, namespace...)
  - spec - description de l'état de l'objet

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

Pods

- Est l’unité de base de Kubernetes
  - La plus petite unité que vous pouvez déployer dans le cluster
- Un pod représente les processus en cours d'exécution sur votre cluster
- Représente une unité de déploiement
  - Une instance unique d'une application, qui peut consister en un conteneur unique ou en un petit nombre de conteneurs étroitement couplés et partageant des ressources
- Encapsule
  - Conteneur d’une application (ou plusieurs conteneurs)
  - Ressources de stockage
  - Adresse IP réseau unique
  - Options d’exécution des conteneurs
- Normalement, vous ne manipulez pas le Pod directement



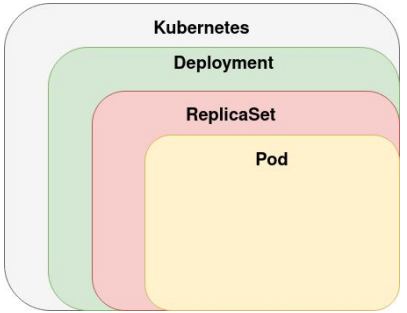
Workload ressources

- **Workload** - une application s'exécutant sur K8s (un ou plusieurs Pods)
- **Workload ressources** - des contrôleurs qui créent et gèrent les Pods pour vous
  - Assurent la réplication, les déploiements et self-healing automatiques
  - Permettent d'effectuer les **Rolling Updates** des Pods (avec mécanismes de **Rollback**)
- Workload ressources
  - **Deployments** et **ReplicaSets**
  - **StatefulSets**
  - **DaemonSets**
  - **Job** et **Cronjob**

Workloads: Deployment

- Workload ressource le plus utilisé qui permet
  - Décrire l'état souhaité pour votre application
  - Mises à jour déclaratives avec le mécanisme de Rollback
  - Mise à échelle (Scale up) pour supporter plus de charge
- Ressource de haut niveau englobant
  - **ReplicaSet**
    - Un ou plusieurs Pods répliqués
    - Garantit qu'un certain nombre des Pods est un cours d'exécution à tout moment
    - Doit toujours être crée par un Deployment
- Quelques commandes:
  - **`kubectl apply -f deployment.yaml`** - `créer un déploiement`
  - **`kubectl get deployments`** - `afficher les déploiements`
  - **`kubectl describe deployments`** - `afficher la description complète d'un 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:1.7.9
        ports:
        - containerPort: 80
```



Workloads: StatefulSet

- Workload ressource pour gérer les applications avec état (Stateful) qui
  - Fournit des garanties relatives à l'ordre et à l'unicité de ces pods
  - Contrairement à un Deployment, conserve une identité permanente pour chacun de ses pods
- Les pods créés ne sont pas interchangeables
- Si un pod est mort, il est remplacé par un autre avec le même nom d'hôte et les mêmes configurations
- Utiles pour les applications qui nécessitent un ou plusieurs des éléments suivants
  - Identifiants réseau stables et uniques
  - Stockage stable et persistant
  - Déploiement, mise à l'échelle et mises à jour ordonnés
- Quelques commandes
  - **`kubectl get statefulsets`**
  - **`kubectl describe statefulsets`**

Workloads: DaemonSet

- Garantit que tous (ou certains) nœuds exécutent une copie du Pod
- N’utilise pas l'ordonnanceur par défaut
- Lorsque des nœuds sont ajoutés au cluster, des pods d’un DaemonSet leur sont ajoutés automatiquement
- Exemples d'utilisation
  - Solution de supervision ou gestion des logs
  - Composants proxy et réseau de Kubernetes (**kube-proxy** et **kube-flannel**)
- Quelques commandes:
  - **`kubectl get daemonset`**
  - **`kubectl describe daemonset`**

Workloads: Job et Cronjob

- **Job** peut être vu comme une tâche
- Crée un ou plusieurs Pods qui réalisent une tâche et s'arrêtent
- Les Pods ne sont pas supprimés automatiquement après leur arrêt
- Permet d'exécuter plusieurs Pods en parallèle
- **Cronjob** permet de planifier l'exécution des Jobs
- Exemples d'utilisation
  - **Job**: Conversion d’une vidéo
  - **Cronjob**: Backup d’une base des données

```
apiVersion: batch/v1
kind: Job
metadata:
  name: blender
spec:
  template:
    spec:
      containers:
      - name: blender
        image: blender-render
        command: ["render", “./movie.zip”]
      restartPolicy: Never
  backoffLimit: 4
```

Labels et Label Selectors

- Labels
  - Paires clé / valeur attachées à des objets K8s (tels que Pods)
  - Utilisés pour identifier des objets et organiser des sous-ensembles d'objets
  - Exemple:
    - `"release" : "stable", "environment" : "production"`
  - Peuvent être attachés aux objets au moment de la création, puis ajoutés et modifiés à tout moment
  - Chaque clé doit être unique pour un objet donné
- Label Selectors
  - Permet de sélectionner un ensemble d'objets

```
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:1.7.9
        ports:
        - containerPort: 80
```

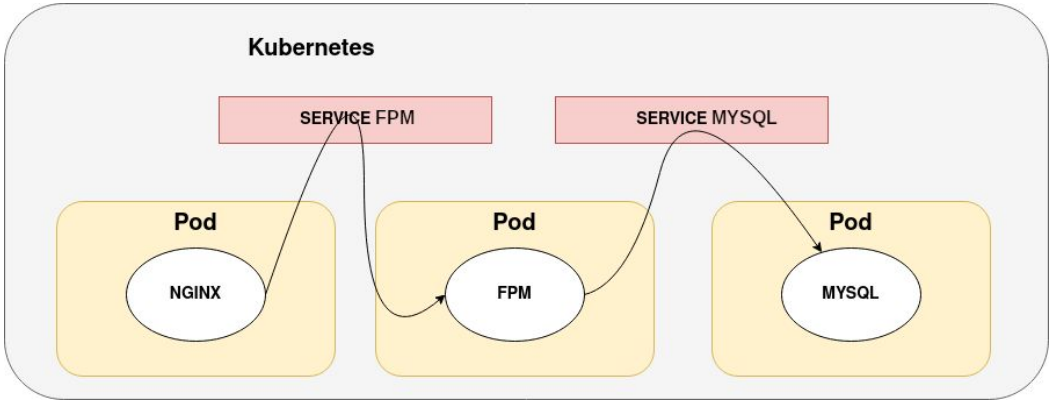
Annotations

- Permettent d’attacher des métadonnées à vos objets
- Ne peuvent pas être utilisés pour identifier et sélectionner des objets
- Peuvent être utilisés par des outils et des bibliothèques
- Exemples d'utilisation
  - Information sur la version de l’application, de l’image
  - Téléphone des personnes responsables
  - Repository avec le code source de l’application
  - Informations d'application utilisables par d'autres composants du système d'information

```
apiVersion: v1
kind: Pod
metadata:
  name: annotations-demo
  annotations:
    imageregistry: "https://hub.docker.com/"
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

Services

- Permet d’exposer vos Pods en tant qu’un service réseau
- Est une abstraction qui définit un ensemble logique de pods et une politique permettant d'y accéder
  - parfois appelée micro-service
  - a une adresse IP définie
  - décide de router le trafic vers l'un des pods (load balancing)
  - peut exposer plusieurs ports
- L'ensemble de Pods ciblés par un service est déterminé par un Label Selector
- Permet d’effectuer la résolution des applications entre les Pods



Services: Découverte

Principales méthodes de découverte d'un Service

- **Variables d'environnement**
  - **Kubelet** ajoute dans chaque Pod un ensemble de variables d'environnement pour chaque service actif
  - Exemple:
    - Pour le service *svcname*
      - Les variables *{SVCNAME}_SERVICE_HOST* et *{SVCNAME}_SERVICE_PORT* seront disponible dans chaque Pod
- **DNS (Un service DNS doit être configuré sur le cluster)**
  - Le serveur DNS surveille les nouveaux services et crée des enregistrements DNS
  - Exemple:
    - Pour le service *my-service* dans le namespace *my-ns* une entrée DNS *my-service.my-ns* sera créé
      - Pods dans le namespace *my-ns* peuvent trouver le service en utilisant le nom DNS *my-service* ou *my-service.my-ns*
      - Pods de l'autre namespace doivent utiliser *my-service.my-ns*

Services: Suite

- Types des services
  - **ClusterIP** (par défaut) - Expose le service sur une adresse IP interne au cluster (Idéal pour backend)
  - **NodePort** - Expose le service sur un port statique sur chaque nœud du cluster
  - **LoadBalancer** - Exposer le service à l'aide d'un load balancer externe (fournisseur de cloud)
  - **ExternalName** - Mappe le service à un nom DNS externe (champ *externalName*)
    - Requête DNS sur le nom de service renvoie CNAME avec la valeur du champ *externalName*
    - Utile lors de la migration progressive de vos services sur K8s

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
  - name: http
    protocol: TCP
    port: 80
    targetPort: 80
```

```
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:1.7.9
        ports:
        - containerPort: 80
```

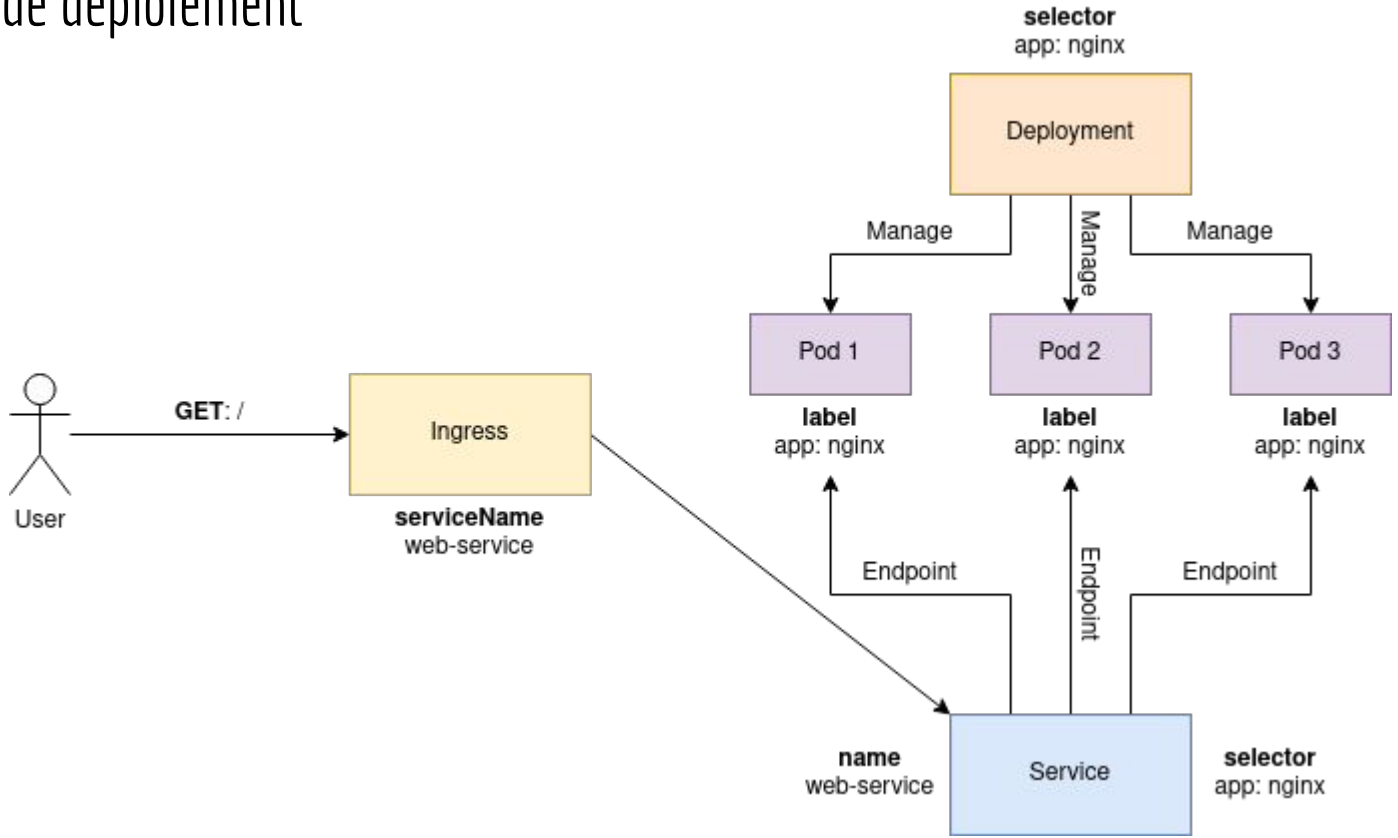
Ingress

- Gère l'accès externe (HTTP ou HTTPS) aux services
- Nécessite un **Ingress Controller**
- Peut fournir
  - un équilibrage de charge
  - une terminaison SSL
  - un hébergement virtuel basé sur le nom de domaine
- Types
  - **Single Service** - vous permet d'exposer un seul service
  - **Simple fanout** - achemine le trafic vers plusieurs services, en fonction de l'URI HTTP demandé
  - **Name based virtual hosting** - routage du trafic HTTP vers plusieurs noms d'hôte à la même adresse IP

fanout-ingress.yml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: simple-fanout-example
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        pathType: Prefix
        backend:
      service:
              name: service1
              port:
                number: 4200
      - path: /bar
        pathType: Prefix
        backend:
      service:
              name: service2
              port:
                number: 8080
```

Exemple de déploiement



Scheduling

- Chargé de trouver le meilleur nœud sur lequel placer votre Pod
- Utilise la fonction de *spread priority* qui permet de s'assurer que les Pods du même jeu de réplicas sont étalés sur des nœuds différents pour éviter les interruptions de service
- *kube-scheduler* sélectionne un nœud pour le Pod en quelques étapes :
  - **Filtering** - trouve l'ensemble de nœuds où il est possible de planifier le Pod
    - Par exemple, si le nœud dispose de suffisamment de ressources disponibles pour répondre aux demandes de ressources spécifiques d'un pod
  - **Scoring** - classe les nœuds restants pour choisir le placement de Pod le plus approprié
    - Attribue un score à chaque nœud qui a survécu au filtrage, en basant ce score sur les règles de notation
  - **kube-scheduler** attribue le Pod au Node avec le classement le plus élevé (aléatoire si plusieurs nœuds ont le même score)
- Vous pouvez créer votre propre scheduler!

Scheduling: Requests et Limits

- **Requests**
  - “Soft cap”
  - Utilisé seulement pendant la planification
- **Limits**
  - “Hard cap”
  - Runtime empêche le conteneur d'utiliser plus
- **Valeurs pour le CPU:** 1000m = 1.0 = 1 vCPU
- **Valeurs pour la mémoire:** un nombre suivi d'un suffixe comme M, K ou Mi, Ki etc.

```
containers:
  resources:
     requests:
       cpu: 1000m
       memory: 20Mi
```

```
containers:
  resources:
     limits:
       cpu: 1000m
       memory: 20Mi
```

Scheduling: Taints et Tolerations

- **Taint**
  - Permet à un nœud de repousser le travail
    - Par exemple, le nœud Master a un taint “*NoSchedule*”
      - Cela signifie que le scheduler ne lui donnera jamais du travail

```
taints:
node-role.kubernetes.io/master:NoSchedule
```

- **Toleration**
  - Permet à un Pod de tolérer un taint
    - Par exemple, si vous incluez la Toleration “*NoSchedule*” dans un Pod, il peut être planifié sur le nœud Master
      - C’est le cas du Pod ***kube-proxy***

```
tolerations:
- effect: NoSchedule
  key: .../unschedulable
  operator: Exists
```

- Ces mécanismes fonctionnent ensemble pour garantir que les Pods ne sont pas planifiés sur des nœuds inappropriés

Networking: Communication

- **Au sein du même nœud**
  - Un Pod voit un périphérique Ethernet normal ***eth0***
  - Un tunnel qui connecte le réseau du Pod avec le réseau du nœud est créé
    - Ce tunnel contient deux interfaces virtuelles
      - Dans le conteneur (***eth0***)
      - Sur le noeud (***vethX***)
  - **Linux Ethernet Bridge** permet la communication entre les pods
    - Contient toutes les interfaces ***vethX*** de tous les Pods exécutés sur le nœud

- **Entre les nœuds**
  - Container Network Interface (CNI)

Networking: Container Network Interface (CNI)

- Utilisé pour la communication entre les nœuds
- Installé via un plugin et n'est pas natif à la solution K8s
  - Installe un agent de réseau sur chaque noeud (***DaemonSet***)
- Une surcouche réseau qui permet de construire des tunnels entre les Pods
- Effectue le remplacement des adresses source et destination des paquets par le NAT
- Les plus utilisés
  - ***Flannel***
  - ***Calico***
  - ***Romana***
  - ***Weave***

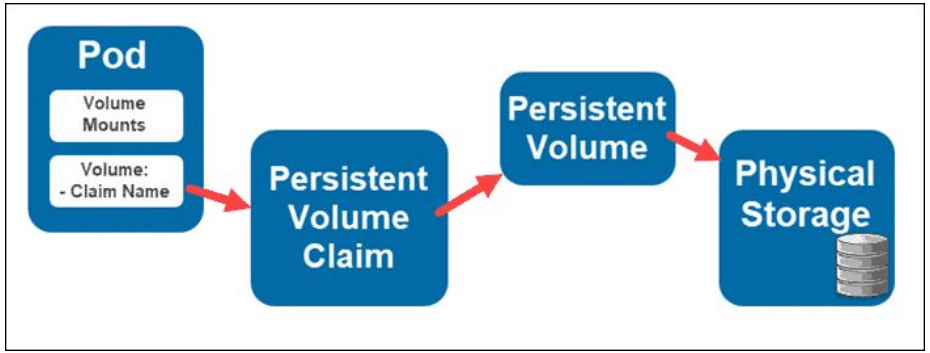
Stockage: Volumes

- Les volumes K8s peuvent être
  - **Éphémères** - ont une durée de vie d'un Pod
  - **Persistantes** - existent au-delà de la durée de vie d'un Pod
- Types des volumes
  - **emptyDir** - un volume initialement vide, créé pour lorsqu'un pod est attribué à un nœud et existe tant que ce pod s'exécute sur ce nœud
  - **hostPath** - monte un fichier ou un répertoire du système de fichiers du nœud dans votre pod
  - **Cephfs, fc, nfs, iscsi** - permet à un volume existant d'être monté dans votre pod
  - **configMap, secret** - un moyen d'injecter des données de configuration ou des secrets dans des pods
  - **persistentVolumeClaim** - utilisé pour monter un volume persistant dans un Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: busybox
      name: test-container
      volumeMounts:
      - mountPath: /cache
      name: cache-volume
  volumes:
  - name: cache-volume
      emptyDir: {}
```

Stockage: Persistent Volumes (PV)

- Sous-système qui sépare la partie fourniture de stockage de la partie consommation de stockage
- Abstraction permettant de découpler les volumes mis à disposition par les administrateurs et les demandes d'espace de stockage des développeurs pour leurs applications
- Est composé de deux éléments
    - **PersistentVolume** - configuré par un administrateur ou provisionné dynamiquement par un Storage Class
    - **PersistentVolumeClaim** - peut être vu comme une demande de stockage et peut être attaché à un Pod



Source: phoenixnap.com

Stockage: Persistent Volumes (PV)

- Configuré par un administrateur (static) ou provisionné dynamiquement par un Storage Class (dynamic)
- A un cycle de vie indépendant de tout Pod individuel qui l'utilise
- Capture les détails de l'implémentation du stockage, qu'il s'agisse de NFS, d'iSCSI ou d'un système de stockage spécifique au fournisseur de cloud
- Types des PVs (implémentés sous forme de plugins)
    - Cephfs, csi, fc, hostPath, iscsi, **local**, nfs, rbd
- **`kubectl get pv`**

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 200Mi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

Stockage: Modes d’accès

- Permettent de spécifier le mode d’accès supporté par un volume persistant
- En le spécifiant, vous autorisez le volume à être monté sur un ou plusieurs nœud, ainsi que lu par un ou plusieurs nœud
- Modes d'accès
  - **ReadWriteOnce** - seul un nœud peut monter le volume en écriture et en lecture
  - **ReadOnlyMany** - plusieurs nœuds peuvent monter le volume en lecture
  - **ReadWriteMany** - plusieurs nœuds peuvent monter le volume en lecture et en écriture
- C'est une capacité de montage de nœud et non de Pod
- Un volume ne peut être monté qu’en utilisant un mode d’accès à la fois, même s’il en supporte plusieurs

Stockage: Politiques de rétention

- Indique au cluster quoi faire avec le volume après qu'il a été libéré de sa claim
- persistentVolumeReclaimPolicy
  - **Retain** - gestion manuelle. Vous ne perdez aucune donnée lors de la suppression de claim.
  - **Recycle** - le contenu sera supprimé pour être utilisé pour les nouveaux claims (*rm -rf /thevolume/**)
  - **Delete** - le stockage sous-jacent sera supprimé (GCP volume, Openstack Cinder volume)

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  persistentVolumeReclaimPolicy: Recycle
  capacity:
    storage: 200Mi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

Stockage: Persistent Volume Claims (PVC)

- Permettent aux développeurs d'applications de demander du stockage pour l'application sans avoir à savoir où se trouve le stockage sous-jacent
- Sont liées aux Volumes Persistants
  - Un Volume Persistant ne sera pas libéré tant que PVC liée existe
- Reste liée au Volume Persistant même si le pod a été supprimé
- **`kubectl get pvc`**

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: task-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 200Mi
```

```
  volumes:
  - name: mongodb-data
      persistentVolumeClaim:
        claimName: task-pv-claim
```

Stockage: StorageClass

- Permet de provisionner automatiquement le stockage afin qu'il n'y ait aucune nécessité de créer un PersistentVolume
- Il suffit de dire à K8s ce qu'est le provisionner et il créera le volume pour vous
- Provisioning dynamique

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: gold
provisioner: kubernetes.io/cinder
parameters:
  availability: nova
```

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongodb-pvc
spec:
  storageClassName: gold
  resources:
      requests:
      storage: 100Mi
  accessModes:
      - ReadWriteOnce
```

Conteneurs d’initialisation (initContainers)

- Utiles lorsque vous souhaitez initialiser un **Pod** avant l'exécution du conteneur d'application
- Peuvent être utilisés pour
  - Télécharger du code
  - Effectuer une configuration
  - Initialiser une base de données
- Un Pod peut avoir un ou plusieurs conteneurs d'initialisation
- Doivent finir avec succès avant que le conteneur principal ne démarre

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod-with-init
spec:
  volumes:
  - name: www
    emptyDir: {}
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
    - mountPath: /www
      name: www
  initContainers:
  - name: init-nginx
    image: busybox
    command: ["init", "/www"]
    volumeMounts:
    - mountPath: /www
      name: www
```

Variables d’environnement

- Moyen le plus simple d'injecter des données dans vos applications
- Peuvent être définis
  - Pour les conteneurs qui s'exécutent dans le Pod
  - En incluant le champ *env* ou *envFrom* dans le fichier de description de l'objet
- Peuvent contenir des secrets (*Secrets*) et des configurations (*ConfigMaps*)
- Peuvent être interdépendants
  - Vous pouvez utiliser *$(VAR_NAME)* dans le champ *value* de *env*

```
apiVersion: v1
kind: Pod
metadata:
  name: print-greetings
spec:
  containers:
  - name: env-print-demo
      image: busybox
      env:
      - name: GREET
        value: "Greetings! $(NAME)"
      - name: NAME
        value: "Kubernetes"
      command: ["echo"]
      args: ["$(GREET)"]
```

Secrets

- Sont utilisés pour sécuriser les données sensibles auxquelles vous pouvez accéder depuis votre pod
- Sont créés indépendamment des pods
- Peuvent être fournis à vos Prods sous forme de variables d'environnement ou de volumes
- Utilisation sous forme des variables d'environnement est une mauvaise pratique
  - Certaines applications peuvent écrire toutes les valeurs des variables d'environnement dans les logs
- Utilisation sous forme de volumes est préférable
  - sont stockés dans ***tmpfs*** et ne sont jamais écrits sur le disque
  - sont mises à jour automatiquement
- Instanciables via un fichier YAML (*encodage base64*) ou directement avec ***kubectl***

```
apiVersion: v1
kind: Secret
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
metadata:
  name: test-secret
```

```
volumes:
  - name: secret-volume
    secret:
      secretName: test-secret
```

```
volumeMounts:
  - name: secret-volume
    mountPath: /secret
    readOnly: true
```

ConfigMaps

- Permettent de stocker des données de configuration non confidentielles dans des paires clé-valeur
- Peuvent être fournis à vos Prods sous forme
  - Variables d'environnement
  - Fichiers de configuration dans un volume
- Sont mises à jour automatiquement (si utilisés sous forme de volumes)

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: game-demo
data:
    game.properties: |
      enemy.types=aliens,monsters
      player.maximum-lives=5
```

```
volumeMounts:
  - name: config
      mountPath: "/config"
      readOnly: true
```

```
volumes:
  - name: config
    configMap:
      name: game-demo
      items:
        - key: "game.properties"
          path: "game.properties"
```

Configuration sera disponible dans /config/game.properties

Surveillance des applications

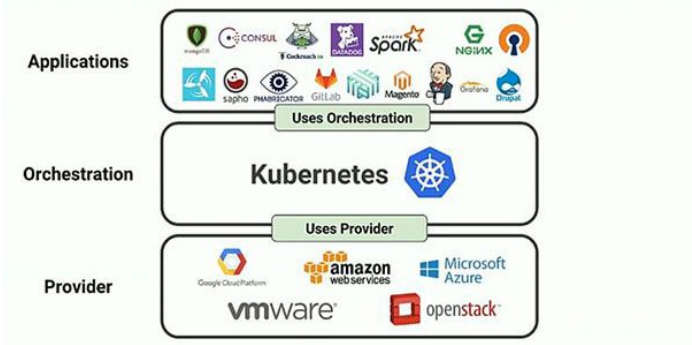
- K8s peut surveiller non seulement l'exécution des vos Pods mais aussi le fonctionnement de vos applications avec les mécanismes de
  - Réparation des vos applications via le redémarrage ou la recréation
  - Rétention du trafic entrant jusqu'au démarrage complet de votre application
- Pour utiliser ces mécanismes vous devez ajouter des sondes (probes) dans vos Pods

Sondes Readiness et Liveness

- **Sonde Liveness** - permet de vérifier si votre application fonctionne correctement
  - **HTTP Get probe** - effectue une requête HTTP GET sur un port et un path. Si une réponse reçue -> OK, sinon redémarrer le conteneur
  - **TCP Socket probe** - tente d'ouvrir une connexion TCP sur un port spécifique. Le comportement est similaire au *HTTP Get probe*
  - **Exec probe** - exécute une commande arbitraire dans le conteneur. Code de sortie 0 -> OK, sinon redémarrer le conteneur
- **Sonde Readiness** - permet de vérifier si le conteneur est capable de recevoir les demandes des clients
  - Si la vérification échoue, le conteneur n'est pas redémarré et le trafic n'est pas redirigé vers ce conteneur
- **Sonde Startup** - utile pour les applications legacy avec un temps de démarrage long
  - Attend le temps de démarrage le plus défavorable et vérifie si le conteneur est disponible
  - Si la vérification échoue, le conteneur est redémarré

Cloud Controller

- Permet de lier votre cluster K8s à l'API de votre fournisseur de cloud
  - Openstack, AWS, Google Cloud...
- Permet d'intégrer les fonctionnalités et des services du cloud provider dans votre cluster
  - Gestion automatisée des noeuds
  - Création et gestion des volumes
  - Creation et configuration des load balancers
  - Utilisation de l’adressage IP et du network filtering
  - Délégation de l’authentification



Source: https://www.fairbanks.nl/

Déployer un cluster Kubernetes

- **Kubeadm** - outil **officiel** pour créer et gérer des clusters Kubernetes
- **Kubespray** - outil d'automatisation du déploiement des clusters Kubernetes avec **Ansible**
- **kOps** - déploiement des clusters Kubernetes sur AWS
- **RKE** (Rancher Kubernetes Engine) - outil qui facilite et automatise le déploiement, les mises à jour et les Roll backs des clusters Kubernetes
- Pendant les TPs, nous allons utiliser le **RKE**

Rancher Kubernetes Engine (RKE)

Déployer un cluster Kubernetes en 5 étapes

1. Téléchargez le binaire RKE et rendez-le exécutable
2. Installez Docker sur vos machines
3. Créez un fichier de configuration avec “rke config”
4. Lancez le déploiement du cluster avec “rke up”
5. Le cluster est fonctionnel

```
nodes:
- address: 192.168.166.150
  port: "22"
  role:
  - controlplane
  - etcd
  user: ubuntu
- address: 192.168.166.200
  port: "22"
  role:
  - worker
  user: ubuntu
- address: 192.168.166.177
  port: "22"
  role:
  - worker
  user: ubuntu
```

```
ubuntu@test-master:~/.kube$ kubectl get nodes
NAME              STATUS   ROLES               AGE     VERSION
192.168.166.150   Ready    controlplane,etcd   3m41s   v1.21.6
192.168.166.177   Ready    worker              3m36s   v1.21.6
192.168.166.200   Ready    worker              3m36s   v1.21.6
```

Helm - Kubernetes application manager

- Le gestionnaire de paquets pour Kubernetes
- Propose un marketplace (~11000 packages)
  - https://artifacthub.io
- Automatise le déploiement des objets Kubernetes

- Exemple: Deployer la stack Grafana Loki (Loki, Promtail, Grafana, Fluent Bit)

```
helm upgrade --install loki grafana/loki-stack  \
      --set fluent-bit.enabled=true,promtail.enabled=false,grafana.enabled=true
```

Merci pour votre attention!