Lyon 1

LUGUS LABS

kubernetes

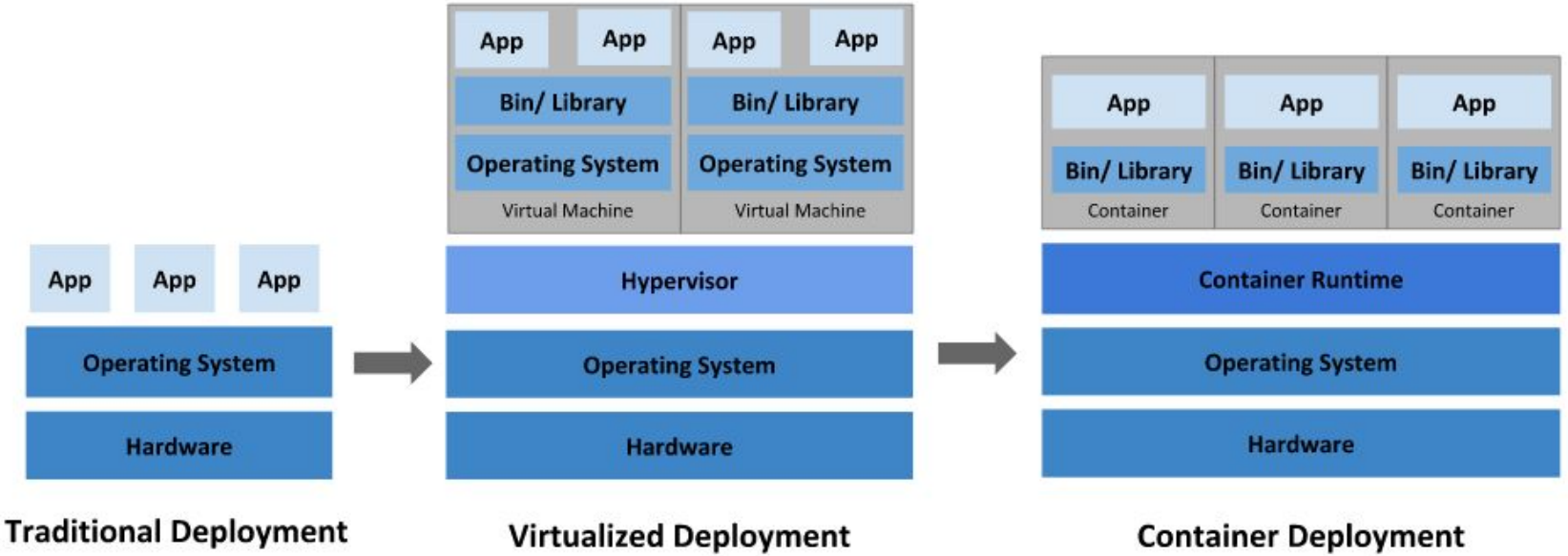
Ostapenco Vladimir - pro@vladost.com

Qui je suis?

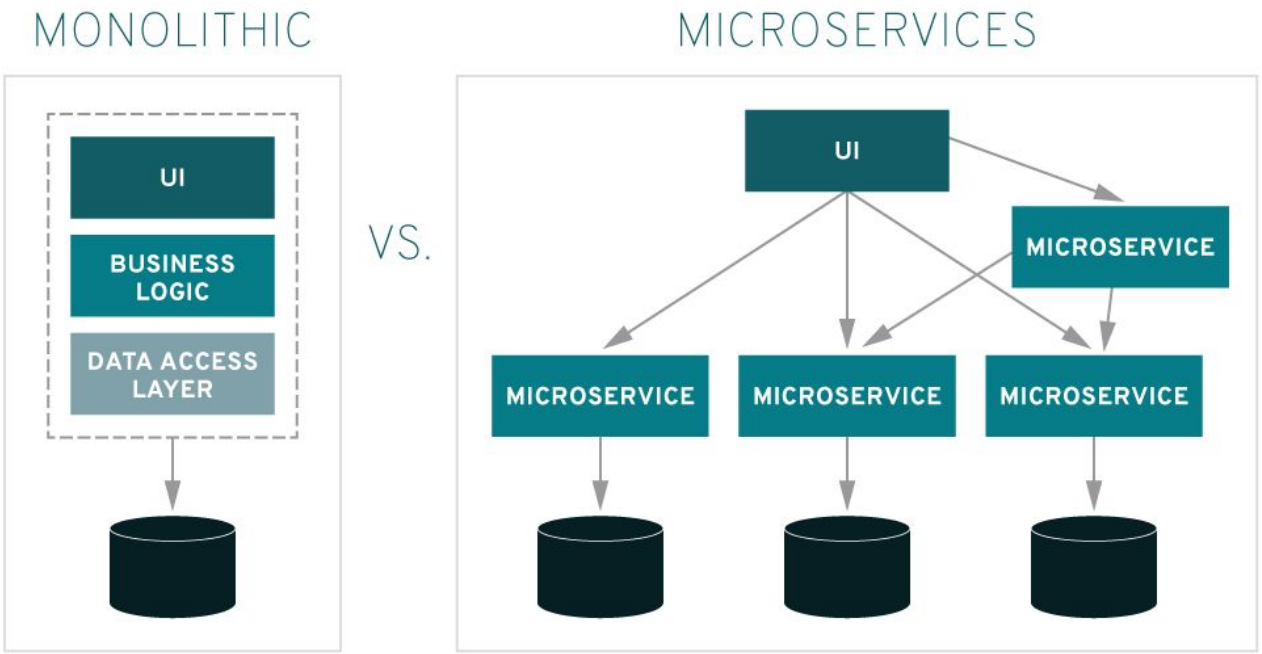
- Entrepreneur
- Ingénieur Système &| DevOps &| Développeur Open-Source
- Chez Lugus Labs
  - https://luguslabs.com/
- Page perso
  - https://vladost.com
- Mail
  - pro@vladost.com

Vladimir Ostapenco

Infrastructure



Microservices



redhat.com

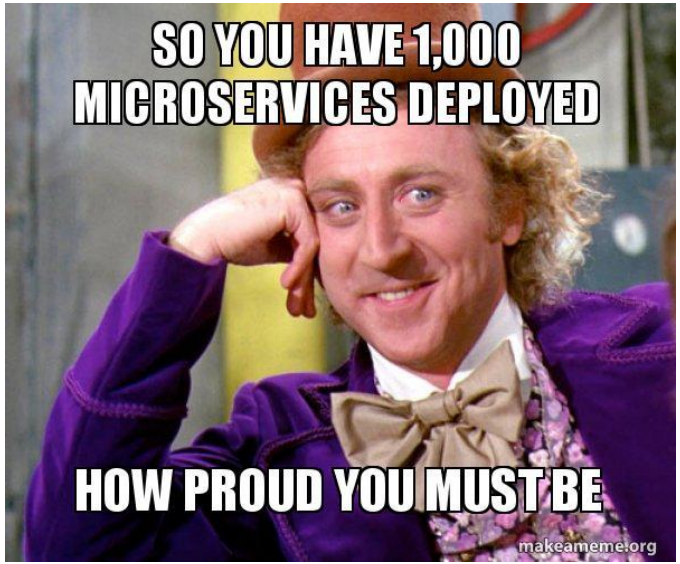
Microservices - Pourquoi?

- 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.***

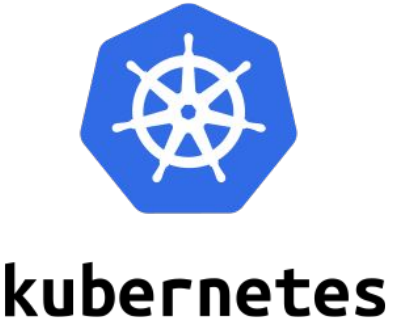
- Adidas (plusieurs releases par jour, 4000 conteneurs, 200 noeuds)
- Google (des milliards des conteneurs par semaine)



Kubernetes (K8s)

Kubernetes (K8s) est un système open-source permettant d'automatiser le déploiement, la mise à l'échelle et la gestion des applications conteneurisées.

Les conteneurs qui composent une application sont regroupés dans des unités logiques pour en faciliter la gestion et la découverte. 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.



Kubernetes.io

Kubernetes (K8S)

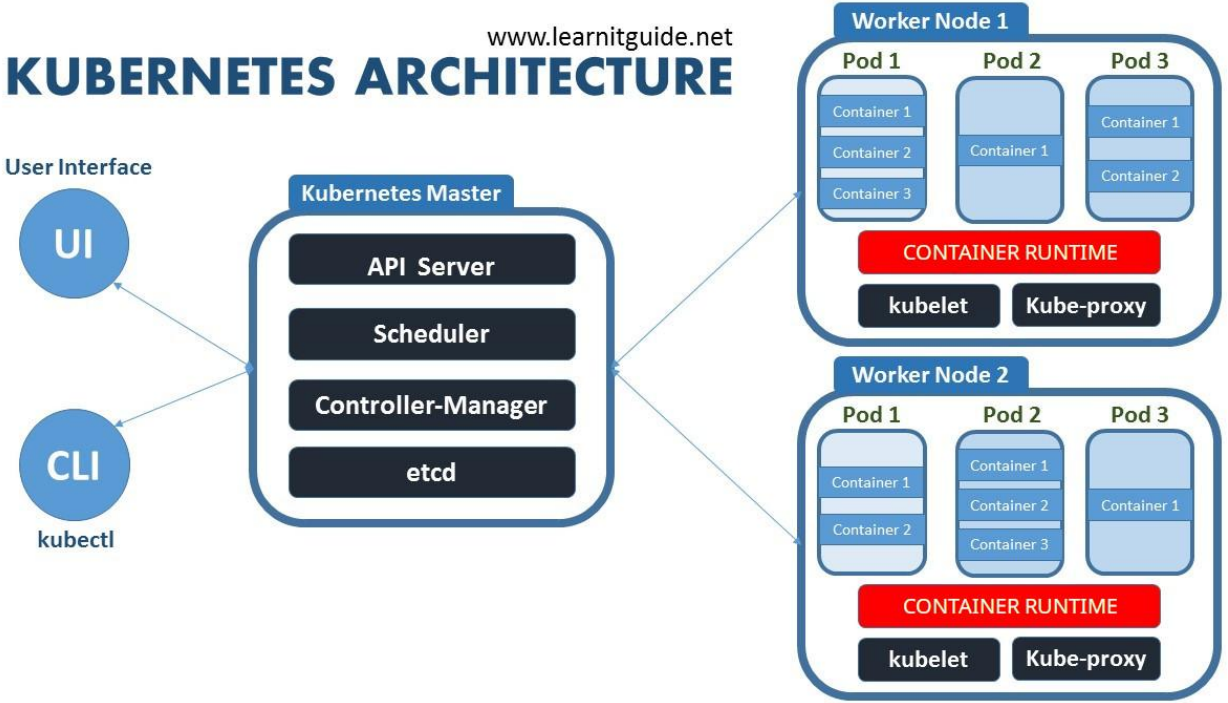
- Conçu pour le déploiement
- Offre une gestion des secrets intégrée
- Est flexible (Run Anywhere)
  - Environments hybrides sont possibles
- Rend la complexité gérable
  - Est capable de gérer des millions des conteneurs
- Est soutenu par une communauté active (~ 3000 contributeurs sur GitHub)
- Optimise l'utilisation des ressources
- Permet zero downtime
- Est DevOps friendly

Kubernetes (K8S)

- Vous avez besoin de K8s
  - Votre application utilise une architecture microservice
  - Vous souhaitez avoir plus de visibilité et de contrôle sur vos déploiements (rollouts/rollbacks)
  - Vous avez besoin d'une haute disponibilité et de l'équilibrage de charge pour vos applications
  - Vous avez besoin d’une mise à l'échelle automatique
  - Vous souhaitez réduire le coût de votre infrastructure
- Vous n’avez pas besoin de K8s
  - Pour les applications simples et légères
  - Vous ne voulez pas refactoriser votre application monolithique
  - Vous avez la flemme =)
- 50% des entreprises du Fortune 100 utilisent Kubernetes

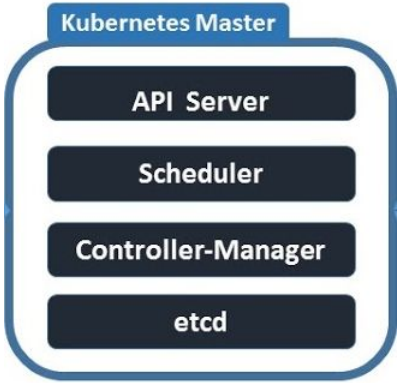
Kubernetes (K8s) - Peut sauver votre entreprise

Architecture



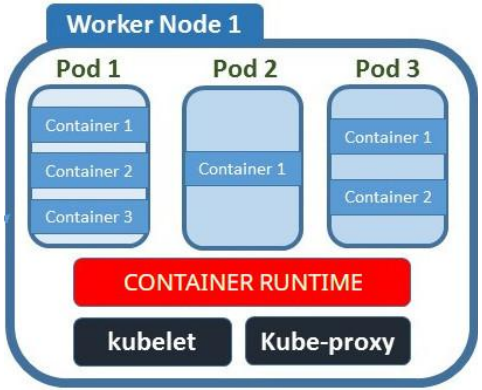
Architecture: Master Node

- Composants de Master Node (Control Plane)
  - **API Server** - Hub de communication pour les composants K8S
    - Expose Kubernetes API
  - **Scheduler** - assigne votre application à un Worker Node
    - Détecte les exigences de l'application et la met sur un bon nœud
  - **Controller manager** - maintient le cluster, gère les pannes de noeuds, réplique les composants, maintient le bon nombre de pods...
  - **Etcd** - base de données pour toute la configuration de cluster
- Par défaut, il ne lance aucun pod et il peut être répliqué pour une haute disponibilité
  - Repliquer un **Master node** = repliquer **etcd** + load balancer **API Server**
  - Demystifying High Availability in Kubernetes Using Kubeadm



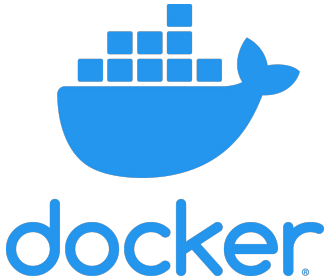
Architecture: Worker Node

- Est responsable de l’exécution de votre application, de son suivi et de la fourniture de services
- Composants de Worker Node
  - **kubelet** - exécute et gère les conteneurs sur le nœud et discute avec l'API Kubernetes
  - **kube-proxy** (service proxy) - effectue l'équilibrage de charge et achemine le trafic des conteneurs
  - **container runtime** - exécute vos conteneurs (docker, containerd, CRI-O...)



Kubernetes (K8s) - Abandon de Docker ?

- Kubernetes utilise CRI (Container Runtime Interface) comme l'API d'exécution des conteneurs
- Docker ne supporte pas CRI (Container Runtime Interface)
  - Kubernetes utilise "**dockershim**" comme un pont entre API docker et CRI (sera supprimé dans la version 1.23 fin 2021)
- Kubernetes n'utilise pas les fonctionnalités de networking et les volumes livrés avec docker par défaut
  - Ces fonctionnalités inutilisées peuvent entraîner des risques de sécurité
- Alternatives
  - **Containerd** (Fait partie et compatible avec Docker)
  - **CRI-O** (Approche plus minimaliste développée par RedHat et utilisé avec OpenShift)



Aujourd'hui

- Namespaces
- Server API et Objets API
- Pods
- Workloads
- Labels, selectors et Annotations
- Services
- Scheduling
- Networking
- Stockage
- Sécurité
- Monitoring
- Cloud controller
- Création d'un cluster K8S

Namespaces

- Est un cluster virtuel, séparation virtuelle de différents environnements
- Permet de séparer le cluster entre différentes équipes ou projets
- Est un moyen de diviser les ressources du cluster entre plusieurs utilisateurs
- Les noms des ressources doivent être uniques dans un namespace
- Namespaces initiés à la création du cluster par défaut
  - **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

Server API

- Tout sur la plateforme K8s est traité comme un objet API et possède une entrée correspondante dans l'API
- Chaque composant Kubernetes communique avec le serveur API et le serveur API uniquement
  - Ils ne se parlent pas directement
- Est le seul qui communique directement avec le **etcd**
- Valide et configure les objets API (les pods, les services, les contrôleurs de réplication, etc)
- **Kubectl -** l'outil CLI qui permet de gérer les objets API K8s
  - **kubectl get nodes** - afficher l'état des noeuds dans un cluster K8S
- Vous pouvez aussi utiliser le serveur API
  - À partir de vos applications via une lib Go ou Python
  - Directement via les appels REST

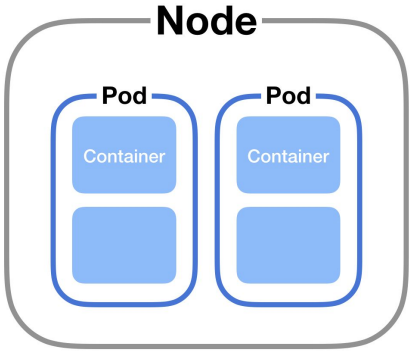
Objet API

- Tout sur la plateforme K8s est traité comme un objet API
  - les pods, les services, les contrôleurs de réplication...
- Definis en YAML
  - langage de sérialisation de données simple et lisible par l'humain
- **Nom unique** pour un type de l'objet dans un **namespace**
- Possedent un **UID** unique à travers le cluster
- 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 et la plus simple unité que vous créez ou déployez
- Un pod représente les processus en cours d'exécution sur votre cluster
- Encapsule
  - Le conteneur d’une application (ou plusieurs conteneurs)
  - Des ressources de stockage
  - Une adresse IP réseau unique
  - Des options d’exécution des conteneurs
- 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
- Normalement vous ne manipulez pas le Pod directement



Workloads

- Une application s'exécutant sur Kubernetes
- Crée et gère plusieurs pods pour vous
  - Assure la replication, les déploiements et self-healing automatiques
- Workloads les plus utilisés
  - **Deployments** et **ReplicaSets**
  - **StatefulSets**
  - **DaemonSets**
  - **Job** et **Cronjob**
- Permet d'effectuer les **Rolling Updates** des Pods
  - Avec une fonctionnalité de **Rollback** en cas d'erreur

Workloads: Deployments

- Le workload qui fournit des mises à jour déclaratives pour
  - Les Pods
  - Les ReplicaSets (plusieurs Pods repliqués)
- Vous décrivez l'état désiré et le Deployment change l'état actuel à l'état souhaité à une vitesse contrôlée
- Quelques commandes:
  - **kubectl apply -f deployment.yaml** - créer un déploiement
  - **kubectl get deployments** - afficher les déploiements
  - **kubectl describe deployments** - afficher la description complete d'un deploiement
  - **kubectl get replicasets** - afficher l'etat des ReplicaSets

```
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: StatefulSets

- Gère le déploiement et la mise à l'échelle d'un ensemble de pods
- Fournit des garanties relatives à l'ordre et à l'unicité de ces pods
- La spécification des conteneurs est identique à celui des déploiements
- Contrairement à un déploiement, 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
- Ce type d’objet est utile quand on utilise des volumes
- Quelques commandes
  - **`kubectl get statefulsets`**
  - **`kubectl describe statefulsets`**

Workloads: DaemonSets

- Garantit que tous (ou certains) nœuds exécutent une copie d'un pod
- N’utilisent pas le scheduler par defaut
- Lorsque des nœuds sont ajoutés au cluster, des pods d’un DaemonSet leur sont ajoutés automatiquement
- Par exemple
  - **kube-proxy et kube-flannel** sont des DaemonSets et leur pods sont exécutés sur chaque noeud
- Nous pouvons spécifier sur quels nœuds le DaemonSet sera exécuté en utilisant un **selector**
- Ignore tous les **taints** des noeuds
- 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 leur travail 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’execution des Jobs
- Exemples
  - **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
  - Sont des paires clé / valeur attachées à des objets, tels que des pods, noeuds et etc
  - Peuvent être utilisés pour organiser et sélectionner des sous-ensembles d'objets
  - 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
  - Exemple:
    - `"release" : "stable", "environment" : "production"`

```
template:
  metadata:
    labels:
      app: nginx
```

- Label Selectors
  - L'utilisateur peut identifier un ensemble d'objets
  - Est la primitive de groupement de base dans Kubernetes
  - Peut être constitué de plusieurs conditions séparées par des virgules

```
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
```

Annotations

- Permettent d’attacher des métadonnées à vos objets
- Les outils et les bibliothèques peuvent récupérer ces métadonnées
- Non utilisés pour identifier et sélectionner des objets
- Quelques exemples
  - 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

- 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
  - peut exposer plusieurs ports
- L'ensemble de pods ciblés par un service est (généralement) déterminé par un Label Selector
- Types des services
  - **ClusterIP** (par defaut) - Expose le service sur une adresse IP interne au cluster
  - **NodePort** - Expose le service sur l'adresse IP de chaque nœud sur un port statique
  - **LoadBalancer** - Exposer le service en externe à l'aide d'un load balancer d'un fournisseur de cloud
  - **ExternalName** - Mappe le service au contenu du champ externalName (par exemple, foo.bar.example.com)

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
  - name: http
    protocol: TCP
    port: 80
    targetPort: 9376
  - name: https
    protocol: TCP
    port: 443
    targetPort: 9377
```

Services: Ingress

- Objet qui gère l'accès externe aux services d'un cluster, généralement HTTP ou HTTPS
- Peut fournir
  - un équilibrage de charge
  - une terminaison SSL
  - un hébergement virtuel basé sur le nom de domaine
- Types d'Ingress
  - **Single Service Ingress** - vous permet d'exposer un seul service
  - **Simple fanout** - achemine le trafic d'une adresse IP unique 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.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: simple-fanout-example
  annotations:

nginx.ingress.kubernetes.io/rewrite-
target: /
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        backend:
          serviceName: service1
          servicePort: 4200
      - path: /bar
        backend:
          serviceName: service2
          servicePort: 8080
```

Scheduling

- Par défaut de K8S tente de trouver le meilleur noeud pour votre Pod en effectuant une série d'étapes
  - Le nœud dispose-t-il de ressources matérielles adéquates?
  - Le nœud manque-t-il de ressources?
  - Le pod demande-t-il un noeud spécifique?
  - Le nœud a-t-il une étiquette correspondante définie par le pod?
  - Si le pod demande un port, est-il disponible?
  - Si le pod demande un volume, peut-il être monté?
  - Est-ce que le pod tolère les taints du noeud? (environnement = production et etc.)
  - Le pod spécifie-t-il une affinity d'un noeud ou d'un pod?
- Utilise la fonction de *spread priority* du sélecteur 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
- Vous pouvez créer votre propre scheduler!

Scheduling: CPU et RAM

- **Requests**
  - “soft cap”
  - sont utilisés pendant le scheduling
- **Limits**
  - “hard cap”
  - peut être sujet à une limitation ou même à une expulsion
- **Valeurs pour le CPU:** 1000m = 1 = 1 vCPU
- **Valeurs pour la RAM:** un nombre à virgule fixe 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
  - Les nœuds ont un taint afin 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/mas
ter:NoSchedule
```

- Toleration
  - Vous permettent de tolérer un taint
  - Vous pouvez mettre une toleration dans la définition d’un pod
    - Et si, par exemple, vous incluez la toleration NoScheduler dans un pod, il peut être schedulé sur le nœud Master
      - **kube-proxy**

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

Networking

- **Au sein du même noeud**
  - utilise les namespaces réseau Linux
  - une paire d'interface Ethernet virtuelle est créée pour chaque conteneur:
    - Un pour le namespace du nœud
    - Un pour le namespace du réseau de conteneur
  - **Linux Ethernet Bridge** permet la communication entre les pods

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

Networking: Container Network Interface (CNI)

- Utilisé pour la communication entre les noeuds
- 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

Networking: Cluster DNS

- Un nom DNS est attribué à chaque service et pod défini dans le cluster
    - **Example:** *my-svc.my-namespace.svc.cluster-domain.example*
- Les searches DNS d'un pod contiennent
    - Son propre espace de noms
    - Le domaine par défaut du cluster
- Kubernetes utilise **Core DNS**
    - Un serveur flexible écrit en Go
    - Prend en charge DNS sur TLS
    - Est configurable
    - Est accessible via un LoadBalancer

```
dnsPolicy: "None"
dnsConfig:
    nameservers:
    - 8.8.8.8
    searches:
    - ns1.svc.cluster.local
    - my.dns.search.suffix
```

Stockage: Volumes Persistants

- Dans K8S les Pods sont éphémères
- Cela nous empêche d'attacher le stockage directement au système de fichiers d'un conteneur
- Persistent Volumes
  - permettent de créer une couche d’abstraction entre l’application et le stockage sous-jacent
  - permet au stockage de suivre plus facilement les pods lorsqu’ils sont supprimés, déplacés et créés dans votre cluster
  - permettent de créer un stockage accessible au-delà de la durée de vie du pod
- Création manuelle = provisioning statique
- **`kubectl get pv`**

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mongodb-pv
spec:
  capacity:
      storage: 1Gi
  accessModes:
      - ReadWriteOnce
      - ReadOnlyMany
  persistentVolumeReclaimPolicy: Retain
  gcePersistentDisk:
      pdName: mongodb
      fsType: ext4
```

Stockage: Modes d’accès

- Permettent de spécifier le mode d’accès d’un nœud à votre volume persistant
- En le spécifiant, vous autorisez le volume à être monté sur un ou plusieurs noeuds, ainsi que lu par un ou plusieurs noeuds
- 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 noeud 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

- persistentVolumeReclaimPolicy
  - **Retain** - gestion manuelle
  - **Recycle** - le contenu sera supprimé pour être utilisé pour les nouveaux VolumeClaims (*rm -rf /thevolume/**)
  - **Delete** - le stockage sous-jacent sera supprimé (GCP volume)

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mongodb-pv
spec:
  capacity:
      storage: 1Gi
  accessModes:
      - ReadWriteOnce
      - ReadOnlyMany
  persistentVolumeReclaimPolicy: Retain
  gcePersistentDisk:
      pdName: mongodb
      fsType: ext4
```

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 et qu'un nouveau pod a été créé
- **`kubectl get pvc`**

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

```
  volumes:
  - name: mongodb-data
      persistentVolumeClaim:
        claimName: mongodb-pvc
```

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: fast
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
```

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

Stockage: hostPath et emptyDir

- hostPath
  - Monte un fichier ou un répertoire du système de fichiers du nœud hôte dans votre pod
- emptyDir
  - Créé lorsqu'un pod est attribué à un nœud
  - Existe tant que ce pod est exécuté sur ce nœud
  - Est initialement vide
  - Peut être utilisé comme le stockage partagé entre les conteneurs de même pod

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-hostpath
spec:
  storageClassName: local-storage
  capacity:
      storage: 1Gi
  accessModes:
      - ReadWriteOnce
  hostPath:
      path: "/mnt/data"
```

```
  containers:
      volumeMounts:
      - mountPath: /tmp/storage
      name: vol
  volumes:
  - name: vol
      emptyDir: {}
```

Sécurité: Authentification

- Le serveur API évalue d’abord si la demande provient d’un compte de service ou d’un utilisateur normal
- Un utilisateur normal peut être
  - une clé privée
  - un magasin d'utilisateurs
  - un fichier contenant une liste de noms d'utilisateur et de mots de passe
- Kubernetes n'a pas d'objet API pour représenter un utilisateur normal
  - L’accès à un utilisateur peut être donné en générant un certificat SSL
- Les comptes de service (service accounts) sont utilisés par K8S pour gérer l'identité de la requête entrant dans le serveur d'API à partir des pods
- Quelques commandes:
  - **`kubectl get serviceaccounts`**
  - **`kubectl create serviceaccount jenkins`**

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  serviceAccountName: jenkins
```

Sécurité: Autorisation RBAC

- Une fois que le serveur d'API a déterminé votre identité (qu'il s'agisse d'un pod ou d'un utilisateur), l'autorisation est gérée par RBAC
- Les règles d'autorisation RBAC (Role Based Access Control) sont configurées via quatre ressources, qui peuvent être regroupées en deux groupes
    - Ce qui peut être fait ?
        - Roles
        - Cluster Roles
    - Qui utilisera ce rôle ?
        - Role Bindings
        - Cluster Role Bindings
- Roles et Role Bindings sont définis au niveau des namespaces
- Cluster Roles et Cluster Role Bindings sont définis au niveau de cluster

Sécurité: Stratégies Réseau (Network Policies)

- Les stratégies réseau vous permettent de spécifier les pods pouvant communiquer avec d'autres pods
  - Par défaut, ils sont ouverts et accessibles à tous
  - Il est donc important de verrouiller les pods
- Vous pouvez appliquer une stratégie réseau à un pod en utilisant des sélecteurs de pod ou de namespace
- Cela aide à sécuriser la communication entre les modules, vous permettant d'identifier les règles d'entrée et de sortie
- Vous pouvez même choisir une plage de bloc CIDR pour appliquer la stratégie réseau
- Il faut avoir un plug-in réseau qui prend en charge les stratégies réseau
  - *Exemple: Canal Plug-in*

Sécurité: Secrets

- Les secrets sont utilisés pour sécuriser les données sensibles auxquelles vous pouvez accéder depuis votre pod
- Les secrets sont créés indépendamment des pods
- Stocker les secrets dans une variable env est une mauvaise pratique
  - Certaines applications peuvent écrire toutes les valeurs des variables env dans les logs
- Stocker des secrets dans des volumes est le meilleur moyen de stocker des secrets
  - sont stockés dans tmpfs et ne sont jamais écrits sur le disque
  - sont mises à jour automatiquement
- Instanciables via un fichier YAML (*encodage base64*)
- Quelques commandes
  - **`kubectl get secrets`**
  - **`kubectl create secret`**

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

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

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

ConfigMaps

- utilisées pour stocker des données non confidentielles dans des paires clé-valeur
- mises à jour automatiquement (si montées comme volume)
- consommées par des pods comme
  - variables d'environnement
  - fichiers de configuration dans un volume

```
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

Monitoring: Serveur des métriques

- K8s fournit des informations détaillées sur les ressources à chaque niveau
- Le monitoring vous permettra d'augmenter les performances de votre cluster et de réduire les goulots d' étranglement
- **Metrics Server**
  - N'est pas fourni par défaut avec K8S
  - Expose les données de métriques via une API
  - Découvre chaque nœud du cluster
  - Interroge chaque nœud pour connaître l'utilisation du processeur et de la mémoire des
    - Noeuds
    - Pods
- Quelques commandes
  - **`kubectl top node`**
  - **`kubectl top pods`**

Monitoring: Surveillance des applications

- K8s peut contrôler automatiquement vos applications
  - Les réparer en les redémarrant ou en les recréant
  - En retenant le trafic entrant jusqu'à leur démarrage complèt

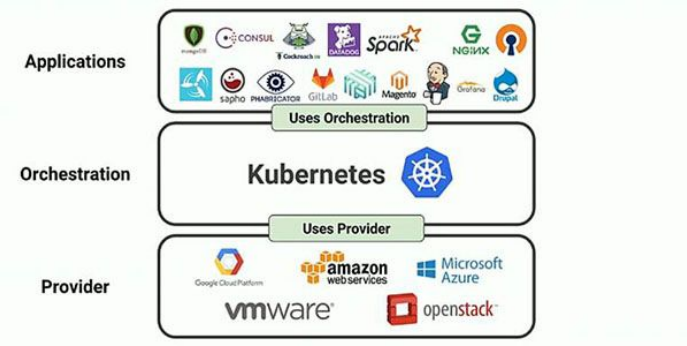
- Vous pouvez insérer des sondes pour surveiller vos applications:
  - **Liveness probes** - permet de vérifier si votre application fonctionne correctement
    - **HTTP Get probe** - Effectue une requête http GET sur le port et l'adresse IP. Si une réponse reçue -> OK, sinon redémarrer le conteneur
    - **Sonde TCP Socket** - 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

  - **Readiness probes** - est un mécanisme dans K8s qui vous permet de vérifier si le conteneur est capable de recevoir les demandes des clients. Si la vérification échoue, le conteneur ne sera pas redémarré, mais le trafic ne sera pas redirigé vers ce conteneur

  - **Startup Probes** - les applications legacy peuvent nécessiter un temps de démarrage supplémentaire lors de leur première initialisation. Attendre le temps de démarrage le plus défavorable et vérifier si le conteneur est disponible. Si la vérification échoue, le conteneur sera 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 Kubernetes
    - Gestion automatisée des noeuds
    - Creation 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/

Création d'un cluster Kubernetes

- Pour deployer un cluster Kubernetes
  - **Kubeadm** - outil **officiel** pour créer et gérer un cluster Kubernetes
  - **RKE** (Rancher Kubernetes Engine) - outil qui permet un déploiement, des mises à jour et des rollbacks faciles d'un cluster Kubernetes
  - **Kubespray** - outil officiel d'automatisation du déploiement d'un cluster avec **Ansible**
  - **Kops** - déploiement d'un cluster Kubernetes sur AWS
- Pendant les TPs, nous allons utiliser le **RKE**
  - Un simple binaire à installer
  - Configuration simple et rapide

RKE

1) Téléchargez le binaire RKE et rendez-le exécutable
2) Installez le Docker sur vos machines
3) Créez un fichier de configuration ou “rke config”
4) Lancez la création 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

- Le gestionnaire de paquets pour Kubernetes
- Propose un marketplace (~6800 packages)
- Automatise le déploiement des objets Kubernetes
- La configuration est possible inline ou via un fichier yaml

- Deployer une stack Grafana Loki (Loki, Promtail, Grafana, Prometheus)

```
helm upgrade --install loki grafana/loki-stack  --set grafana.enabled=true,prometheus.enabled=true
```

Best Practices

- Utilisez la dernière version stable de l'API
- Les fichiers de configuration YAML doivent êtres versionnées dans un Git
- Regroupez les objets liés dans un seul fichier YAML
- Les commandes **kubectl** peuvent êtres exécutées sur un répertoire
  - "**kubectl apply**" sur un répertoire avec des fichiers YAML
- Ne déployez jamais les Pods seuls (Utilisez toujours un Workload)
- Creez les Services avant les Workloads
- Evitez l'utilisation des **hostPort** et **hostNetwork** pour vos Pods
- Définissez et utilisez des labels qui identifient les attributs sémantiques de votre application

Merci pour votre attention!