



kubernetes

Vladimir Ostapenco
pro@vladost.com

Qui je suis?

- **Background**

- Plus de 8 ans d'expérience dans les technologies cloud
- Master SRIV - Université Lyon 1
- Ingénieur infrastructure / DevOps
- Entrepreneur et développeur open-source blockchain
- Thèse sur les impacts environnementaux du cloud

- **Actuellement**

- Ingénieur DevOps - Freelance
- **Spécialisé dans**
 - Orchestration de conteneurs, automatisation et l'optimisation des déploiements et des processus, intégration des pratiques et de la culture DevOps au sein des équipes

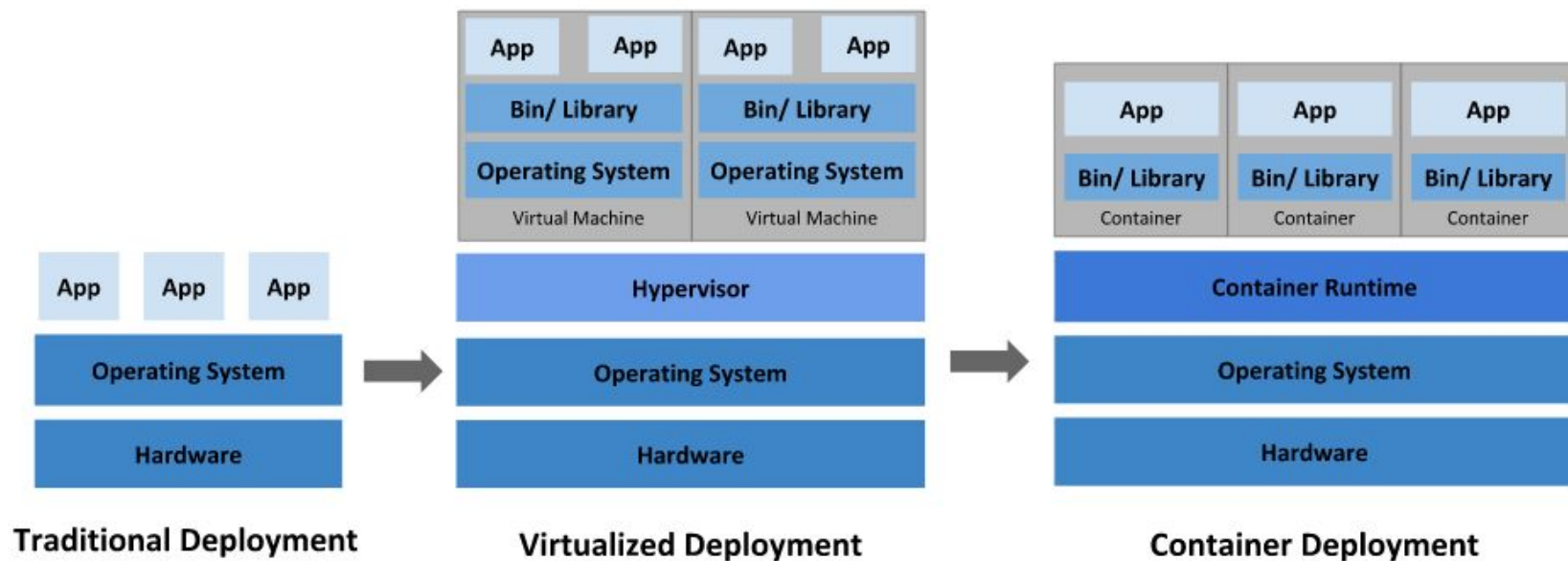
- **Page perso et mail**

- <https://vladost.com>
- pro@vladost.com



Vladimir Ostapenco

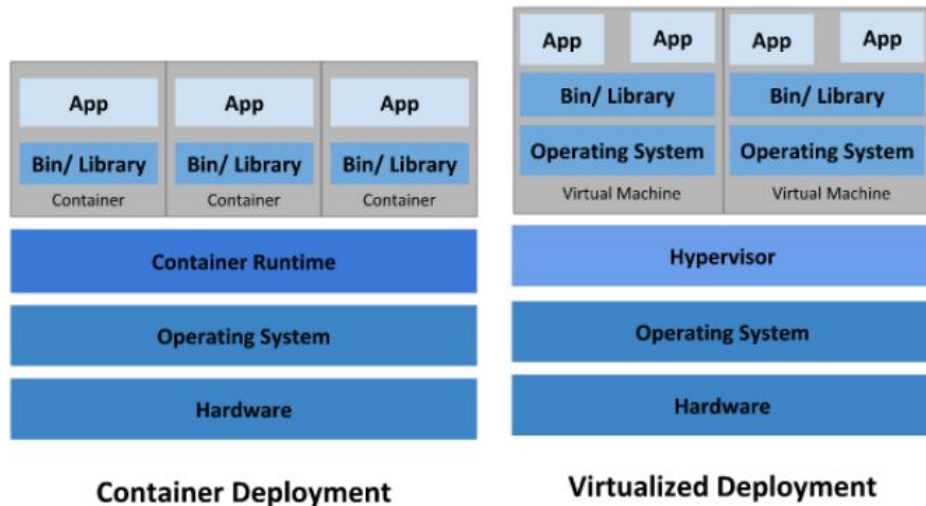
Infrastructure et Déploiement



Source : kubernetes.io

Conteneurs vs VMs

- Conteneurs sont similaires aux VMs
- Ont des propriétés d'isolation assouplies, car partagent
 - Noyau du système d'exploitation
 - Certaines bibliothèques et binaires
- Sont légers
 - Taille de quelques mégaoctets
 - Démarrent en quelques secondes
- Ont leur propre système de fichiers avec
 - Bibliothèques et binaires
- 2 à 3 fois plus de conteneurs que de VMs sur le même serveur



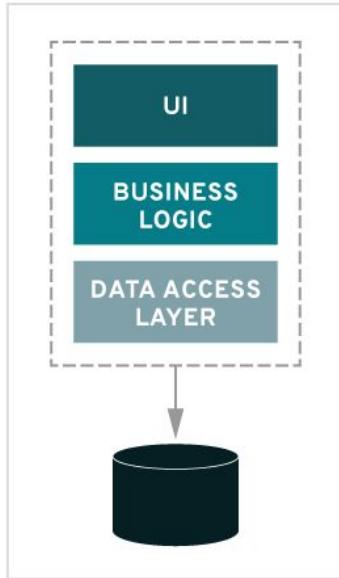
Source : kubernetes.io

Avantages de la conteneurisation

- **Très bon niveau de portabilité**
 - Fonctionne sur Ubuntu, RHEL, CoreOS...
 - On premise, sur les clouds publics et partout ailleurs
- **Cohérence de l'environnement**
 - Fonctionne de la même manière sur un ordinateur portable que dans le cloud
 - Tout au long du cycle de développement, des tests et de la production
- **Agilité dans le développement et déploiement**
 - Facilité et efficacité de la création d'images par rapport aux images VM
 - Facilité de déploiement des applications
- **Facilité d'utilisation dans le contexte d'intégration et déploiement continu**
 - Permet des déploiements fiables et fréquents
 - Avec des mécanismes de Rollbacks rapides et efficaces
- **Isolation des ressources**
- **Faible surcharge en ressources**

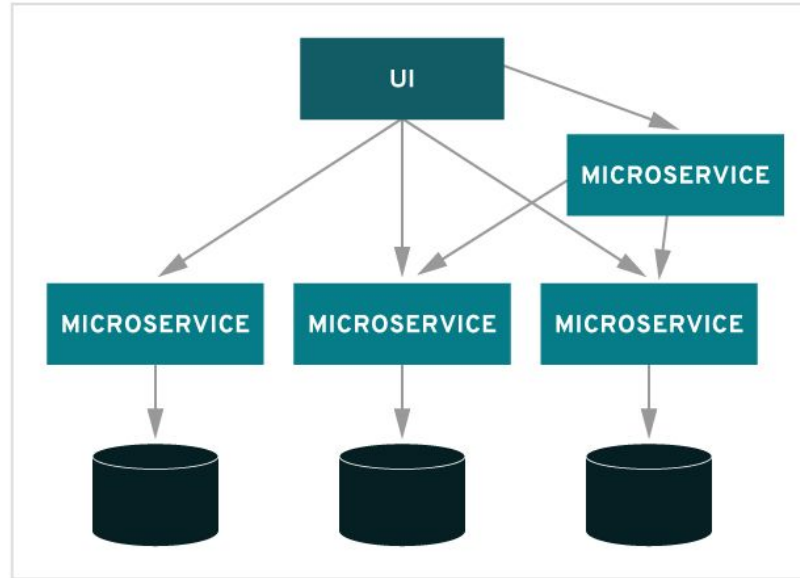
Microservices

MONOLITHIC



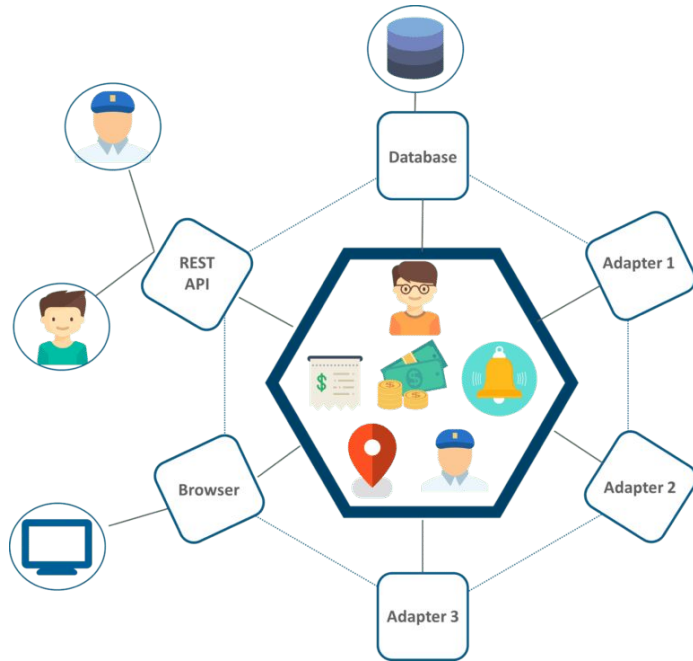
VS.

MICROSERVICES

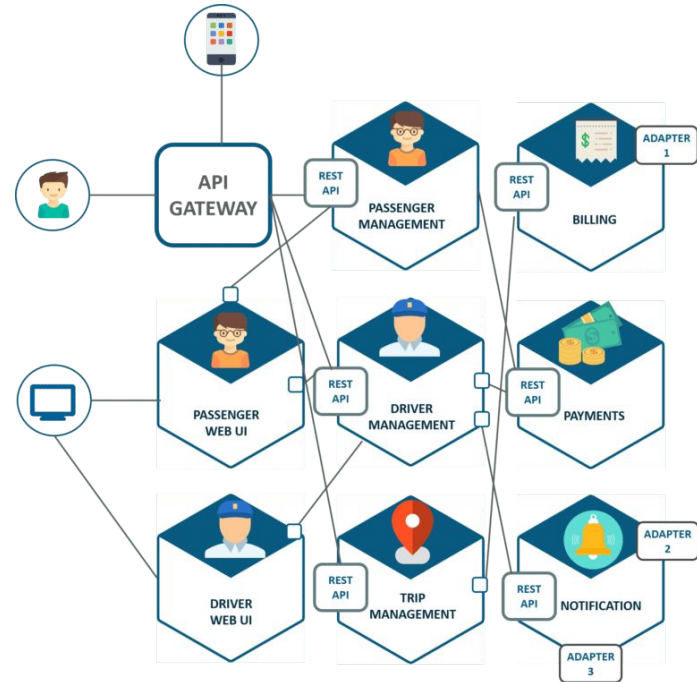


Source : redhat.com

Exemple d'architecture en microservices - Uber



Architecture monolithique



Architecture en microservices

Avantages des microservices

- **Time-to-market plus rapide**

- Cycles de développement plus courts
- Déploiement et des mises à jour plus agiles

- **Évolutivité améliorée**

- Lorsque la demande pour certains services augmente, il suffit d'en déployer plus

- **Résilience améliorée**

- Microservices sont indépendants
- Défaillance d'un élément ne provoque pas la panne de l'ensemble de l'application

- **Facile à déployer et à déboguer**

- Microservices sont relativement petits
- Plus facile à créer, tester, déployer, déboguer et réparer

- **Accessibilité améliorée**

- Développeurs peuvent plus facilement les comprendre, les mettre à jour et les améliorer

- **Flexibilité dans l'utilisation des technologies**

- Grâce à l'utilisation d'API, les développeurs ont la liberté de choisir le meilleur langage et la meilleure technologie pour la fonction nécessaire

Microservices - Use Cases

- **Airbnb**

- Livraisons de code en production lentes, car multiples Reverts et Rollbacks
- *Création et maintenance des applications plus simples*



- **Walmart**

- 6 millions des requêtes par minute, gestion quasi impossible avec une architecture monolithique
- *Performances améliorées*



- **Amazon**

- Modification de la structure des équipes à trois niveaux à de petites équipes efficaces et autonomes gérant une application tout au long de son cycle de vie
- *Équipes autonomes et multifonctionnelles plus efficaces*



- **Netflix**

- Croissance de la base d'utilisateurs très rapide, besoin d'une grande flexibilité
- *Évolutivité simple et mise à l'échelle dynamique*



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 nœuds)
- Google (des milliards des conteneurs par semaine)



Kubernetes (K8s) vient à la rescousse !

Kubernetes (K8s) est une plateforme open source portable et extensible permettant de gérer les charges de travail et les services conteneurisés, permettant à la fois la configuration déclarative et l'automatisation

- S'appuie sur **15 années d'expérience** dans la gestion de charges de travail de production chez Google, associé aux meilleures idées et pratiques de la communauté

Source : kubernetes.io



kubernetes

Kubernetes (K8s)

- Conçu pour le déploiement
- Est flexible (Run anywhere)
- 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 (~ 3900 contributeurs sur GitHub)

Kubernetes (K8s)

- **Kubernetes propose**

- Découverte de service et équilibrage de charge
- Orchestration du stockage
- Déploiements automatisés avec un mécanisme de Rollback
- Bin packing automatique
- Self-healing
- Gestion des secrets et des configurations
- Exécutions en mode batch
- Mise à l'échelle horizontale
- Extensibilité

« Kubernetes is the new Linux OS of the Cloud »

Source : sumlogic.com

Kubernetes (K8s)

- 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 (en 2024) ¹

- 91 % des organisations utilisent des conteneurs en production
- 80 % des organisations utilisent Kubernetes en production
 - Avec une croissance de 14 % par rapport à 2023
 - 13 % des organisations le testent ou l'évaluent activement

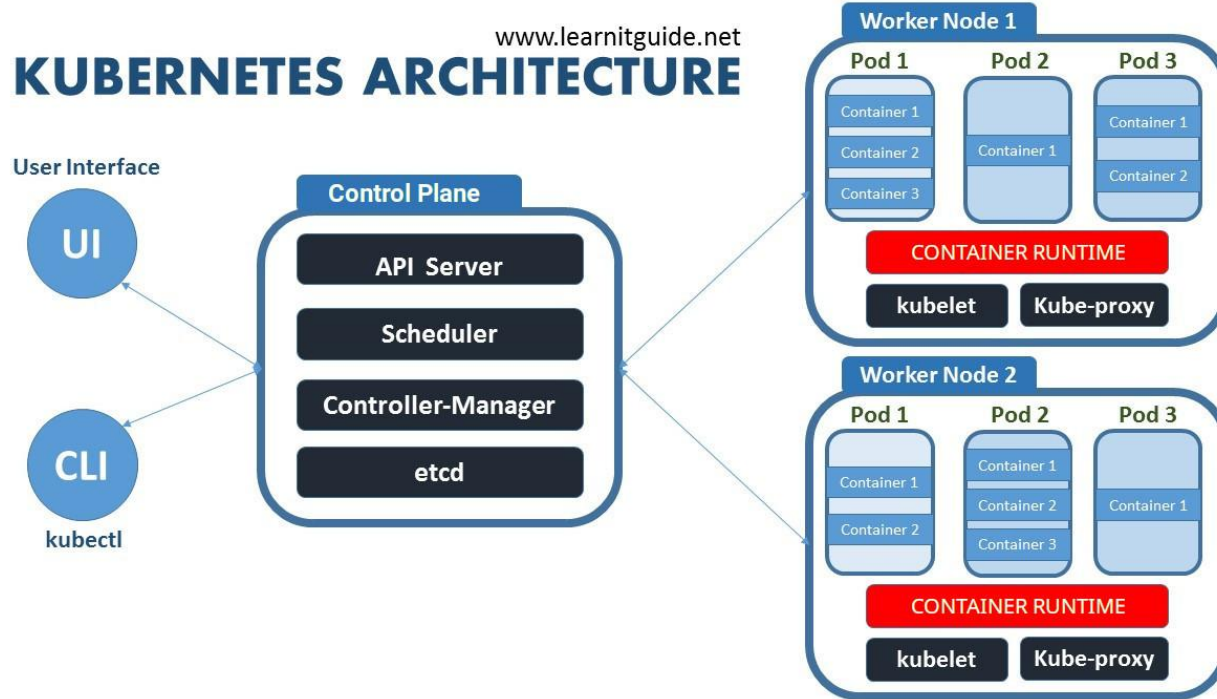


¹ - <https://www.cncf.io/reports/cncf-annual-survey-2024/>

Kubernetes (K8s) : Sauve des entreprises !

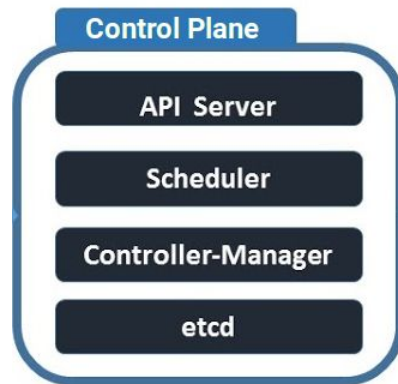


Architecture



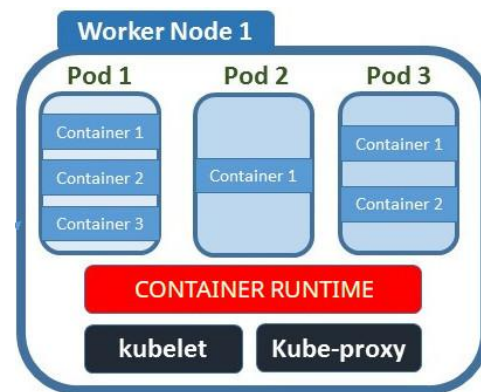
Architecture : Control Plane

- Composants du **Control Plane** prennent des décisions globales concernant le cluster, ainsi que détectent et répondent aux événements du cluster
- Composants du **Control Plane**
 - **API Server** (kube-apiserver) — frontend pour le Control Plane
 - Expose Kubernetes API
 - Hub de communication pour les composants K8s
 - **Etcd** — base de données clé/valeur pour les données du cluster
 - **Scheduler** (kube-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** (kube-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 conteneur et peut être répliqué pour améliorer la disponibilité
 - [Options for Highly Available Topology](#)



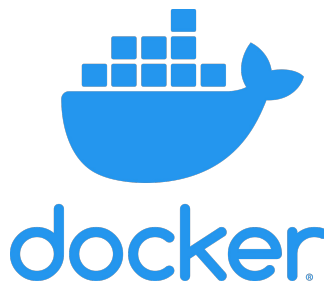
Architecture : Worker Node

- Maintient des conteneurs en état opérationnel et fournit l'environnement d'exécution Kubernetes
- Composants du **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
 - Plusieurs runtimes possibles au sein du même cluster



Kubernetes (K8s) : Abandon de Docker

- **Kubelet** utilise **CRI (Container Runtime Interface)** comme protocol pour communiquer avec **Container Runtime**
- Docker Engine API n'est pas compatible avec le CRI
 - **dockershim** a été utilisé comme un pont entre API docker et CRI
- Kubernetes n'utilise pas les fonctionnalités de networking et de volume livrées avec Docker par défaut
 - Ces fonctionnalités inutilisées peuvent entraîner des risques de sécurité
- **Dockershim** a été supprimé dans la version v1.24 (2022)
- 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 optimisé pour K8s
 - Installation et configuration de **cri-dockerd** avec Docker Engine



Moving Forward - Concepts Kubernetes

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



API Server

- Composant principal du Control Plane qui expose l'API Kubernetes
- Hub de communication
 - Permet aux différentes parties de votre cluster de communiquer entre eux
- Chaque composant Kubernetes communique exclusivement via l'API server
 - Composants ne communiquent pas directement
- Seul composant qui communique directement avec le **etcd**
- Permet d'interroger, manipuler et valider les objets K8s (services, volumes, etc)
- **Kubectl** — l'outil CLI qui permet d'utiliser l'API server pour gérer les objets K8s
 - **kubectl get nodes** — afficher l'état des nœuds du cluster
- API server peut aussi être utilisé
 - À partir de vos applications via une lib C, Go, Python, Java, Rust...
 - Directement via les appels à l'API REST



Objets Kubernetes

- Tout sur la plateforme K8s est traité comme des objets K8s
 - Services, volumes, secrets...
- Unités persistantes représentant l'état de votre cluster
 - Stockés dans **etcd**
- Sont généralement définis au format **YAML**
 - Langage de sérialisation de données simple et lisible
- Une fois créés, K8s s'assure qu'ils existent et sont fonctionnels
- Doivent contenir les champs suivants
 - apiVersion - version de l'API Kubernetes utilisé pour créer l'objet (v1alpha1, v3beta2, v1)
 - kind - type de l'objet
 - metadata - données qui aident à identifier l'objet (nom, label, UID, namespace...)
 - spec - description de l'état souhaité pour l'objet

nginx.yaml

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

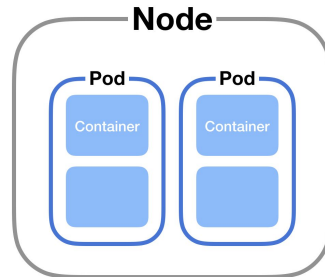
Namespaces

- Permet d'isoler des groupes de ressources au sein d'un seul cluster
- Moyen de diviser les ressources du cluster entre plusieurs utilisateurs
 - Séparation du cluster entre différentes équipes, projets ou environnements
 - Permet de définir des limites d'utilisation des ressources (*ResourceQuota*)
- Namespaces initiés à la création du cluster
 - **default** — namespace par défaut pour les objets sans autre namespace
 - **kube-system** — namespace pour les objets nécessaires au cycle de vie d'un cluster K8s
 - **kube-public** — namespace créé automatiquement et lisible par tous les utilisateurs
 - Utilisé par le cluster, au cas où certaines ressources devraient être visibles et lisibles publiquement
 - **kube-node-lease** — namespace dédié aux objets responsables des heartbeats des nœuds afin de détecter les pannes de nœuds
- **NB :** Noms des ressources doivent être uniques dans un namespace



Pod

- Unité de base de Kubernetes
 - Plus petite unité que vous pouvez déployer dans K8s
- Représente une unité de déploiement
 - Instance unique d'une application
 - Regroupe **un** ou **plusieurs conteneurs étroitement couplés** qui partagent des ressources (colocalisés et coplanifiés)
- Encapsule
 - Un ou plusieurs conteneurs
 - Ressources de stockage
 - Ressources réseau
 - Possède une adresse IP unique à l'échelle du cluster
 - Options d'exécution des conteneurs
- Normalement, vous ne manipulez pas les Pods directement
 - Conçus comme des entités relativement éphémères et jetables

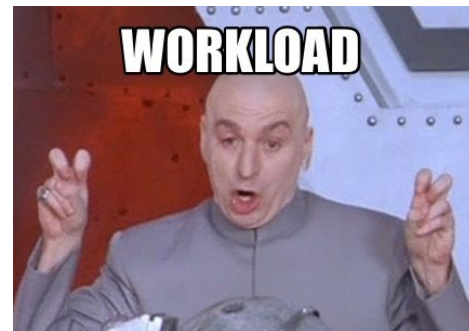


nginx.yaml

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


Workload objects

- **Workload objects** - niveau d'abstraction plus élevé qu'un Pod
- Permettent une gestion déclarative des applications
- En fonction de la spécification de workload object, le Control Plane crée et gère automatiquement les Pods
- Workload objects par défaut
 - **Deployment** et **ReplicaSet**
 - **StatefulSet**
 - **DaemonSet**
 - **Job** et **Cronjob**



Workloads: Deployment

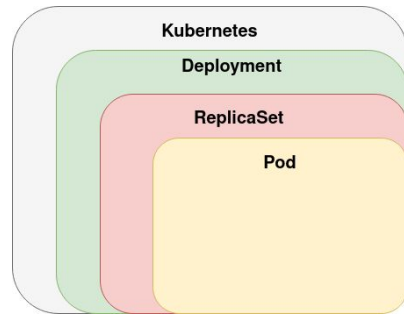
- Manière la plus courante pour déployer une application sur un cluster K8s
- Dans un Deployment
 - Vous décrivez l'état souhaité pour votre application
 - K8s change l'état actuel en l'état souhaité à un rythme contrôlé
- Propose des mécanismes
 - Mises à jour déclaratives avec les mécanismes de Rolling update et de Rollback
 - Mise à échelle (Scale up) pour supporter plus de charge
- 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

nginx-deployment.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: Deployment

- Ressource de haut niveau englobant un **ReplicaSet**
 - Un ou plusieurs Pods répliqués
 - Garantit qu'un certain nombre des Pods est en cours d'exécution à tout moment
 - Doit toujours être créé par un Deployment
- Dans un Deployment **les Pods sont éphémères**
 - Peuvent être remplacés, recréés ou redémarrés à tout moment
 - Ensemble de Pods d'un Deployment exécuté sur le cluster à l'instant T n'est pas forcément celui qui sera exécuté à l'instant T+1
 - Seul leur nombre est garanti
 - Bon choix **pour** gérer **les applications sans état**



Workloads: StatefulSet

- Workload object adapté à la gestion des 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
- Pods créés ne sont pas interchangeables
- Si un Pod est défaillant, il est remplacé par un autre avec le même nom d'hôte et les mêmes configurations
- Utile pour les applications qui nécessitent un ou plusieurs des éléments suivants
 - Identifiants réseau stables et uniques
 - Stockage persistant
 - Déploiement, mise à l'échelle et mises à jour ordonnés
- Quelques commandes
 - **kubectl get statefulsets**
 - **kubectl describe statefulsets**



Workloads: DaemonSet

- Workload object qui
 - 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 y sont ajoutés automatiquement
- Exemples d'utilisation
 - Solutions de supervision ou de gestion des logs
 - Composants proxy et réseau de Kubernetes (**kube-proxy**)
- 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 exécutent une tâche et s'arrêtent
 - 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

blender-job.yaml

```
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
 - "app" : "nginx", "environment" : "production"
- Peuvent être attachés aux objets au moment de la création ou ajoutés et modifiés à tout moment
- Chaque clé doit être unique pour un objet donné

- **Label selectors**

- Permettent d'identifier et de sélectionner un ensemble d'objets

nginx-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx
    name: nginx-deployment
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

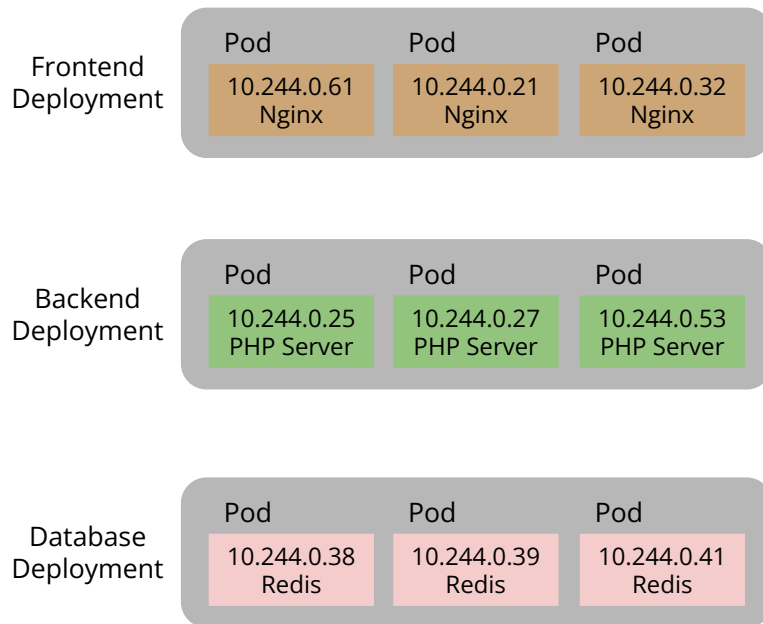
- Permettent d'attacher des métadonnées à vos objets
- Ne peuvent pas être utilisés pour identifier ou sélectionner des objets
- Peuvent être utilisés par des outils et des bibliothèques
- Exemples
 - Information sur la version de l'application ou de l'image docker
 - Repository avec le code source de l'application
 - Coordonnées des personnes responsables

annotations-pod.yaml

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

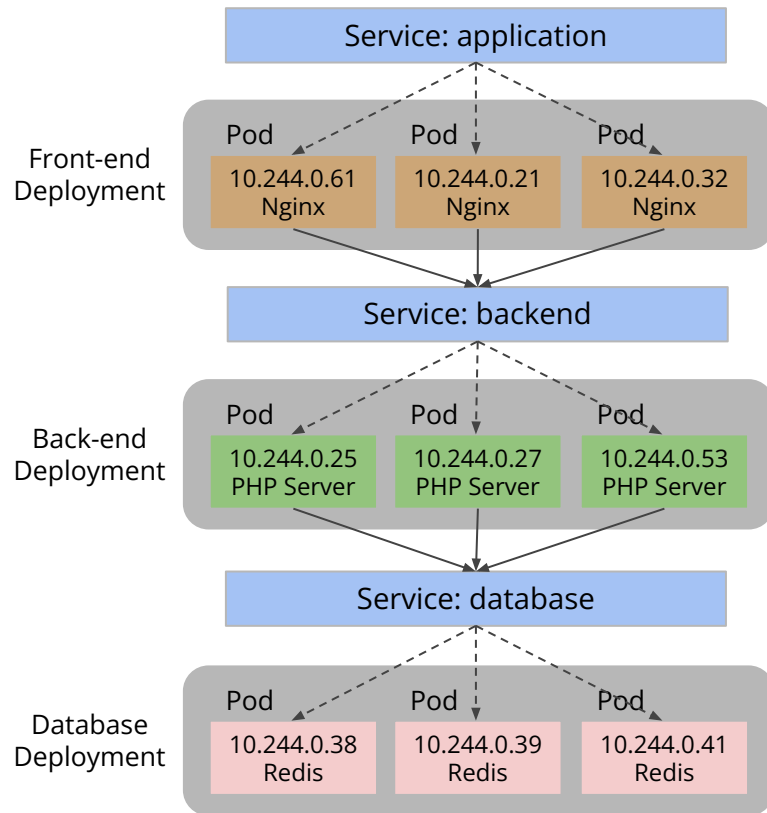

Accès aux Pods

- Chaque Pod obtient **sa propre adresse IP**
- Dans un déploiement
 - Ensemble de Pods s'exécutant à un moment donné, peut-être différent de l'ensemble de Pods exécutant un instant plus tard
- Problème
 - Dans une configuration frontend/backend
 - Comment les Pods du Frontend trouvent-ils et gardent-ils la trace de l'adresse IP à laquelle se connecter afin d'utiliser le backend ?



Services

- Permet d'exposer vos Pods en tant que service réseau et d'effectuer une résolution d'applications entre les Pods
- Est une abstraction qui définit un ensemble logique de Pods et une politique permettant d'y accéder
 - Parfois appelée microservice
 - A une adresse IP définie
 - Décide de router le trafic vers l'un des Pods (load balancing)
- Ensemble de Pods ciblés par un service est déterminé par un **Label Selector**



Services : Types

- **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

nginx-service.yaml

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

nginx-deployment.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
```

Services : Découverte

Principales méthodes de découverte d'un Service dans le cluster

- **Variables d'environnement**

- **Kubelet** ajoute dans chaque Pod un ensemble de variables d'environnement pour chaque service actif
- Pour le service `svcname`
 - Les variables `{SVCNAME}_SERVICE_HOST` et `{SVCNAME}_SERVICE_PORT` seront disponibles dans chaque Pod

- **DNS**

- Un service DNS doit être configuré sur le cluster
- Le service DNS surveille les nouveaux services et crée des enregistrements DNS
- 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`

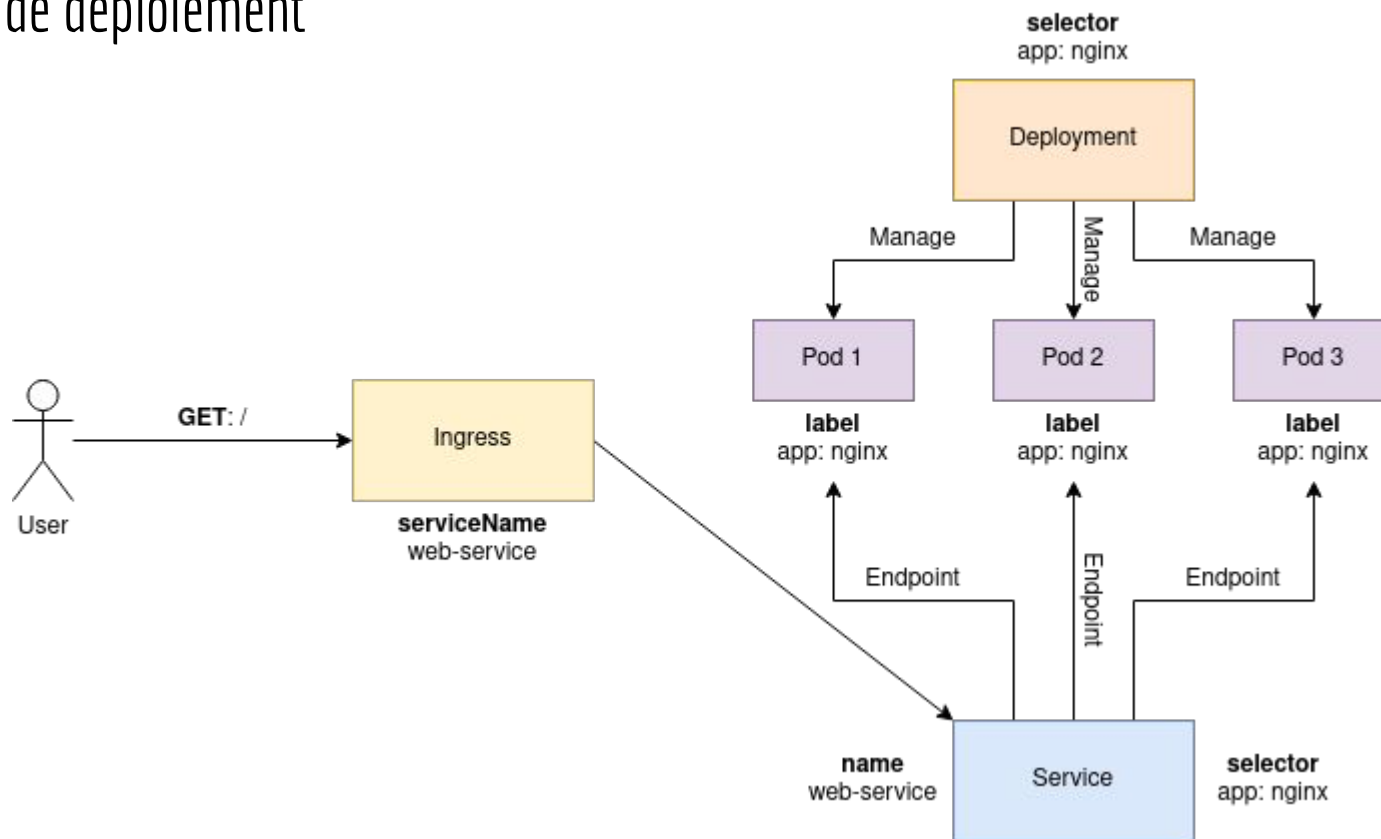
Ingress

- Gère l'accès externe HTTP ou HTTPS aux services
- Peut fournir
 - URLs accessibles de l'extérieur pour les services
 - Équilibrage de charge
 - Terminaison SSL / TLS
 - Hébergement virtuel basé sur le nom de domaine
- Nécessite un **Ingress Controller**
- Types
 - **Single service** — 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** — achemine le trafic vers plusieurs services, en fonction du hostname demandé

simple-fanout-ingress.yaml

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

- Scheduler surveille les Pods nouvellement créés qui n'ont pas été attribués à un nœud
 - Pour chaque Pod découvert, trouve le meilleur nœud pour ce Pod
- *kube-scheduler* sélectionne un nœud pour un Pod en deux 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
 - En tenant compte de l'utilisation des ressources, de la capacité des nœuds et des exigences de communication entre les Pods...
 - **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

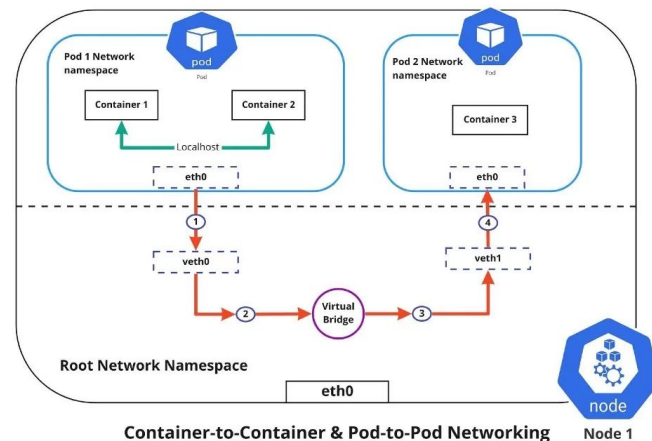
- Permet de spécifier la quantité des ressources dont un conteneur a besoin
- **Requests**
 - "Soft cap"
 - Utilisé seulement pendant le scheduling
- **Limits**
 - "Hard cap"
 - Container Runtime empêche le conteneur d'utiliser plus
- **Types de ressources**
 - **CPU** exprimé en CPU units : 1 CPU unit = 1 CPU ou vCPU = 1000m
 - **Mémoire** exprimé en nombre des octets
 - Peut être suivi d'un suffixe comme M, K ou Mi, Ki etc.

frontend-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
    - name: app
      image: app:v4
      resources:
        requests:
          memory: "64M"
          cpu: "250m"
        limits:
          memory: "128M"
          cpu: "500m"
```


Networking : Communication

- Tous les Pods peuvent communiquer avec tous les autres Pods, qu'ils se trouvent sur le même nœud ou sur des nœuds différents
- **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 nœud (**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)



(Nived Velayudhan, CC BY-SA 4.0)

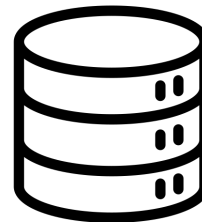
Networking: Container Network Interface (CNI)

- Utilisé pour la communication entre les nœuds
- Garantit que tous les Pods peuvent communiquer avec tous les autres Pods exécutés sur n'importe quel autre nœud sans NAT
- Installé via un plugin et n'est pas natif à la solution K8s
 - Installe et lance un agent de réseau sur chaque nœud (**DaemonSet**)
- CNI les plus connus
 - **Flannel** — configure un réseau overlay IPv4 de niveau 3 basé souvent sur VXLAN
 - **Calico** — configure un réseau niveau 3 basé sur BGP, pas d'encapsulation
 - **Cilium** — basé sur eBPF (très performant), il prend en charge le chiffrement, l'authentification, les politiques de sécurité avancées et l'observabilité



Stockage

- Stockage dans un conteneur est éphémère
- Cela pose deux problèmes
 - **Perte de fichiers** lorsqu'un conteneur plante
 - État du conteneur n'est pas enregistré, donc tous les fichiers créés ou modifiés sont perdus
 - **kubelet** redémarre le conteneur, mais avec un état propre
 - **Partage des fichiers** entre des conteneurs exécutés au sein du même Pod
- Nécessité d'un stockage persistant
 - Volumes Kubernetes



Stockage : Volumes

- Volumes Kubernetes 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** — 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 un Pod
 - **configMap, secret** — un moyen d'injecter des données de configuration ou des secrets dans des Pods
 - **Cephfs, fc, nfs, iscsi** — permet à un volume existant d'être monté dans un Pod
 - **persistentVolumeClaim** — utilisé pour monter un volume persistant dans un Pod

pod-with-volume.yaml

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

Stockage : Persistent Volumes

- Sous-système qui sépare la partie fourniture de stockage de la partie consommation de stockage
- Abstraction permettant de découpler
 - Volumes mis à disposition par les administrateurs
 - Demandes d'espace de stockage des développeurs pour leurs applications
- Composé de deux éléments (objets Kubernetes)
 - **PersistentVolume** — configuré par un administrateur ou provisionné dynamiquement par un *StorageClass*
 - **PersistentVolumeClaim** — peut être vu comme une demande de stockage et peut être attaché à un Pod

Stockage : Persistent Volume (PV)

- Configuré par un administrateur (statique) ou provisionné dynamiquement par un *Storage Class* (dynamique)
- 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, de FC ou d'un système de stockage spécifique au fournisseur cloud
- Types des PVs (implémentés sous forme de plugins)
 - **CSI**, FC, iSCSI, NFS, local et hostPath
- **kubectl get pv**

pv-volume.yaml

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

Stockage : Persistent Volume Claims (PVC)

- Permettent aux développeurs de demander du stockage pour leur applications sans avoir besoin de savoir où se trouve le stockage sous-jacent
- Utilisent (consomment) des Persistent Volumes
- Sont utilisés par les Pods
- Restent liées au Persistent Volume même si le Pod qui les utilisait a été supprimé
- Persistent Volume ne sera pas libéré tant que PVC lié existe
- **kubectl get pvc**

pv-claim.yaml

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

pod.yaml

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

Stockage : Modes d'accès

- Permet de spécifier le mode d'accès pris en charge par un Persistent Volume
- Modes d'accès
 - **ReadWriteOnce** — seul un nœud peut monter le volume en lecture-écriture
 - **ReadOnlyMany** — plusieurs nœuds peuvent monter le volume en lecture
 - **ReadWriteMany** — plusieurs nœuds peuvent monter le volume en lecture-écriture
 - **ReadWriteOncePod** — le volume peut être monté en lecture-écriture par un seul Pod
- N'assurent pas la protection en écriture une fois le stockage monté
 - Utilisés pour associer les *PersistentVolumeClaims* aux *PersistentVolumes*
- PV ne peut être monté qu'en utilisant un mode d'accès à la fois, même s'il prend en charge plusieurs

Stockage : Politiques de rétention

- Indique au cluster que faire avec le Persistent 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)

pv-recycle.yaml

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

Stockage : StorageClass

- Permet aux administrateurs de décrire les classes de stockage qu'ils proposent
- Permet le provisionnement automatique du stockage, sans qu'il soit nécessaire de créer un *PersistentVolume*
- Il suffit d'indiquer simplement au K8s de quel stockage vous avez besoin avec un *PersistentVolumeClaim* et il créera le *PersistentVolume* pour vous
- ***Provisioning dynamique***

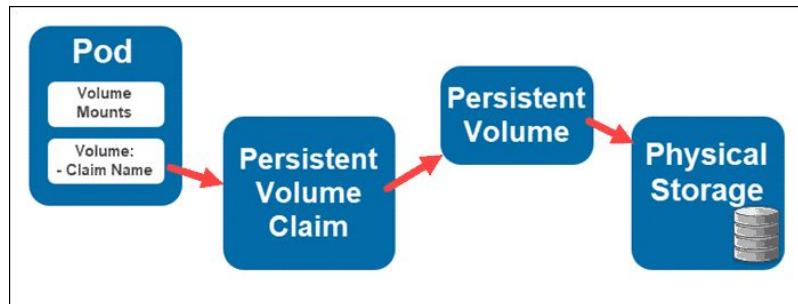
sc-csi-cinder.yaml

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: csi-cinder
provisioner: cinder.csi.openstack.org
parameters:
  availability: nova
```

mongodb-pvc.yaml

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

Stockage: Persistent Volumes



Source: phoenixnap.com

app-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: app-pod
spec:
  containers:
    - image: busybox
      name: test-container
      volumeMounts:
        - mountPath: /app/data
          name: app-volume
  volumes:
    - name: app-volume
      persistentVolumeClaim:
        claimName: app-pvc
```

app-pvc.yaml

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: app-pvc
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 100Mi
```

app-pv.yaml

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: app-pv
spec:
  storageClassName: manual
  capacity:
    storage: 100Mi
  accessModes:
    - ReadWriteMany
  hostPath:
    path: "/mnt/app"
```

Conteneurs d'initialisation (initContainers)

- Conteneurs spécialisés qui s'exécutent avant les conteneurs de l'application dans un Pod
- 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 se terminer avec succès avant le démarrage du conteneur principal

pod-with-init.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: 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
- Sont définis dans la description du Pod au niveau du conteneur
 - Avec le champ `env` ou `envFrom`
- Peuvent contenir des secrets (`Secrets`) et des configurations (`ConfigMaps`)
- Peuvent être interdépendants
 - Vous pouvez utiliser `$(VAR_NAME)` dans le champ `value` d'une autre variable d'environnement

pod-with-env-vars.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-env-vars
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 fournir les données sensibles aux Pods de manière sécurisée
- Sont créés indépendamment des Pods qui les utilisent
- Peuvent être fournis aux Pods 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 **kubectf**

credentials-secret.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: credentials
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

pod-with-secret.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-secret
spec:
  containers:
    - name: nginx
      image: nginx
      volumeMounts:
        - name: credentials-vol
          mountPath: "/cred"
  volumes:
    - name: credentials-vol
      secret:
        secretName: credentials
```

ConfigMaps

- Permettent de stocker et de fournir aux conteneurs des données de configuration non confidentielles dans des paires clé-valeur
- Utiles pour définir les données de configuration séparément du code de l'application
- 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)

configmap-game-demo.yaml

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

Dans la partie spec du Pod

```
containers:
...
  volumeMounts:
  - name: config
    mountPath: "/config"
```

```
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 Pods mais aussi le fonctionnement des applications avec les mécanismes de
 - **Réparation des applications** via le redémarrage ou la recreation des conteneurs
 - **Rétention du trafic entrant** jusqu'au démarrage complet de l'application
- Pour utiliser ces mécanismes, vous devez **ajouter des sondes (probes) dans vos Pods**

Types de sondes

- **Sonde Liveness** — permet de vérifier si l'application exécutée dans un Pod fonctionne correctement
 - Si la vérification réussit, tout est en ordre
 - Si la vérification échoue, le conteneur est redémarré
- **Sonde Readiness** — permet de vérifier si l'application est capable de recevoir les demandes des clients
 - Si la vérification échoue, le conteneur n'est pas redémarré, mais le trafic n'est pas redirigé vers ce conteneur
- **Sonde Startup** — utile pour les applications avec un temps de démarrage long
 - Un moyen de différer l'exécution des sondes Liveness and Readiness
 - Attendre le temps de démarrage le plus défavorable et vérifier si l'application fonctionne

Mécanismes de vérification

- **HTTP Get probe**

- Effectue une requête HTTP GET sur un port et un chemin
- Vérification est considérée comme réussie si la réponse a un code d'état supérieur ou égal à 200 et inférieur à 400

- **TCP Socket probe**

- Tente d'ouvrir une connexion TCP sur un port spécifique
- Vérification est considérée comme réussie si le port est ouvert

- **Exec probe**

- Exécute une commande arbitraire dans le conteneur
- Vérification est considérée comme réussie si la commande se termine avec un *code* d'état de 0

- **GRPC probe**

- Effectue un appel de procédure à distance à l'aide de gRPC
- Vérification est considérée comme réussie si le *state* de la réponse est *SERVING*

Dans la partie spec du Pod

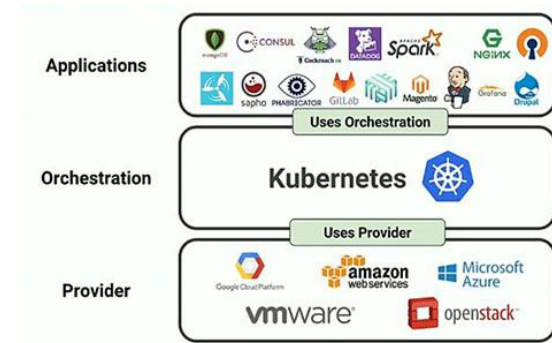
```
containers:
...
  livenessProbe:
    httpGet:
      path: /
      port: 80
      periodSeconds: 5
```

```
containers:
...
  livenessProbe:
    tcpSocket:
      port: 8080
      periodSeconds: 5
```

```
containers:
...
  livenessProbe:
    exec:
      command: ["ping"]
      periodSeconds: 5
```

Cloud Controller Manager

- Composant de **Control Plane** qui permet de **lier un cluster K8s à l'API d'un fournisseur cloud**
 - Openstack, AWS (Amazon Web Services), GCP (Google Cloud Platform), Azure...
- Permet d'intégrer les fonctionnalités et des services du cloud provider dans un cluster K8s
 - Gestion automatisée des nœuds
 - Création et gestion des volumes
 - Création 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
- **kOps**
 - Outil de déploiement des clusters Kubernetes sur AWS et GCE
- **Kubespray**
 - Outil d'automatisation du déploiement des clusters Kubernetes avec **Ansible**
 - Sur GCE, Azure, OpenStack, AWS, vSphere, Equinix Metal, Oracle Cloud Infrastructure ou Baremetal
- **RKE** (Rancher Kubernetes Engine)
 - Outil qui facilite et automatise le déploiement, les mises à jour et les Rollbacks des clusters Kubernetes
- **Cluster API**
 - Sous-projet Kubernetes fournissant des outils pour simplifier le déploiement des clusters K8s
- Pendant les TPs, vous utiliserez le **RKE** et **Kubeadm** dans la partie bonus

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

```
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
```

cluster.yml

```
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
```



Helm - Kubernetes application manager

- Gestionnaire de paquets pour Kubernetes
- Propose un marketplace (~18000 packages)
 - <https://artifacthub.io>
- Automatise le déploiement des objets Kubernetes



- **Exemple :** Déployer une solution de centralisation des logs Grafana Loki (Loki, Grafana, FluentBit)

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

Merci pour votre attention !

