

# **CLOUD COMPUTING : DOCKER, KUBERNETES DÉPLOIEMENT ET SÉCURITÉ DES APPLICATIONS CONTENEURISÉES**

---

Dr. Mohamed TALHA

Technical Leader in Cloud Computing, Java / JEE and Big Data

# Plan

- Virtualisation
  - Machine Virtuelle, Conteneur, VM versus Container
- Docker
  - Dockerfile, Image Docker, Docker Container
  - Ateliers : installer Docker, manipuler les images et les conteneurs, Docker Hub
- Kubernetes (K8s)
  - Ateliers : mettre en place un Cluster K8s avec minikube & kubectl
  - Objets K8s, Workloads K8s, Cluster K8s
  - Ateliers : déployer une API REST Spring Boot sur un Cluster K8s
- Sécurité des applications conteneurisées avec K8s
  - Config Map et Secrets K8s
  - Ateliers : sécuriser une API REST conteneurisée
  - Bonnes pratiques de gestion des Secrets K8s
  - HashiCorp Vault pour K8s
  - Ateliers : gérer les Secrets K8s avec HashiCorp Vault

# VIRTUALISATION

---

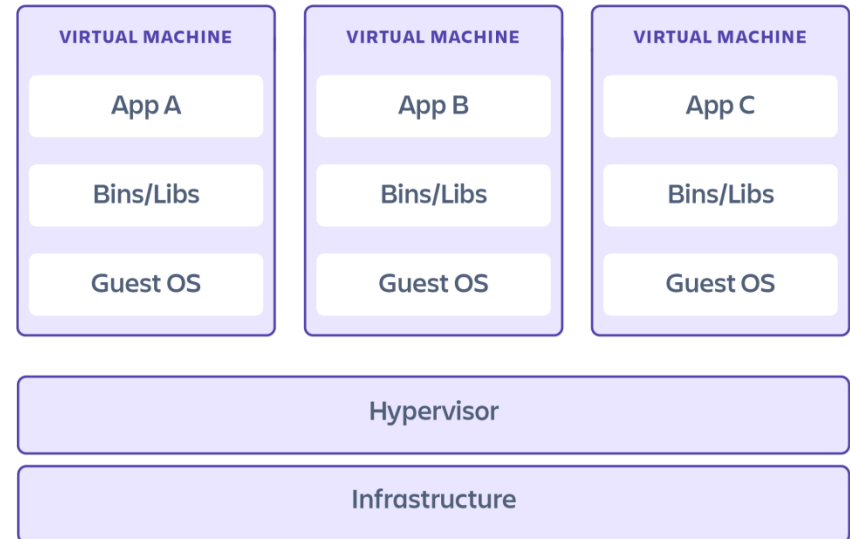
- Virtualisation
- Machine Virtuelle
- Conteneur
- VM vs Container

# Virtualisation

- La virtualisation est le processus permettant de "virtualiser" une ressource matérielle singulière (RAM, CPU, Stockage ou Réseau) pour pouvoir la représenter sous forme de ressources multiples.
  - **Une Machine Virtuelle (VM)** virtualise toute une machine y compris son système d'exploitation. Cette technologie est basée sur un **hyperviseur**, comme VMware, Virtual Box, Microsoft Hyper-V, etc.
  - **Un conteneur (Container)** ne virtualise que les couches logicielles au-dessus du système d'exploitation hôte à travers un **Container Engine**, comme Docker, RKT, LXC ou CRI-O. Un conteneur peut être vu comme une VM allégée n'ayant pas d'OS mais utilise celui de la machine hôte.

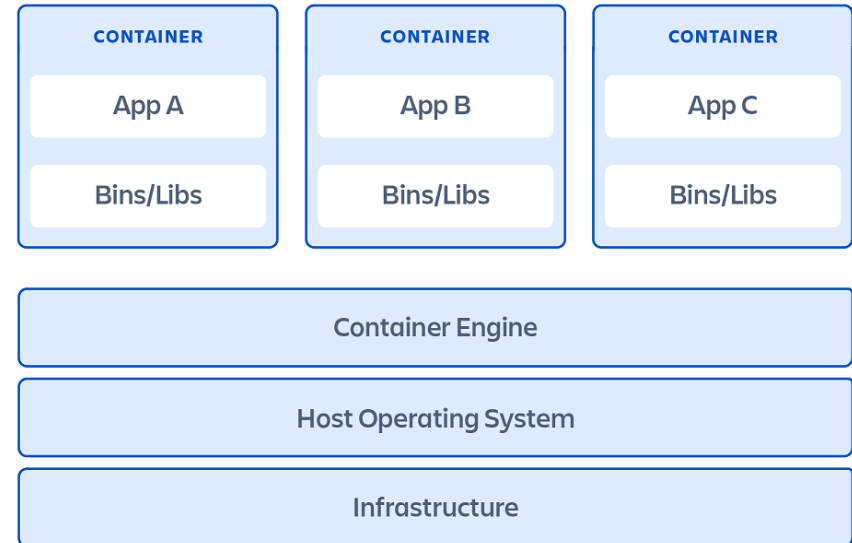
# Machine Virtuelle

- Un **hyperviseur** constitue la couche logicielle reliant l'ordinateur hôte et les machines virtuelles.
- Il supervise l'exécution des VM et gère l'allocation des ressources matérielles (CPU, RAM, Stockage et Réseau) entre les différentes VM.
- Un hyperviseur peut être :
  - de **type 1** : il s'exécute directement sur le matériel physique hôte.
  - de **type 2** : il s'exécute comme étant une application installée sur l'OS de la machine hôte.



# Conteneur

- Les **conteneurs** s'exécutent de manière indépendante et isolée sur le même système d'exploitation hôte via un **moteur de conteneurs**.
- Un conteneur encapsule une application avec l'ensemble des bibliothèques, des binaires et des fichiers nécessaires à son exécution.
- Le **moteur de conteneurs** gère un conteneur à travers une **image de conteneur** créées en exécutant un ensemble de commandes.



# VM versus Container

	VM	Container
Performance	chaque VM dispose de son propre OS ce qui implique une surcharge supplémentaire.	les conteneurs partagent le même noyau OS ce qui rend leur exécution plus légère et plus rapide.
Isolation	isolation totale.	isolation au niveau du processus d'exécution.
Infrastructure	Gérée par un hyperviseur comme VMware, Virtual Box, Microsoft Hyper-V, etc.	géré par un moteur de conteneur, comme Docker, et un orchestrateur comme Docker Swarm ou Kubernetes.
Portabilité	dépendance avec l'hyperviseur ➔ Transfert plus complexe d'un système à un autre.	dépendance avec le moteur de conteneurs ➔ Transfert plus léger d'un système à un autre.

# DOCKER

---

- Docker, qu'est-ce que c'est ?
- Image Docker & Conteneur Docker
- Dockerfile
- Ateliers Docker



# Docker, qu'est-ce que c'est ?

- **Docker** est un logiciel open source permettant d'automatiser le déploiement des applications sous formes de packages dans des **conteneurs** virtuels.



- les **conteneurs Docker** ne contiennent que les applications et leurs dépendances (i.e. le binaire avec toutes ses librairies) et partagent tous le même noyau du système d'exploitation hôte.

# Image Docker

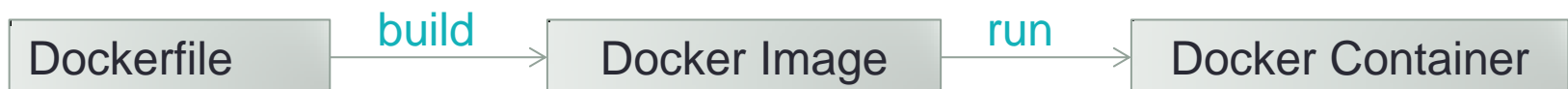
- Une **Image Docker** est un modèle, en lecture seule, composé de plusieurs couches qui représentent l'application que l'on souhaite déployer ainsi que les bibliothèques et les fichiers binaires requis.
- Exemple :



➔ Un **Conteneur Docker** est une instance d'une **Image Docker**.

# Dockerfile

- Un **Dockerfile** est un fichier texte qui définit une suite de commandes UNIX qui s'exécutent les unes après les autres pour créer une Image Docker.
- Un Dockerfile précise l'OS de base, les bibliothèques, les variables d'environnement, les emplacements des fichiers à charger, le port réseau, les commandes à exécuter, etc.
- Docker dispose de deux utilitaires :
  - **build** qui permet de créer une Image Docker à partir d'un Dockerfile
  - **run** qui permet de lancer le conteneur en instanciant l'image Docker



# Dockerfile - exemple

```
# ----- DÉBUT COUCHE OS -----  
FROM debian:stable-slim  
# ----- FIN COUCHE OS -----  
  
# MÉTADONNÉES DE L'IMAGE  
LABEL version="1.0" maintainer="mohamed.talha"  
  
# VARIABLES TEMPORAIRES  
ARG DOCUMENTROOT="/var/www/html"  
  
# ----- DÉBUT COUCHE APACHE -----  
RUN apt-get update -y && apt-get install apache2  
# ----- FIN COUCHE APACHE -----  
  
# ----- DÉBUT COUCHE MYSQL -----  
RUN apt-get install mariadb-server  
COPY db/articles.sql /  
# ----- FIN COUCHE MYSQL -----  
  
# ----- DÉBUT COUCHE PHP -----  
RUN apt-get \  
  php-mysql \  
  php && \  
  rm -f ${DOCUMENTROOT}/index.html && \  
  apt-get autoclean -y  
COPY app ${DOCUMENTROOT}  
# ----- FIN COUCHE PHP -----  
  
# OUVERTURE DU PORT HTTP  
EXPOSE 80  
  
# RÉPERTOIRE DE TRAVAIL  
WORKDIR ${DOCUMENTROOT}  
  
# DÉMARRAGE DES SERVICES LORS DE L'EXÉCUTION DE L'IMAGE  
ENTRYPOINT service mariadb start && mariadb < /articles.sql && apache2ctl -D FOREGROUND
```

# Les commandes Dockerfile (1/3)

- **FROM** : elle définit l'image de base qui sera utilisée par les instructions suivantes → chaque image est forcément basée sur une autre image. Une des images les plus utilisées est « **Alpine Linux** » qui est une distribution légère qui existe depuis 2006 et réputée pour sa sécurité. Le choix de la bonne image de base est primordial pour une bonne performance.
- **LABEL** : elle ajoute des métadonnées à l'image avec un système de clés-valeurs ; elle permet par exemple d'indiquer l'auteur du Dockerfile ou la version du fichier.
- **ENV** : elle permet de définir des variables d'environnements utilisables dans le Dockerfile et le conteneur.
- **ARG** : elle permet de définir des variables temporaires qu'on peut utiliser dans un Dockerfile.

# Les commandes Dockerfile (2/3)

- **RUN** : elle exécute des commandes Linux ou Windows lors de la création de l'image.
- **COPY** : elle permet de copier des fichiers depuis la machine locale vers le conteneur Docker.
- **ADD** : elle permet de copier des fichiers depuis la machine locale vers le conteneur Docker (tout comme COPY) mais prend en charge des liens ou des archives.
- **ENTRYPOINT** : comme son nom l'indique, c'est le point d'entrée du conteneur, en d'autres termes, c'est la commande qui sera toujours exécutée au démarrage du conteneur.
- **CMD** : elle spécifie les arguments qui seront envoyés à l'ENTRYPOINT.

# Les commandes Dockerfile (3/3)

- **WORKDIR** : elle définit le répertoire de travail à utiliser pour le lancement des commandes CMD et/ou ENTRYPOINT et ça sera aussi le dossier courant lors du démarrage du conteneur.
- **EXPOSE** : elle expose un port.
- **VOLUMES** : elle crée un point de montage qui permettra de persister les données.
- **USER** : elle désigne quel est l'utilisateur qui lancera les prochaines instructions RUN, CMD ou ENTRYPOINT (par défaut c'est l'utilisateur root).
- et bien d'autres... → <https://docs.docker.com/reference/dockerfile/>

# Ateliers Docker

- **Atelier 1** : Télécharger et installer Docker sur votre machine locale (ayant un OS récent) ou une machine virtuelle.
  - **Atelier 2** : Manipuler les images et les conteneurs Docker.
  - **Atelier 3** : Créer un Dockerfile, générer l'Image Docker et exécuter un conteneur Docker.
  - **Atelier 4** : Publier une image sur le repository Docker Hub.
- ➔ Suivre les 4 ateliers sur [GitHub Ateliers Docker](#)



# KUBERNETES

---

- K8s, qu'est-ce que c'est ?
- Atelier : minikube & kubectl
- Objets K8s, Workloads K8s, Cluster K8s
- Atelier : déployer une API REST Spring Boot sur K8s
- ConfigMap & Secrets K8s
- Atelier : sécuriser le code d'une API REST conteneurisée

# K8s, qu'est-ce que c'est ?

- **Kubernetes** (ou **K8s**) est un système open-source permettant d'automatiser le déploiement, la mise à l'échelle (scalabilité) et la gestion des applications conteneurisées.
- En d'autres termes, **K8s** est un orchestrateur de conteneurs → il permet de gérer le cycle de vie des conteneurs (déployer, exécuter, surveiller, mettre à l'échelle et coordonner).



## kubernetes

- Du fait de sa flexibilité, **K8s** est présent dans la plupart des fournisseurs Cloud :
  - ✓ Azure Kubernetes Service (**AKS**)
  - ✓ Amazon Elastic Kubernetes Service (**EKS**)
  - ✓ Google Kubernetes Engine (**GKE**)

# Atelier K8s : minikube et kubectl

- **minikube** est une version allégée de K8s déployable sur un poste de travail permettant d'éviter d'avoir un réseau de plusieurs machines pour déployer Kubernetes.
  - ➔ installer minikube en suivant les étapes de la doc officielle : <https://minikube.sigs.k8s.io/docs/start/?arch=%2Fwindows%2Fx86-64%2Fstable%2F.exe+download>
- **kubectl** est une CLI (**C**ommande **L**ine **I**nterface) très populaire utilisée pour interagir avec les clusters Kubernetes. Par exemple, il est possible d'afficher l'ensemble des pods d'un cluster K8s avec la commande suivante : `kubectl get pods`
  - ➔ Pour plus d'informations sur les commandes kubectl : <https://kubernetes.io/docs/reference/kubectl/>

# Atelier K8s : minikube et kubectl

```
C:\Users\Mohamed>minikube start
* minikube v1.34.0 sur Microsoft Windows 8.1 6.3.9600.20778 Build 9600.20778
* Choix automatique du pilote virtualbox
* Téléchargement de l'image de démarrage de la VM...
  > minikube-v1.34.0-amd64.iso....: 65 B / 65 B [-----] 100.00% ? p/s 0s
  > minikube-v1.34.0-amd64.iso: 333.55 MiB / 333.55 MiB 100.00% 3.01 MiB p/
* Démarrage du nœud "minikube" primary control-plane dans le cluster "minikube"
* Téléchargement du préchargement de Kubernetes v1.31.0...
  > preloaded-images-k8s-v18-v1....: 326.69 MiB / 326.69 MiB 100.00% 2.56 MiB p/
* Création de VM virtualbox (CPUs=2, Mémoire=4000MB, Disque=20000MB)...
! Failing to connect to https://registry.k8s.io/ from both inside the minikube VM and host machine
* Pour extraire de nouvelles images externes, vous devrez peut-être configurer un proxy : https://minikube.sigs.k8s.io/docs/reference/networking/proxy/
* Préparation de Kubernetes v1.31.0 sur Docker 27.2.0...
  - Génération des certificats et des clés
  - Démarrage du plan de contrôle ...
  - Configuration des règles RBAC ...
* Configuration de bridge CNI (Container Networking Interface)...
  - Utilisation de l'image gcr.io/k8s-minikube/storage-provisioner:v5
* Vérification des composants Kubernetes...
* Modules activés: storage-provisioner, default-storageclass
* kubectl introuvable. Si vous en avez besoin, essayez : 'minikube kubectl -- get pods -A'
* Terminé ! kubectl est maintenant configuré pour utiliser "minikube" cluster et espace de noms "default" par défaut.

C:\Users\Mohamed>minikube kubectl -- get pods
  > kubectl.exe.sha256: 64 B / 64 B [-----] 100.00% ? p/s 0s
  > kubectl.exe: 55.20 MiB / 55.20 MiB [-----] 100.00% 3.55 MiB p/s 16s
No resources found in default namespace.

C:\Users\Mohamed>minikube kubectl get pods
No resources found in default namespace.

C:\Users\Mohamed>
```

# Atelier K8s : minikube et kubectl

- Exécutez et interprétez quelques commandes :
  - minikube start
  - minikube status
  - minikube dashboard
  - minikube kubectl -- get pods -A
  - minikube kubectl -- get nodes
  - minikube ssh
    - docker version
    - docker image ls
    - docker run hello-world
    - docker images
    - exit
  - minikube stop

# OBJETS K8S

---

- POD
- VOLUME
- SERVICE
- NAMESPACE

# Objets K8s : PODS

- Kubernetes ne manipule pas directement les conteneurs, mais il les regroupe dans une structure de niveau supérieur appelés **POD**. Il s'agit de la plus petite entité dans un cluster K8s.
- Chaque pod possède :
  - **Une adresse IP** unique dans le cluster pour pouvoir communiquer avec les conteneurs qu'il héberge.
  - **Un espace de stockage** partagé à travers des volumes K8s, pour que ses conteneurs puissent échanger des données entre eux.
  - **Un ensemble de métadonnées** définissant son comportement, ses règles d'affinité et ses ressources allouées.
- Un pod peut contenir un ou plusieurs conteneurs, mais en pratique, un pod n'héberge qu'un seul conteneur principal.

# Objets K8s : PODS

- La configuration d'un POD se fait par le biais d'un **Manifest** qui est un fichier YAML (ou JSON) comprenant 4 sections :
  - **apiVersion** : la version
  - **kind** : Pod (d'autres type existent comme Deployment, ReplicaSet, etc.)
  - **Metadata** : par ajouter des informations (name, labels, etc.).
  - **spec** : pour spécifier les conteneurs à exécuter dans le pod
- Voici un exemple d'un Manifest basique (*pod-exemple.yaml*) :

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

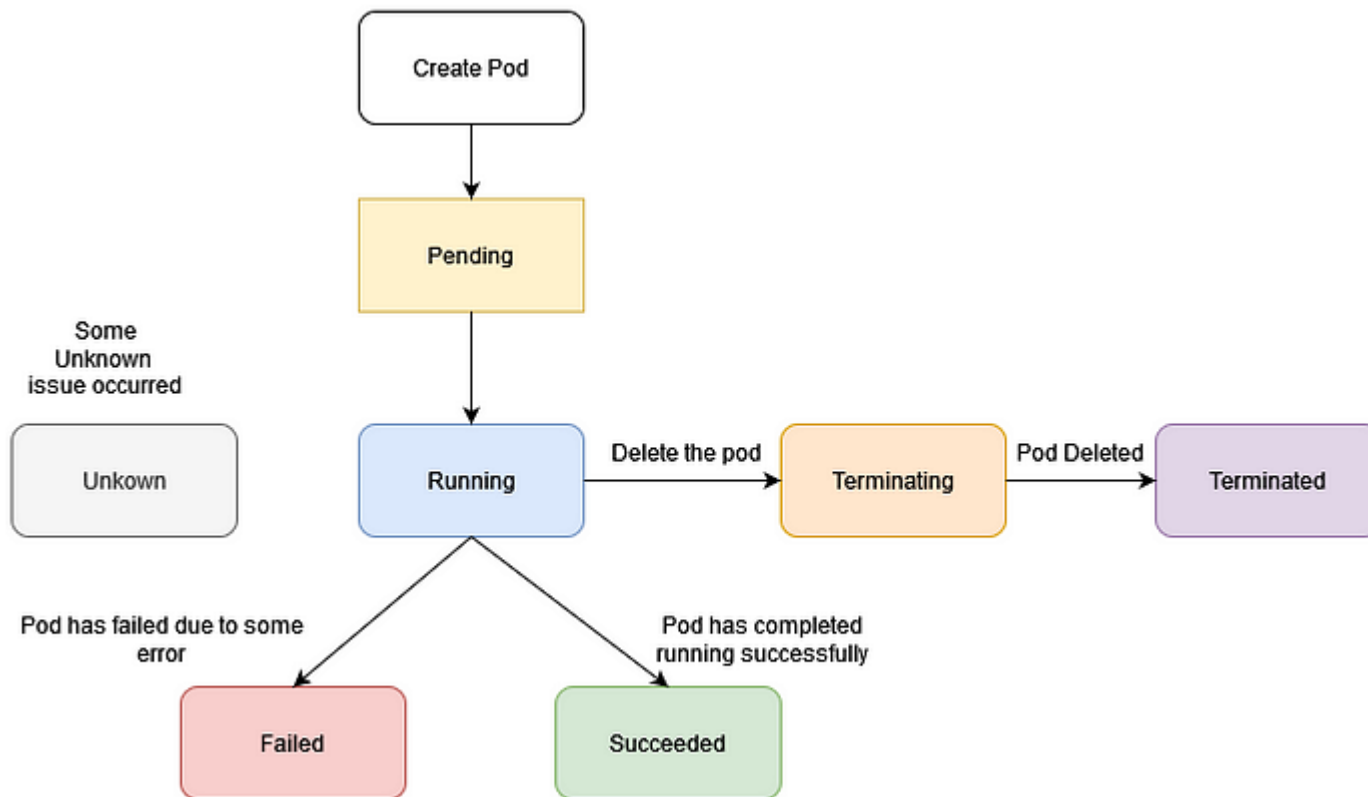
- Le Pod précédent peut être créé par la commande kubectl suivante :

```
kubectl apply -f pod-exemple.yaml
```



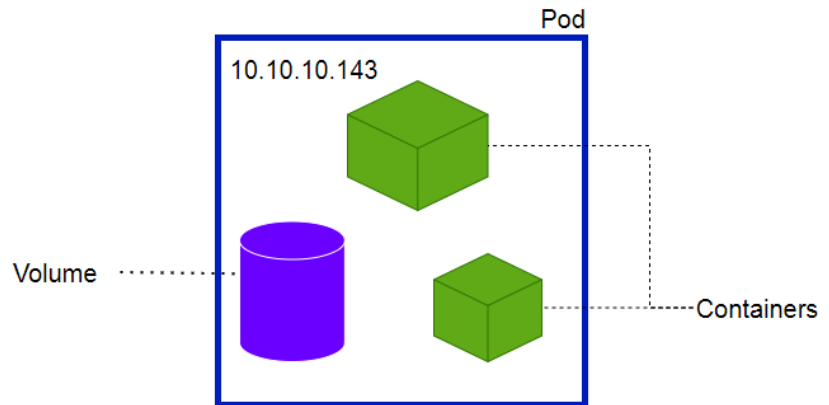
# Objets K8s : PODS


- Le cycle de vie d'un Pod est déterminé au travers d'un ensemble d'états (**Pod STATUS**) :



# Objets K8s : VOLUMES

- Un **Volume** K8s fournit un stockage qui persiste pendant toute la durée de vie du pod.
- Un Volume peut être utilisé comme espace disque partagé pour les conteneurs à l'intérieur du pod.



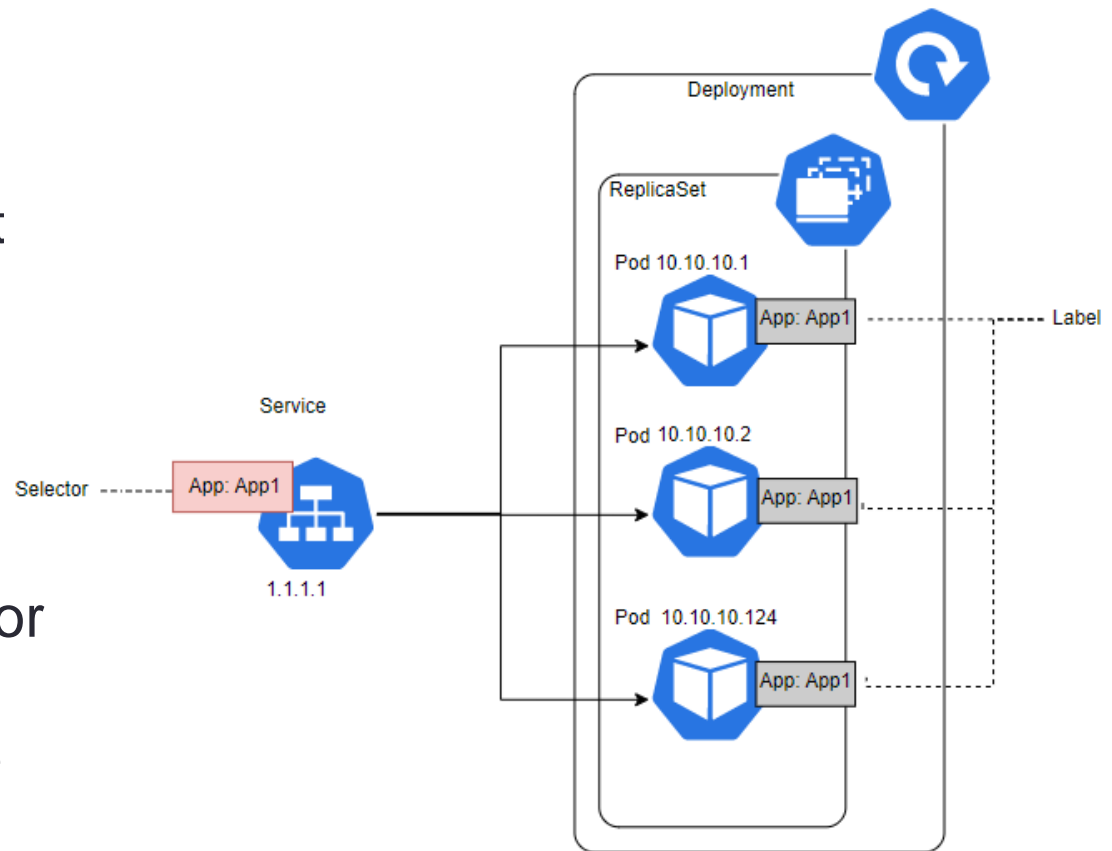
 Le redémarrage du pod effacera toutes ses données. Selon le besoin, on peut utiliser d'autres type de stockage K8s comme **Persistent Volume**, **config Map**, **Secret**, etc.

# Objets K8s : SERVICES

- L'accès à un conteneur se fait via l'adresse IP de son pod. Si ce dernier redémarre, un nouveau pod sera créé et aura une nouvelle adresse IP. Comment peut-on accéder au conteneur sans se soucier de l'adresse IP du pod ? → **Service K8s**.
- Un Service K8s permet aux pods de recevoir du trafic même après le redémarrage (changement d'adresse IP).
- Concrètement, pour cibler des pods, un service se base sur un **Selector** qui va chercher dans le cluster des objets possédant des paires **clé / valeurs** qui correspondent à celles qu'il attend.

# Objets K8s : SERVICES

- Le **Service**, exposé via son adresse IP 1.1.1.1, utilise un **Selector** qui surveille les pods ayant label "**App: App1**" et le status "**Running**".
- Cela permet au Selector de récupérer leurs IP pour permettre ensuite de les interroger.



# Objets K8s : NAMESPACES

- un **NameSpace** fournit un mécanisme pour isoler des groupes de ressources au sein d'un même cluster.
- Les namespaces sont utilisés dans des environnements avec de nombreux utilisateurs répartis sur plusieurs équipes ou projets, ou même séparant des environnements tels que le développement, l'intégration, la qualification et la production.
- La portée basée sur un NameSpace s'applique uniquement aux objets avec NameSpace (ex. pod, services, etc.) et non aux objets à l'échelle du cluster (ex. StorageClass, Nodes, Persistent Volume, etc.).

# WORKLOADS K8S

---

- Deployment
- ReplicaSet
- StatefulSet
- DaemonSet
- Job / CronJob

# Workloads

- Un Pod K8s a un cycle de vie défini. Lorsqu'un nœud du cluster subit une panne, tous les pods de ce nœud seront en échec. K8s traite cet échec en recréant automatiquement des nouveaux pods grâce à ses **Workloads** qui s'assurent du bon nombre de pods souhaités et que le bon type de pods soient en fonction.
- Kubernetes fournit plusieurs ressources workload :
  - Deployment
  - ReplicaSet
  - StatefulSet
  - DaemonSet
  - Job & CronJob

# Workloads K8s : Deployment

- Le **Deployment** est une bonne approche pour gérer une application stateless sur un cluster K8s où tous les Pods d'un Deployment sont interchangeables et peuvent être remplacés si besoin.
- Un Deployment représente un ensemble de **ReplicaSets** d'une application (une abstraction de haut-niveau) pour contrôler les pods.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```



# Workloads K8s : ReplicaSet

- Un **ReplicaSet** est le workload qui s'assure que le nombre de pods souhaités est égal à ceux en fonction (status RUNNING).
- Un ReplicaSet peut être utilisé pour gérer un pod, tout comme un Deployment, mais toute modification apportée au conteneur n'est pas prise en charge → il faudra créer manuellement un nouveau ReplicaSet.
- En revanche, toute modification apportée à un Deployment est appliquée automatiquement.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
        - name: php-redis
          image: gcr.io/google_samples/gb-frontend:v3
```

## Workloads K8s : StatefulSet, DaemonSet, Job

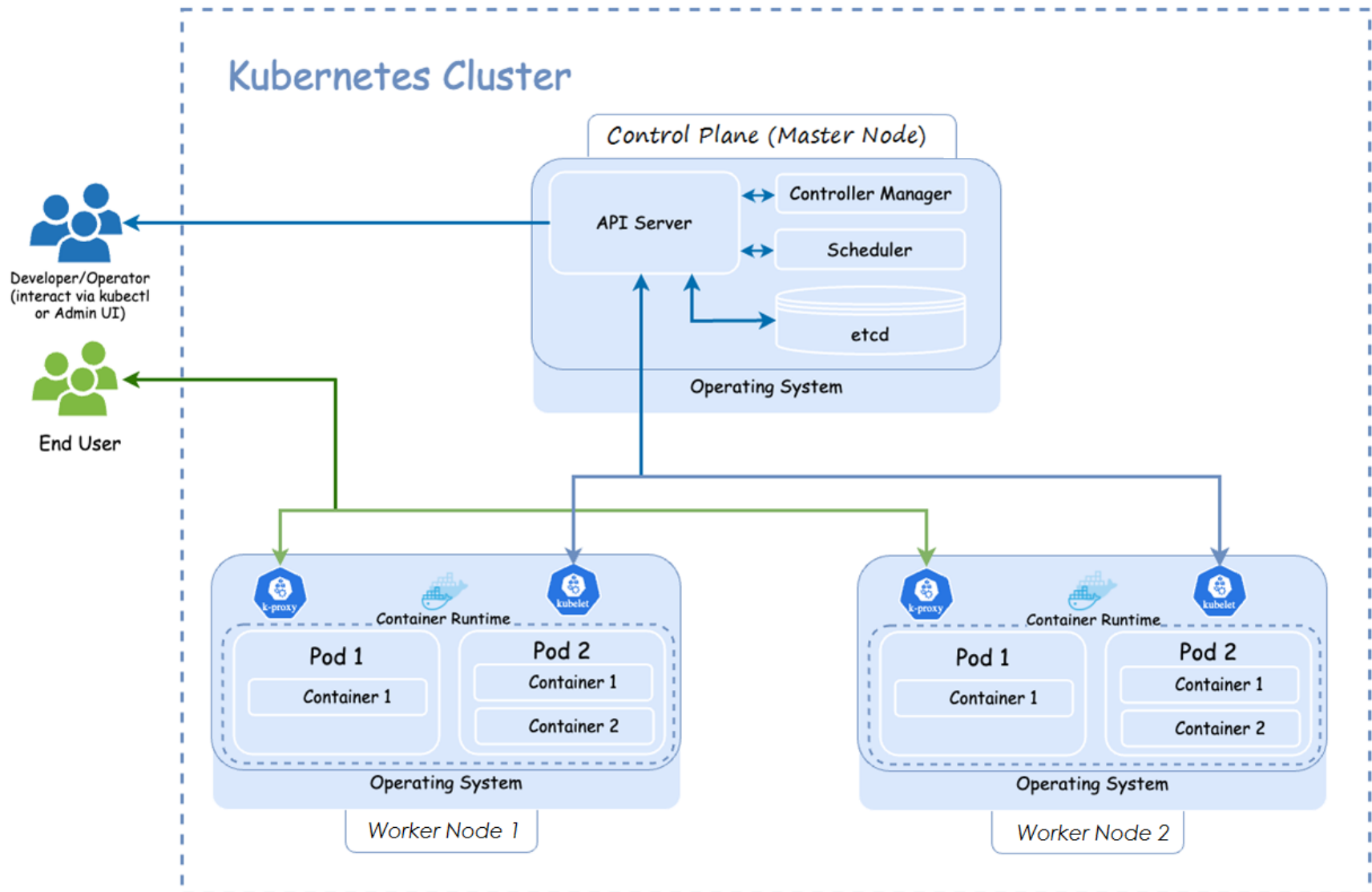
- Un **StatefulSet** est un workload qui permet de lancer un ou plusieurs Pods qui ***persistent*** des données. Le StatefulSet fait le lien entre le Pod et le Persistent Volume. [Voici un exemple](#).
- Un **DemonSet** gèrent les Pods qui effectuent des actions sur le nœud sur lequel ils tournent. On utilise DaemonSet le plus souvent pour gérer du stockage, du monitoring ou de la journalisation de log. [Voici un exemple](#).
- Un **Job** crée des Pods et s'assure qu'ils se terminent avec succès. Un Job est une tâche ponctuelle (exécutée une seule fois). Un **CronJob**, en revanche, est une tâche récurrente planifiée. Voici deux exemples : [Job](#) et [CronJob](#).

# CLUSTER K8S

---

- Architecture
- Control Plane
- Worker nodes

# Cluster K8s : Architecture



# Cluster K8s : Control Plane (Master)

- **Control Plane** est un ensemble de processus qui assignent les tâches, contrôlent les nœuds, s'assurent que les configurations décrites dans les différents manifests sont respectées et assurent les ressources pour que les conteneurs peuvent fonctionner correctement.
- Le Control Plane est composé de quatre éléments principaux :
  - **kube-apiserver** : c'est une API REST qui prend en charge les demandes internes et externes (via kubectl ou kubectl par exemple).
  - **kube-scheduler** : ce processus planifie l'attribution d'un pod à un nœud en fonction des besoins en ressources (CPU, Mémoire, etc.) du pod.
  - **kube-controller-manager** : c'est un processus qui combine un ensemble de contrôleurs (contrôleur de nœuds, de tâches, de réplication, etc.) dans le but de surveiller l'état du cluster en faisant appliquer la configuration des pods, des déploiements, des services et de toutes les ressources K8s.
  - **etcd** : c'est une base de données clé-valeur, hautement disponible, où sont stockées les données de configuration et les informations sur l'état du cluster.

# Cluster K8s : Worker nodes

- Les pods sont exécutés dans des **nœuds** de calcul en fonction des ressources requises pour chaque pod et celles disponibles dans chaque nœud. La mise à l'échelle d'un cluster K8s (scalabilité) consiste tout simplement à ajouter des nœuds.
- Dans un nœud on trouve :
  - des **pods** : un pod est une instance d'une application exécutant un ou plusieurs conteneurs étroitement couplés. Chaque pod est instancié à partir d'un Manifest, dispose d'une adresse IP, d'un stockage éphémère et peut se connecter à un stockage persistant.
  - **kubelet** : c'est un agent qui assure la communication entre le plan de contrôle et le nœud (exécuter les actions) et s'assure que les conteneurs des pods sont exécutés.
  - **Container Runtime** : c'est le logiciel utilisé pour exécuter les conteneurs comme Docker, RKT, LXC ou CRI-O.
  - **kube-proxy** : c'est un proxy réseau qui s'exécute sur chaque nœud du cluster, et qui aide à faire le routage du trafic réseau aux pods.

# Ateliers K8s

- **Atelier 1** : Installer minikube & kubectl
  - **Atelier 2** : Créer une simple API REST avec Spring Boot
  - **Atelier 3** : Créer un Dockerfile et générer l'image Docker
  - **Atelier 4** : Créer le Deployment et le Service
  - **Atelier 5** : Déployer et lancer l'API REST sur minikube
- ➔ Suivre les 5 ateliers sur : [GitHub Ateliers K8s](#)

# SÉCURITÉ KUBERNETES

---

- ConfigMap et Secret
- Atelier : intégrer un ConfigMap et un Secret à l'API REST
- Bonnes pratiques de gestion des Secrets K8s
- Atelier : gérer les Secrets K8s avec HashiCorp Vault
- Sécuriser un Cluster Kubernetes



# ConfigMap & Secret

- Pour sécuriser une application, les données sensibles ou secrètes ne doivent pas apparaître dans le dépôt de gestion de version (ni dans le code, ni en tant que variables d'environnement). Pour cela, Kubernetes offre une solution puissante à travers deux types d'objets : les **ConfigMaps** et les **Secrets**.
- Un **ConfigMap** est une ressource K8s dédiée au stockage des éléments de configuration sensibles mais non secrets (e.g. les adresses et les ports d'accès aux services externes d'une BDD ou une API REST), les niveaux de logs etc.
- Un **Secret** ressemble à un ConfigMap mais dédiée au stockage des valeurs secrètes (mot de passe, clé privée, etc.). La différence majeure étant que les données à l'intérieur d'un Secret sont encodées en base64.

# Ateliers K8s

- **Atelier 6** : Créer, déployer et utiliser un ConfigMap
  - **Atelier 7** : Créer, déployer et utiliser un Secret en tant que variable d'environnement
  - **Atelier 8** : Créer, déployer et utiliser un Secret en tant que fichier dans un volume
- ➔ Suivre les 3 ateliers sur : [GitHub Ateliers K8s](#)

# Secrets K8s : bonnes pratiques

- Les Secrets K8s permettent de stocker des informations sensibles mais, par défaut, ils sont encodés en base64, ce qui ne constitue pas un chiffrement. Il est donc essentiel d'adopter certaines bonnes pratiques de sécurité :
  - ✓ Changer régulièrement les Secrets.
  - ✓ Chaque pod doit avoir un rôle lui attribuant des privilèges selon lesquels il peut, ou pas, accéder aux Secrets (RBAC).
  - ✓ K8s stocke les Secrets en clair dans **etcd**. Il est donc recommandé d'activer le chiffrement des Secrets dans etcd.
  - ✓ Utiliser des solutions tierces de gestion des Secrets (comme HashiCorp **Vault**, AWS Secrets Manager, Azure Key Vault ou Google Cloud Key Management Service) qui offrent des fonctionnalités avancées pour la gestion des Secrets.

# HashiCorp Vault pour Kubernetes

- Les secrets Kubernetes sont stockés dans **etcd**, qui, même chiffrée au repos, peut ne pas offrir le niveau de sécurité et de contrôle d'accès requis pour les informations hautement sensibles. C'est ainsi que **Vault** s'impose comme une solution idéale pour stocker et gérer les informations sensibles.
- Vault offre des mécanismes robustes pour le chiffrement en tant que service et le contrôle d'accès dans un environnement Kubernetes.
- L'installation de Vault se fait généralement via **Helm** qui est un gestionnaire de paquets pour Kubernetes. Helm permet de définir, d'installer et de mettre à jour des applications Kubernetes sous forme de paquets via l'API Kubernetes.

# Ateliers K8s

- **Atelier 9** : Installer **Helm** et **HashiCorp Vault** sur le Cluster Kubernetes
  - **Atelier 10** : Gérer les Secrets K8s avec HashiCorp Vault
- ➔ Suivre les 2 ateliers sur : [GitHub\\_Ateliers\\_K8s](#)

# Références

- [https://github.com/Cloud-Elit/Docker\\_Kubernetes/tree/main](https://github.com/Cloud-Elit/Docker_Kubernetes/tree/main)
- <https://docs.docker.com/reference/dockerfile/>
- <https://containers.goffinet.org/>
- <https://kubernetes.io/fr/docs/home/>
- <https://helm.sh/fr/docs/>
- <https://developer.hashicorp.com/vault/docs>