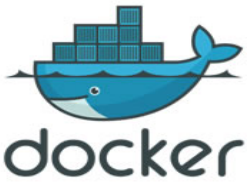




Kubernetes



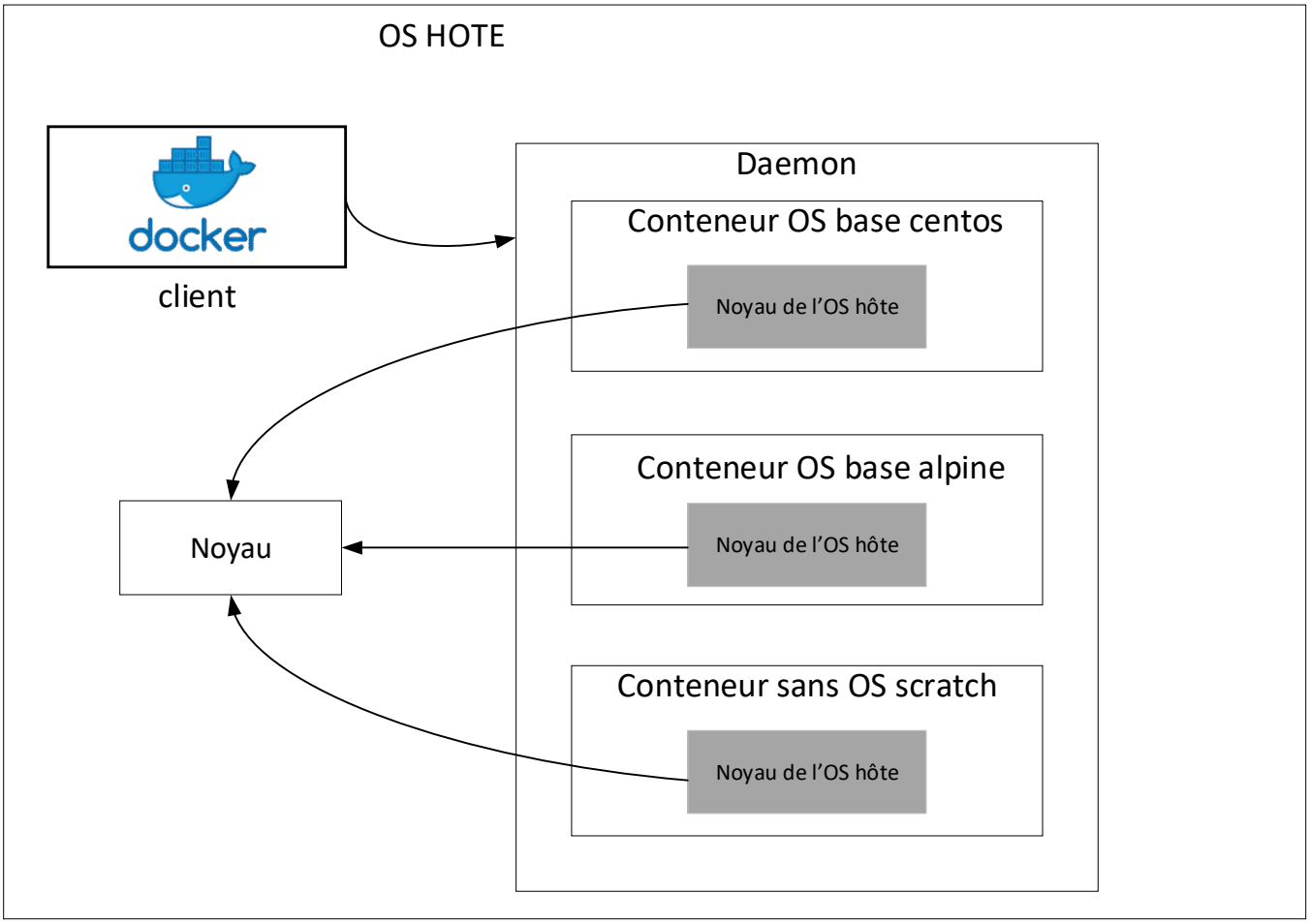


Notion sur Kubernetes

Kubernetes aussi connu comme **k8s** est un orchestrateur de conteneur. Il permet d'automatiser le déploiement, le démarrage et la mise en œuvre de conteneurs d'application sur des clusters de serveurs. Avant d'aborder l'infrastructure logicielle de kubernetes faut-il encore comprendre ce qu'est un conteneur.

1) Les conteneurs

Un conteneur permet d'embarquer un applicatif auquel on a rajouté toutes ses dépendances (librairies, fichier de configuration, Les variables d'environnement …). Les conteneurs sont donc une abstraction au niveau de la couche d'application qui regroupe le code et les dépendances. Plusieurs conteneurs peuvent s'exécuter sur la même machine et partager le noyau du système d'exploitation hôte avec d'autres conteneurs, chacun s'exécutant en tant que processus isolés dans un espace utilisateur du noyau du système d'exploitation hôte. Le processus de conteneur est isolé (espaces de noms) du reste du système et restreint à la fois du point de vue de la consommation des ressources (cgroups) et de la sécurité.

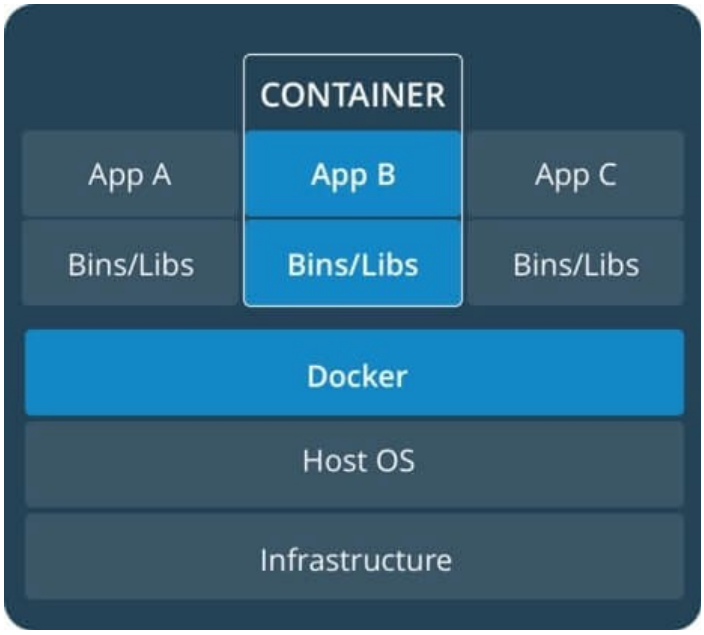


Le démon conteneur s'exécute dans l'espace utilisateur du système d'exploitation hôte et traduit tous les appels système du système d'exploitation des conteneurs en appels système du système d'exploitation hôte et vice versa. En bref, dans les conteneurs, l'espace noyau de leur système d'exploitation est émulé alors que dans les machines virtuelles, il ne l'est pas.

Mais en fin de compte, il s'agit toujours d'un processus régulier, comme tout autre processus sur le système hôte. les processus conteneurisés reçoivent souvent un système de fichiers racine grâce à un OS de base et un ensemble d'utilisateurs complètement différents, ce qui donne presque l'impression qu'ils s'exécutent sur une machine distincte (Mais ce n'est pas le cas; il partage toujours le processeur, la mémoire, la bande passante d'E / S de l'hôte et, surtout, le noyau Linux de l'hôte**).** Les conteneurs sont donc isolés du système d'exploitation hôte et de son système de fichiers.

Docker en coulisse utilise le **système d'exploitation hôte** qui est Linux lui-même pour exécuter ses conteneurs. Les programmes inclus dans l'instantané de système de fichiers de type Linux que nous voyons sous la forme de conteneurs docker **s'exécutent en fait sur le système d'exploitation hôte de manière isolée**.

Le runtime de conteneur (ici docker) est le logiciel installé sur un système hôte qui lui permet d'isoler ses ressources pour les conteneurs, d'extraire des images de conteneur et de gérer le cycle de vie des conteneurs.



2) Les images

Un conteneur Docker est une instance en cours d'exécution **d'une image Docker**. En termes simples, l'image Docker est extraite d'un registre et exécutée en tant que conteneur.

Une image de conteneur docker a le plus souvent une sorte d'instantané de système de fichiers de l'image linux dont dépend **l'image** de conteneur.
L'image conteneur comprend des programmes de base comme bash-shell, vim-editor, etc. pour faciliter le développement à travailler facilement avec l'image docker. En outre, les images docker peuvent inclure des dépendances préinstallées telles que nodeJS, redis-server, etc., comme nous pouvons le trouver sur docker hub.

Les images définissent l'état initial du système de fichiers des nouveaux conteneurs. Ils regroupent le code source de votre application et ses dépendances dans un package autonome prêt à l'emploi avec un runtime de conteneur. Dans l'image, le contenu du système de fichiers est représenté sous la forme de plusieurs couches indépendantes.

Une image docker est créée à partir d'un fichier nommé le **Dockerfile**.

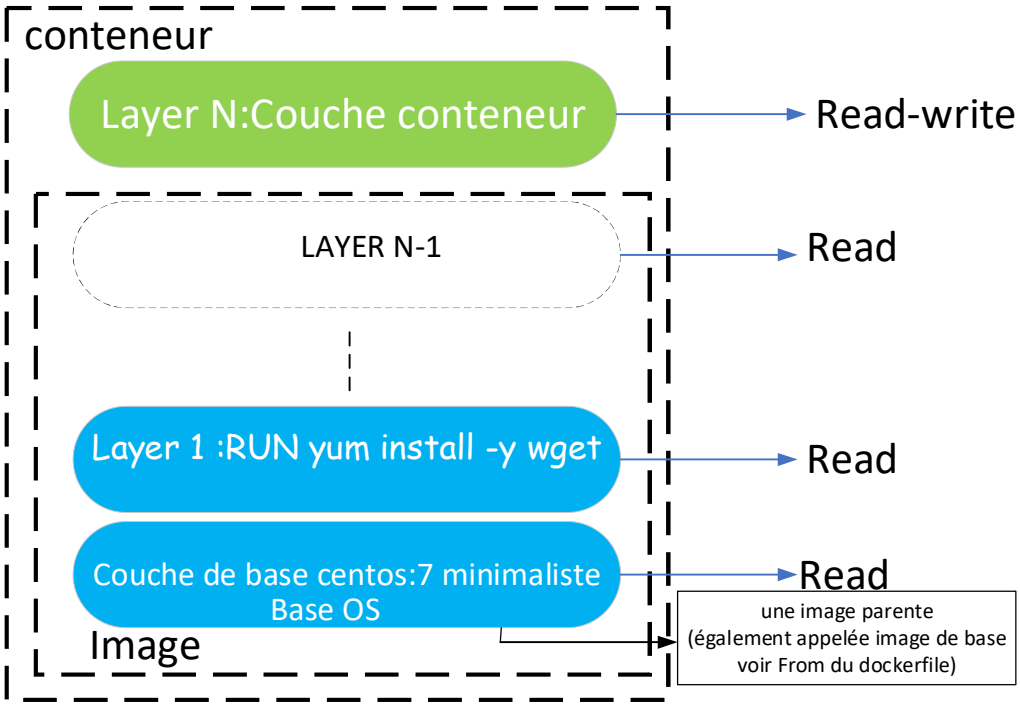
Les calques sont le résultat de la façon dont les images Docker sont construites. Chaque étape d'un Dockerfile crée une nouvelle « couche » qui est essentiellement une différence des modifications apportées au système de fichiers depuis la dernière étape.

```
FROM node:centos:7
RUN yum install -y wget
RUN apt-get update
RUN apt-get install -y build-essential
WORKDIR /app
COPY app/hello.c /app/
RUN gcc -o hello hello.c
ENTRYPOINT [ "/app/hello" ]
```

Dans Ce Dockerfile, le processus de fabrication en couche de l'image est la suivante :

- Commencez avec une image de base assez petite et allégée sans noyau
- Ajouter les paquets dont j'ai (approximativement) besoin
- Ajouter mon code source
- Construire
- exécuter

Chaque ligne de ce Dockerfile est une couche Docker et, si elle n'est pas modifiée, elle sera réutilisée à partir des couches mises en cache dans les versions ultérieures. Le diagramme ci-dessous montre un conteneur basé sur une image.



La ligne FROM et l'OS fournit le système de fichiers initial, c'est-à-dire les fichiers, les répertoires, les logiciels préinstallés, etc. pour la nouvelle image.
Vous avez besoin de programmes qui à leur tour ont besoin de bibliothèques, etc. Ceux-ci viennent sous forme de fichiers dans l'image docker et sont appelés en cas de besoin.
il est maintenant devenu un modèle très courant, d'emballer simplement l'ensemble du système de fichiers à partir d'un système d'exploitation donneur (base OS) dans un conteneur, car de cette façon, vous savez que toutes les dépendances du processus à exécuter seront présentes. Cela ajoute également la commodité de signifier que des outils de gestion de paquets seront présents, ce qui facilite la construction du conteneur. Cela signifie également que des utilitaires tels que des shells sont également présents, ce qui facilite le débogage d'un conteneur
l'OS de base lié à FROM peut être beaucoup plus petit qu'une image de système d'exploitation conventionnelle.

Le Système d'exploitation de base implique une pile de bibliothèques, d'utilitaires et de programmes qui s'exécutent au-dessus du noyau. Docker vous permet d'exécuter un espace utilisateur Ubuntu, ou un espace utilisateur Alpine, ou un autre espace utilisateur... ou pas d'espace utilisateur du tout.

Dans certains cas, le processus conteneur n'a pas de dépendances au-delà du noyau. Dans ces cas, un os de base spécial peut être utilisé qui n'ajoute rien d'autre. C'est une image vide le FROM: scratch
Si vous voulez faire une image à partir de rien, vous pouvez commencer votre dockerfile avec:

```
FROM scratch
```

Une image comporte plusieurs calques. Lorsqu'un conteneur démarre, une seule couche de lecture-écriture est attachée au-dessus de toutes les couches d'images.
Les couches sont empilées les unes sur les autres. Lorsque vous créez un conteneur, vous ajoutez un nouveau calque accessible en écriture au-dessus des calques sous-jacents. Cette couche est souvent appelée « couche conteneur ».
Toutes les modifications apportées au conteneur en cours d'exécution, telles que l'écriture de nouveaux fichiers, la modification de fichiers existants et la suppression de fichiers, sont écrites dans cette couche de conteneur inscriptible mince.

La principale différence entre un conteneur et une image est le calque supérieur inscriptible.
Les conteneurs sont isolés du système d'exploitation hôte et de son système de fichiers.

Toutes les couches créées par un Dockerfile sont en lecture seule – elles sont immuables. Cela est nécessaire, afin qu'ils puissent être réutilisés.
Lorsque vous démarrez un conteneur, Docker prend tous les calques de votre image et en ajoute un nouveau par-dessus C'est le calque de lecture-écriture et celui contenant toutes les modifications que vous apportez à votre système de fichiers: modifications de fichiers, ajouts de fichiers, suppressions de fichiers.
Si vous supprimez un fichier sur votre conteneur - vous ne modifiez aucun des calques d'image, vous ajoutez simplement une « note » sur votre couche de lecture-écriture indiquant « ce fichier a été supprimé »

Les images docker sont stockées et distribuées dans un Registry qui est une application côté serveur le Docker Hub Registry (registre officiel de Docker). Ce serveur et les images disponibles sont consultable à l'adresse suivante :

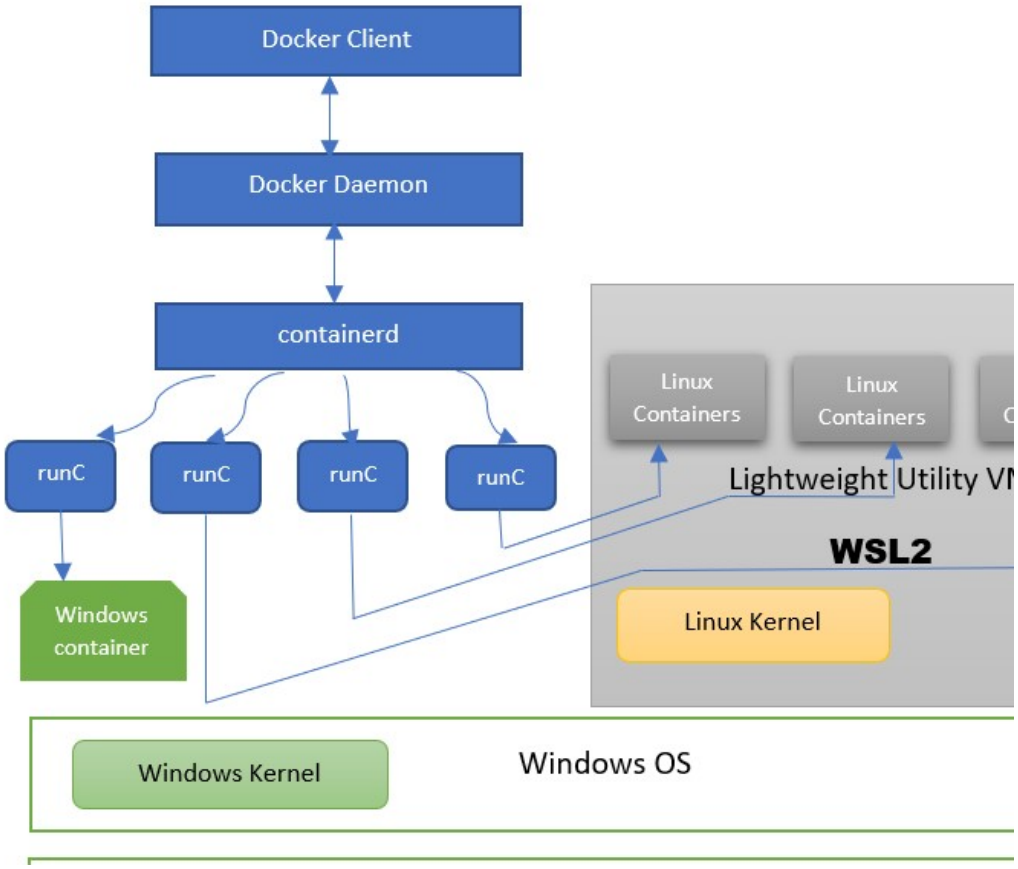
https://hub.docker.com/.

La commande **docker pull** permet de retirer une image de ce serveur



3) Les conteneurs sous Windows

Pour utiliser les containers sous Windows, il est possible d'utiliser le client Docker Desktop. Dans ces containers, on peut faire tourner des applications Windows. Néanmoins, il ne permet pas d'exécuter des containers Linux directement sur Windows, à cause des dépendances et de l'absence du noyau Linux. Lorsque l'on est sur Linux, on peut exécuter des applications Linux au sein de containers. Depuis Windows 10, nous pouvons exécuter des containers Linux grâce à une distribution Linux qui tourne avec WSL 2 sans être obliger de monter une VM Linux.Pour exécuter Docker sous Windows, vous devez d'abord installer le sous-système Windows pour Linux (WSL)sorte de VM.



4) Avantages des conteneurs

Cela réduit la complexité grâce:

- ✓ A l'**intégration** de toutes les dépendances avec tout ce qu'il faut pour que notre applicatif fonctionne
- ✓ L'isolation des containers permet d'éviter que les logiciels cohabitent entre eux. (Les dépendances ne rentrent pas en conflit).
- ✓ Ces boîtes sont fermées (on peut ne peux plus les modifier). On parle d'**immutabilité**
- ✓ L'agrandissement d'und infrastructure logiciel est beaucoup plus facile lorsqu'on a des boîtes autonomes qu'on peut répliquer et déployer.
- ✓ Les conteneurs ont un encombrement beaucoup plus faible et sont donc beaucoup plus rapides à remonter (et à démonter) que les machines virtuelles. les conteneurs partagent simplement le noyau du système d'exploitation du système hôte

5) Kubernetes

5.1) Présentation

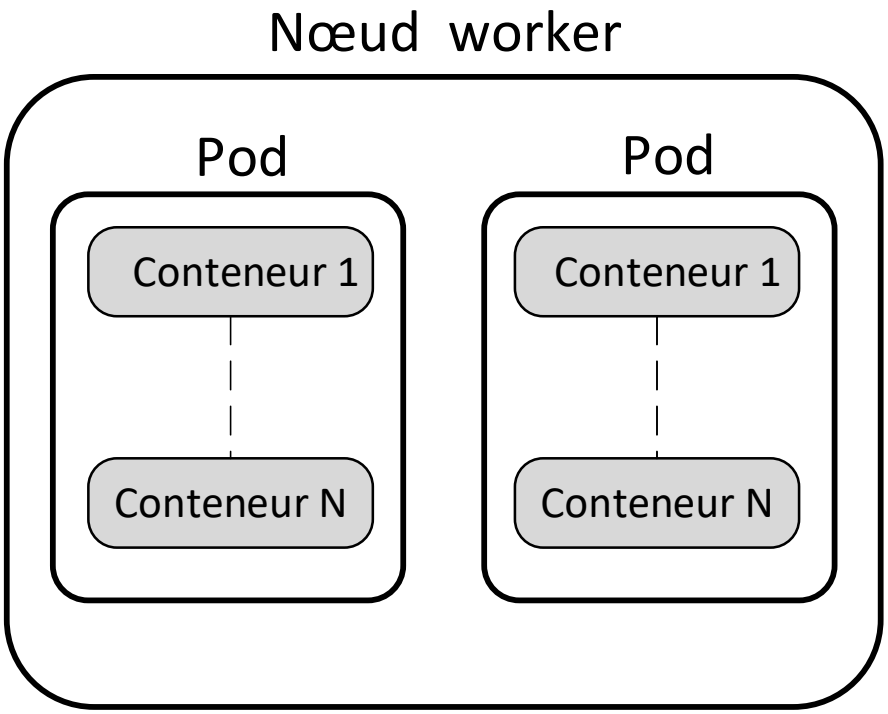
Kubernetes est un orchestrateur de conteneur. Il permet d'automatiser le déploiement, le démarrage et la mise en œuvre de conteneurs d'application sur des clusters de serveurs.

Lorsque Kubernetes a été publié pour la première fois, il utilisait le runtime Docker comme valeur par défaut codée en dur. Au fur et à mesure que la plate-forme grandissait, le besoin de prendre en charge des environnements d'exécution alternatifs augmentait également. Pour rendre Kubernetes plus indépendant du runtime, l'interface d'exécution du conteneur a été créée la CRI (Container Runtime Interface).

5.2) Architecture logicielle interne

Un système qui exécute des conteneurs ne le fait qu'en raison de son exécution de conteneur. Le runtime de conteneur est le logiciel installé sur un système hôte qui lui permet d'isoler ses ressources pour les conteneurs, d'extraire des images de conteneur et de gérer le cycle de vie des conteneurs. En ce qui concerne Kubernetes, un runtime de conteneur est requis sur chaque nœud de votre cluster. Sans cela, les pods dans Kubernetes sont inexistants.

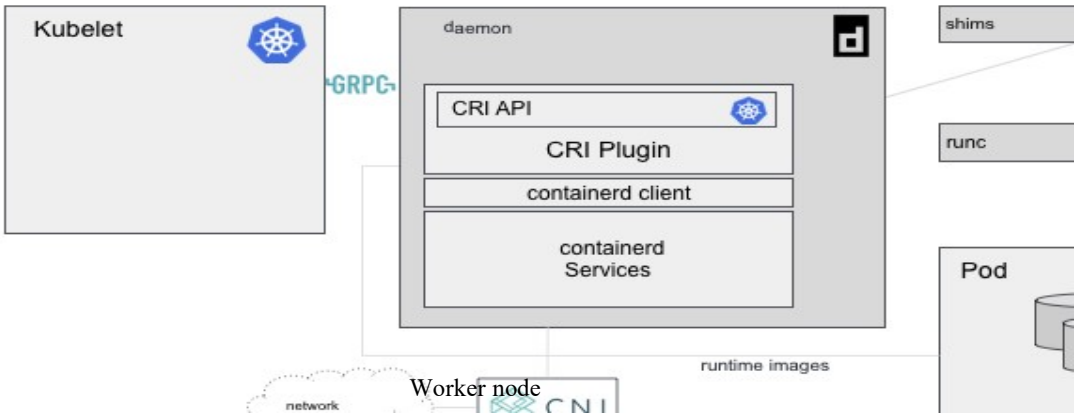
un pod est la plus petite unité de référence dans Kubernetes. Chaque pod gère un ou plusieurs conteneurs, qui forment ensemble une seule unité fonctionnelle.



L'OCI (Oracle Cloud Infrastructure) fournit des spécifications qui doivent être implémentées par les moteurs d'exécution de conteneur. Il s'agit d'une spécification de haut niveau axée sur l'orchestration de conteneurs. CRI ou IRC (Container Runtime Interface) traite d'aspects supplémentaires de la gestion des conteneurs, tels que la gestion des images, les instantanés et la mise en réseau, tout en déléguant l'exécution du conteneur à un environnement d'exécution conforme à OCI.

Les runtimes de conteneur qui adhèrent à la spécification IRC (Container Runtime Interface) sont les suivants :

- Dockershim était un composant de Kubernetes qui a ajouté l'abstraction CRI requise devant Docker Engine pour que Kubernetes reconnaisse Docker Engine comme compatible CRI. Il a été déprécié.
- containerd est le moteur d'exécution de conteneur le plus populaire, et il est géré, développé et utilisé par Docker. Il utilise runC sous le capot pour l'exécution du conteneur.
- CRI-O a été spécialement conçu pour Kubernetes par Red Hat. CRI-O est une implémentation CRI qui permet d'utiliser n'importe quel runtime compatible OCI.



Kubernetes suit l'architecture maître-esclave, le maître plus communément appelé node **master** existe principalement pour gérer votre cluster Kubernetes. . Les esclaves sont quant à eux plus connus sous le nom de **workers nodes ou agent**(on les appellent aussi **minions**) et ne sont là que pour fournir de la capacité et n'ont pas le pouvoir d'ordonner à un autre workers node ce qu'il peut ou ne peut pas faire. Les composants clés du master et worker sont définis dans la section suivante.

Sur un node de type master, nous aurons les composants suivants :

kube-apiserver : point d'entrée exposant l'API HTTP Rest de k8s depuis le maître du cluster Kubernetes. Différents outils et bibliothèques peuvent facilement communiquer avec l'API.

kube-scheduler : Il est responsable de la répartition et l'utilisation de la charge de travail sur les noeuds du cluster selon les ressources nécessaires et celles disponibles.

kube-controller-manager : ce composant est responsable de la plupart des collecteurs qui récupèrent des informations du cluster tout en effectuant des actions de correctives en cas de besoin, en apportant des modifications pour amener l'état actuel du serveur à l'état souhaité. Il est composé de plusieurs contrôleurs, on peut par exemple retrouver un contrôleur de réplication qui va s'assurer que vous avez le nombre désiré de répliques sur vos pods, mais aussi d'autres contrôleurs clés comme , le contrôleur de Endpoints, le contrôleur d'espace de noms et le contrôleur de compte de service.

cloud-controller-manager : effectue les mêmes actions que le **kubecontroller- manager** mais pour des fournisseurs de cloud sous-jacents (AWS, Azure, Google Cloud Platform, etc …).

etcd : il stocke les informations de configuration pouvant être utilisées par chacun des noeuds du cluster. Ces informations sont conservées sous forme de clé et valeurs à haute disponibilité

kubedns : Kubernetes exécute également un serveur DNS, qui fournit un nommage et une fonctionnalité de découverte pour les services définis dans le cluster. Ce serveur DNS s'exécute également en tant que service répliqué sur le cluster.

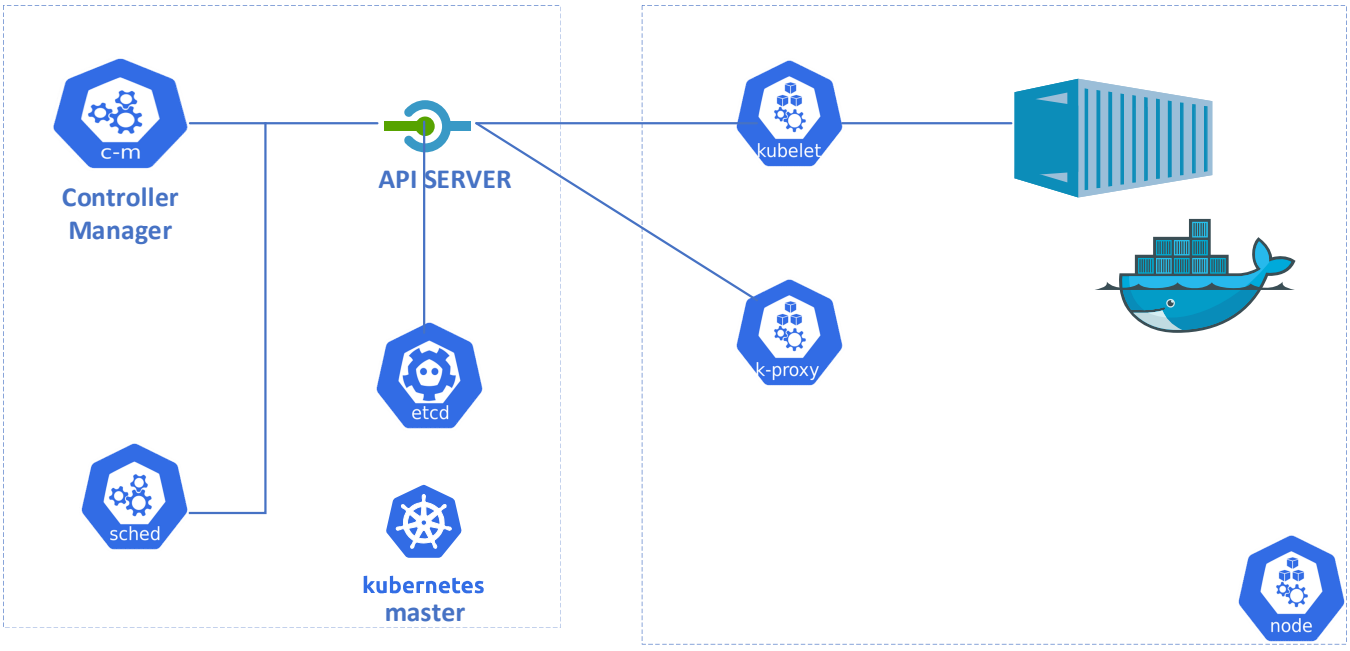
Selon la taille de votre cluster, vous pouvez voir un ou plusieurs serveurs DNS qui s'exécutent dans votre cluster. Le service DNS est exécuté en tant que déploiement Kubernetes

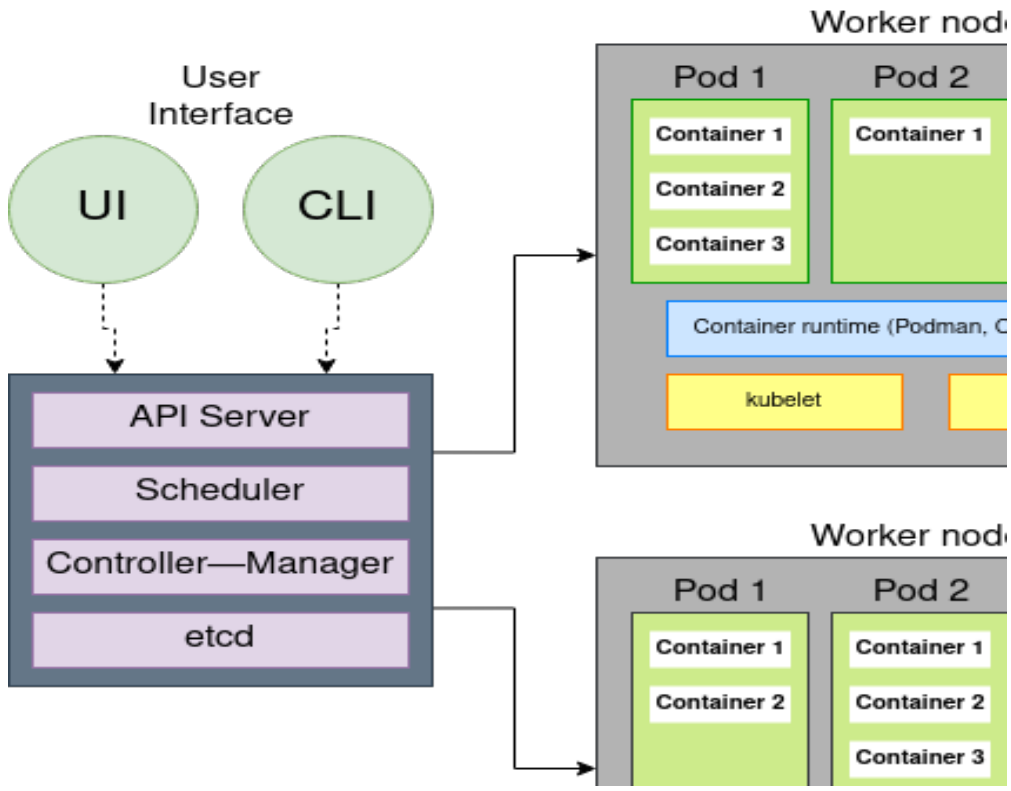
Sur chaque worker node ou agent, nous aurons les composants suivants :

kubelet : Il s'agit d'un agent qui s'exécute dans chaque noeud chargé de relayer les informations au Master. Il interagit avec la base de données **etcd** du Master pour récupérer des informations afin de connaître les tâches à effectuer. Il assume la responsabilité de maintenir en bon état de fonctionnement les conteneurs d'un pod et s'assure qu'ils tournent conformément à la spécification. Il ne gère pas les conteneurs qui n'ont pas été créés par Kubernetes. Il communique avec le Master et redémarre le conteneur défaillant en cas de crash.
Les adresses IP du cluster sont des adresses IP virtuelles stables qui équilibrent la charge du trafic sur tous les points de terminaison d'un service. Ce miracle est accompli par un composant fonctionnant sur chaque noeud du cluster et qui se nomme kube-proxy.

kube-proxy : kube-proxy surveille les nouveaux services dans le cluster via le serveur d'API. Il programme ensuite une série de règles iptables dans le noyau de cet hôte pour réécrire la destination des paquets de telle sorte qu'ils soient dirigés vers l'un des points de terminaison de ce service. Si l'ensemble des points de terminaison d'un service change (en raison des mouvements des pods ou en raison d'un échec de la vérification de la disponibilité), l'ensemble des règles iptables est réécrit.
L'adresse IP du cluster lui-même est généralement assignée par le serveur d'API quand le service est créé. Toutefois, lors de la création du service, l'utilisateur peut choisir une adresse IP de cluster spécifique. Une fois qu'elle est définie, l'IP du cluster ne peut pas être modifiée sans supprimer et recréer l'objet Service.

Si on reprend tous les composants vus précédemment, nous aurons alors un





5.3) création d'un objet sous kubernetes

Kubernetes utilise un template au format YAML pour créer ces différents objets qui commence toujours par les deux lignes suivantes:

```
apiVersion: v1
kind: PersistentVolumeClaim
……
……
---
apiVersion: apps/v1
kind: deployment
……
……
```

Une définition d'objet dans Kubernetes nécessite un champ **apiversion**. Cependant, la documentation officielle de Kubernetes fournit peu d'indications sur apiVersion. Prenons les deux plus connus :

apps/v1
est le groupe d'API le plus courant dans Kubernetes, avec de nombreux objets principaux en étant tirés et . Il inclut des fonctionnalités liées à l'exécution d'applications sur Kubernetes, telles que Deployments, RollingUpdates et ReplicaSets

certificates.k8s.io/v1beta1
Cette version de l'API ajoute des fonctionnalités pour valider les certificats réseau pour une communication sécurisée dans votre cluster.

Le champ **kind** spécifie le type d'objet que vous souhaitez créer (pod, replicaset , deployments …)

Les lignes suivantes du template sont liées au type d'objet utilisé

Chaque création d'objet dans le template est séparé par trois moins

5.4) Communication avec le cluster

Pour communiquer avec le cluster Kubernetes, on peut utiliser la commande **kubectl**, ou une interface web de management nommé **Dashboard**.

kubeclt (ligne de commande)

L'outil en ligne de commande Kubernetes, kubectl, vous permet d'exécuter des commandes sur les clusters Kubernetes afin de créer et gérer des objets k8s ou k3s et interagir avec l'API Kubernetes. Vous pouvez utiliser kubectl pour déployer des applications, inspecter et gérer les ressources du cluster et afficher les logs. Nous aurons l'occasion d'utiliser cet outil dans les chapitres suivants.

Dashboard (interface web)

Dashboard est l'interface utilisateur Web officielle de Kubernetes. Il fournit une vue d'ensemble des applications en cours d'exécution sur votre cluster Kubernetes. Il permet aux administrateurs de surveiller et gérer les opérations de leurs clusters Kubernetes, dépanner les applications conteneurisées et gérer les ressources du cluster.

5.4) L'objet Volume

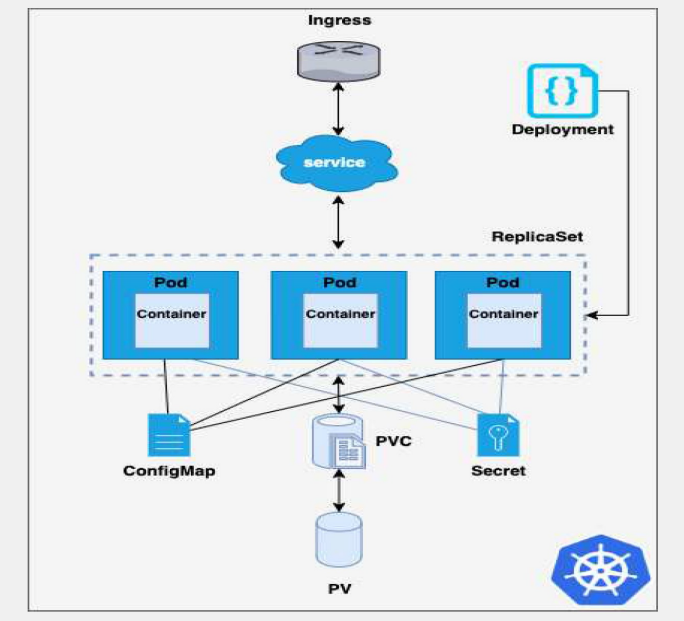
5.4.1) Les différents stockages

Les données dans un conteneur sont éphémères. Lorsqu'un conteneur atteint sa fin de vie, ses fichiers sont effacés. Il est parfois nécessaire de sauvegarder des données afin qu'elles restent disponibles même après l'arrêt du conteneur.

Dans Kubernetes, un volume peut être considéré comme un répertoire accessible aux conteneurs d'un pod. Il existe différents types de volumes dans Kubernetes, ce type va définir la façon dont le volume va être créé ainsi que son contenu. Kubernetes prend en charge différents types de stockage dans lesquels le pod peut utiliser plusieurs d'entre eux en même temps.

Voici une liste - non exhaustive - des principaux types de stockage :

- **hostPath** : Ce type de volume monte un fichier ou un répertoire du système de fichiers du nœud hôte dans votre pod.
- **emptyDir** : Il s'agit d'un type de volume créé lors de la première affectation d'un pod à un nœud. Il reste actif tant que le pod s'exécute sur ce nœud. Le volume est initialement vide et les conteneurs du pod peuvent lire et écrire les fichiers dans le volume emptyDir. Une fois le pod retiré du nœud, les données du répertoire vide sont effacées.
- **secret** : Un volume *secret* est utilisé pour transmettre des informations sensibles, telles que des mots de passe, à des pods.
- **ConfigMap** : La ressource **ConfigMap** fournit un moyen d'injecter des données de configuration dans les pods.
- **persistentVolumeClaim** : Un volume **persistentVolumeClaim** est utilisé pour monter un PersistentVolume dans un pod. Les volumes persistants sont un moyen pour les utilisateurs de «revendiquer» un stockage durable (tel qu'un disque persistant GCE ou un volume iSCSI) sans connaître les détails de l'environnement cloud en particulier.



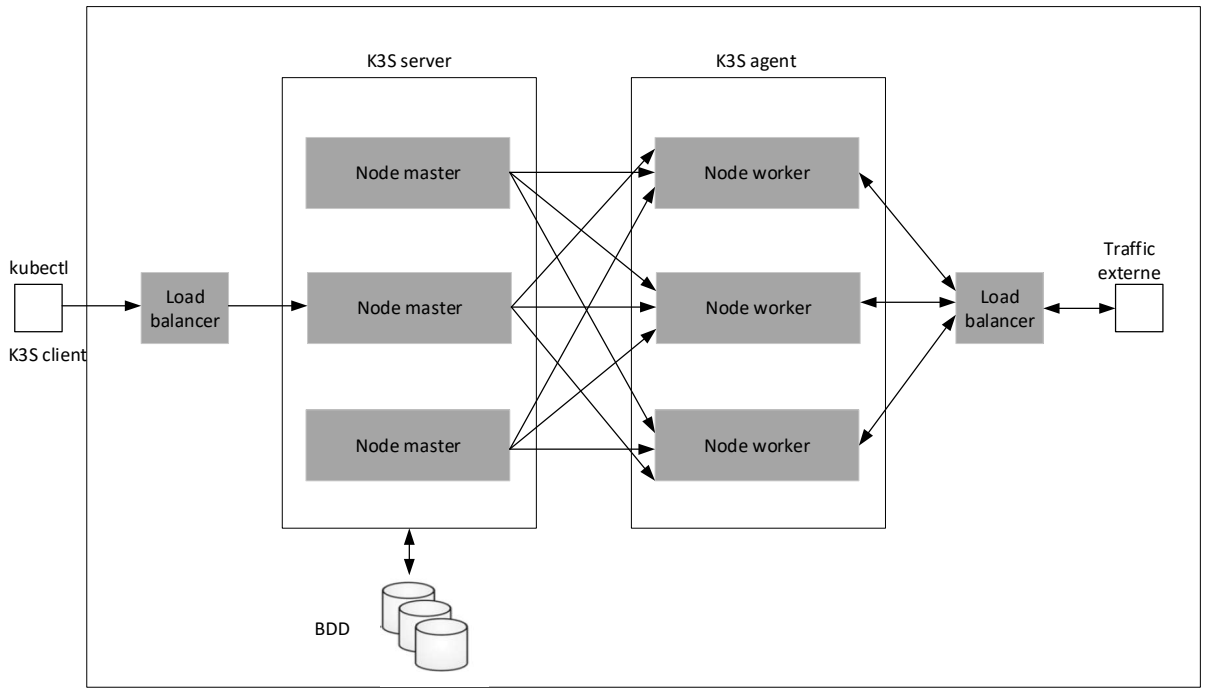
K3s est distribution Kubernetes légère. Particulièrement adaptée pour l'IoT, le edge computing. En effet, k3s tient dans un binaire d'à peine 50Mo et peut donc être déployée extrêmement rapidement.

Sont inclus :

- Traefik
- metrics-server
- flannel
- klipper-lb
- Local path provisioner

Ces deux derniers éléments permettent de disposer du type loadbalancer pour les Services ainsi que d'une StorageClass fonctionnelle utilisant le stockage local de votre machine. Ces deux éléments font partis des points rencontrés par la plupart des équipes ayant tenté l'aventure de Kubernetes en dehors d'un cloud public.

Jusqu'à la version 1.19, K3S était livré avec Dqlite, plus tard, ils sont passés à etcd pour la base de données embarquée.v1.19.1 pour gérer tous les déploiements et l'état général du cluster. Parfois vous devrez peut-être découpler la base de données et demander à K3S d'utiliser une base de données externe , par exemple, si vous essayez de configurer un cluster Kubernetes hautement disponible.



5.4.2) Le stockage sous K3S

Les données dans un conteneur sont éphémères, c'est-à-dire que lorsqu'un conteneur est supprimé tous ses fichiers le sont aussi. Cela peut donc engendrer de réels problèmes pour vos applications conteneurisées. Il a donc fallu vite trouver une solution pour sauvegarder ces données, d'où la création des volumes, qui restent la solution privilégiée quant à la persistance des données générées par les conteneurs.
C'est le même principe dans Kubernetes, lorsqu'un conteneur tombe en panne, kubelet le redémarre, et ses fichiers sont perdus. Dès lors lorsque vous exécutez un ensemble de conteneurs au sein d'un Pod, il est souvent nécessaire de **sauvegarder ou de partager des fichiers** entre ces conteneurs, et là encore les volumes résolvent ces deux problèmes. Nous allons donc dans ce chapitre étudier et comprendre le fonctionnement et la gestion des volumes dans Kubernetes.

Voici une liste - non exhaustive - des principaux types de stockage sous K3S :

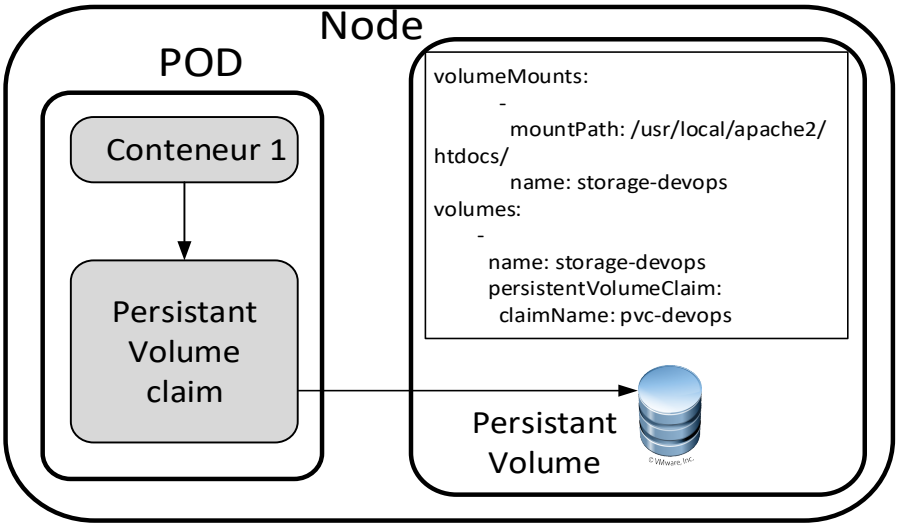
hostPath : Ce type de volume monte un fichier ou un répertoire du système de fichiers du noeud hôte dans votre pod.

emptyDir : Il s'agit d'un type de volume créé lors de la première affectation d'un pod à un noeud. Il reste actif tant que le pod s'exécute sur ce noeud. Le volume est initialement vide et les conteneurs du pod peuvent lire et écrire les fichiers dans le volume emptyDir. Une fois le pod retiré du noeud, les données du répertoire vide sont effacées.

secret : Un volume secret est utilisé pour transmettre des informations sensibles, telles que des mots de passe, à des pods.

persistentVolume : Un volume **persistentVolumeClaim** est utilisé pour monter un PersistentVolume dans un pod. Les volumes persistants sont un moyen pour les utilisateurs de «revendiquer» un stockage durable (tel qu'un disque persistant GCE NFS ou un volume iSCSI) sans connaître les détails de l'environnement cloud .

Dans cet exemple nous utiliserons le type de volume **hostPath** afin de vous démontrer la **simplicité de création d'un volume Kubernetes** mais également les inconvénients de ce type de volume.
Ce type de volume monte un fichier ou un répertoire du système de fichiers du noeud abritant votre pod



Rajoutons dans notre template du pod le volume non persistant suivant :

```
volumes:
- name: html
  hostPath:
    path: /data
    type: DirectoryOrCreate
```

Volume : définit la config du volume que nous allons réclamer, dans notre cas il sera de type **hostPath** et le volume sera monté dans le dossier **/data** du noeud hébergeant les Pods.

type: DirectoryOrCreate

Si rien n'existe au chemin fourni, un dossier vide y sera créé au besoin avec les permissions définies à 0755, avec le même groupe et la même possession que Kubelet.

Quand vous travaillez dans un grand environnement avec beaucoup d'utilisateurs qui déploient beaucoup de pods, les utilisateurs devront configurer à chaque fois le volume de stockage pour chaque pod (type de stockage hostpath non permanent). De ce fait chaque fois que des changements doivent être faits, l'utilisateur doit alors les appliquer pour tous ses pods.

Ici la solution serait de gérer le stockage de manière plus centralisée, pour ceci Kubernetes nous propose l'objet **PersistentVolume** (PV).

Un PersistentVolume est un élément de stockage dans le cluster qui a été provisionné par un administrateur, leur cycle de vie est indépendant de tout pod. Ensuite, nous avons aussi à notre disposition le **PersistentVolumeClaim** (PVC) qui permet aux utilisateurs de revendiquer l'utilisation des ressources d'un des PersistentVolume disponible dans le cluster k8s, dans le but de consommer une partie de ses ressources.

5.4.3) Les volumes persistants dynamiques

Volumes persistants dynamiques

Créez comme suit :docker-registry-pvc.yml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: docker-registry-pvc
 namespace: docker-registry
spec:
 accessModes:
 - ReadWriteOnce
 resources:
  requests:
   storage: 1Gi
```

Cela crée une revendication de volume persistante pour 1Gi d'espace. Comme il n'est pas associé à un PersistentVolume, il utilise le provisionnement dynamique.

Nous utilisons ce volume persistants dynamiques dans notre déploiement

```
volumes:
    - name: docker-registry-vol
      persistentVolumeClaim:
        claimName: docker-registry-pvc
```

```
kubectl --namespace docker-registry get pvc

NAME              STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
docker-registry-pvc   Bound    pvc-46cdbfbe-7db7-4079-a6ce-625740f658d5   1Gi        RWO            local-path     64m

$ kubectl --namespace docker-registry get pv

NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM                          STORAGECLASS
REASON   AGE
pvc-46cdbfbe-7db7-4079-a6ce-625740f658d5   1Gi        RWO            Delete           Bound    docker-registry/docker-registry-pvc
local-path           62m
```

Nous observons le PV créé dynamiquement pvc-46cdbfbe-7db7-4079-a6ce-625740f658d5

Où sont mes fichiers ?

root@k8s-server [~]# kubectl get pv

```
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM                          STORAGECLASS   REASON
AGE
docker-repo-pv                             3Gi        RWO            Retain           Available                                                   8d
pvc-7e3ae46b-fc8f-4312-b0e2-5ab01ac4279f   3Gi        RWO            Delete           Bound       default/docker-repo-pvc         local-path
8d
```

root@k8s-server [~]# kubectl describe pv pvc-7e3ae46b-fc8f-4312-b0e2-5ab01ac4279f

```
Name:            pvc-7e3ae46b-fc8f-4312-b0e2-5ab01ac4279f
Labels:          <none>
Annotations:     pv.kubernetes.io/provisioned-by: rancher.io/local-path
Finalizers:      [kubernetes.io/pv-protection]
StorageClass:    local-path
Status:          Bound
Claim:           default/docker-repo-pvc
```

```
Reclaim Policy:  Delete
Access Modes:    RWO
VolumeMode:      Filesystem
Capacity:        3Gi
Node Affinity:
  Required Terms:
    Term 0:      kubernetes.io/hostname in [k8s-node1]
Message:
Source:
    Type:        HostPath (bare host directory volume)
    Path:        /var/lib/rancher/k3s/storage/pvc-7e3ae46b-fc8f-4312-b0e2-5ab01ac4279f_default_docker-repo-pvc
    HostPathType:  DirectoryOrCreate
Events:          <none>
```

Les fichiers se trouvent dans :

Path: /var/lib/rancher/k3s/storage/pvc-7e3ae46b-fc8f-4312-b0e2-5ab01ac4279f_default_docker-repo-pvc

Comme nous n'avons spécifié aucune classe de stockage celle-ci est pris par défaut sur K3s :

StorageClass: local-path

le stockage de chemin local a une affinité de nœud

kubectl get pods -o wide

```
NAME                READY  STATUS    RESTARTS      AGE  IP          NODE        NOMINATED NODE  READINESS GATES
docker-registry-pod  1/1    Running  3 (42m ago)  8d   10.42.1.50  k8s-node1   <none>          <none>
```

Le noeud pour le Pod associé à ce volume persistant pvc est k8s-node1 Vérifions l'espace de stockage

```
root@k8s-node1 [ /var/lib/rancher/k3s/storage ]# ls
pvc-7e3ae46b-fc8f-4312-b0e2-5ab01ac4279f_default_docker-repo-pvc
root@k8s-node1 [ /var/lib/rancher/k3s/storage ]#
```

Parce qu'il a une affinité de nœud, si ce nœud est mort k8s-node1 , nous avons des problèmes

5.4.4) volumes persistants statiques

Nous avons utilisé le provisionnement dynamique (par défaut), ce qui signifie que nous n'avons pas autant de contrôle sur *l'emplacement du* stockage.

Si vous utilisez le fichier suivant le provisionnement dynamique est aussi mise en place ce que nous ne recherchons pas

```
apiVersion: v1
kind: PersistentVolume
metadata:
 name: docker-repo-pv
spec:
 capacity:
  storage: 1Gi
 accessModes:
 - ReadWriteOnce
 hostPath:
  path: /tmp/repository
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: docker-repo
spec:
 accessModes:
 - ReadWriteOnce
 resources:
  requests:
   storage: 1Gi
```

```
kubectl get pvc

NAME             STATUS   VOLUME                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
docker-repo-pvc     Bound   pvc-938faff3-c285-4105-83bc-221bb93a6603   1Gi      RWO         local-path    8m32s

$ kubectl get pv

NAME                        CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS     CLAIM                     STORAGECLASS
REASON   AGE
docker-repo-pv                 1Gi      RWO         Retain         Available                                    18m
pvc-938faff3-c285-4105-83bc-221bb93a6603   1Gi      RWO         Delete         Bound      default/docker-repo-pvc
local-path          3m42s
```

Vous pouvez voir que nous avons un PVC répertorié, mais deux PV. L'un des PV docker-repo-pv est marqué Disponible », ce qui indique qu'il est inutilisé.

Pour qu'il soit utilisé, il doit y avoir une sorte d'association entre la revendication et le volume.

Dans un cluster de production, en tant qu'administrateur vous n'utiliseriez pas le type **hostPath** . Au lieu de cela, mettez en place un service de partage réseau tel qu'un partage NFS, Google Compute Engine Persistent Disk, Amazon Elastic Block Store, etc ...

Créons notre volume persistant qui prend en charge le hostPath

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

Le fichier de configuration spécifie que le volume se trouve sur le nœud du cluster. La configuration spécifie également une taille de 10 gigaoctets et un mode d'accès de ReadWriteOnce, ce qui signifie que le volume peut être monté en lecture-écriture par un seul nœud. Il définit le nom StorageClass manual pour persistentVolume, qui sera utilisé pour lier les demandes PersistentVolumeClaim à ce PersistentVolume.

L'étape suivante consiste à créer un PersistentVolumeClaim. Les pods utilisent PersistentVolumeClaims pour demander un stockage physique.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: task-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
kubectl get pv task-pv-volume
```

Maintenant, la sortie affiche un de .STATUS Bound

```
NAME             CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS    CLAIM
STORAGECLASS   REASON    AGE
task-pv-volume   10Gi       RWO           Retain          Bound     default/task-pv-claim
manual                    2m
```

Regardez le PersistentVolumeClaim :

```
kubectl get pvc task-pv-claim
```

La sortie indique que PersistentVolumeClaim est lié à votre PersistentVolume, .task-pv-volume

```
NAME            STATUS    VOLUME           CAPACITY   ACCESSMODES   STORAGECLASS   AGE
task-pv-claim   Bound     task-pv-volume   10Gi       RWO           manual         30s
```

Lorsque vous allez faire une copie d'un fichier vers le point de montage du volume du pod, les fichier seront stockées dans le répertoire "/mnt/data" du node ou se situe ce pod

Une autre façon de le faire serait d'utiliser le provisionnement statique, où nous créons explicitement un PersistentVolume, puis le spécifions dans notre PersistentVolumeClaim.

secret est un type de volume utilisé pour **transmettre des informations sensibles**, telles que des mots de passe. Il permet de mieux contrôler la manière dont l'information est utilisée et réduit le risque d'exposition accidentelle.

```
echo -n '1f2d1e2e67df' > ./password.txt
```

Ensuite, nous **construisons un objet k8s de type secret** en prenant en compte notre fichier :

```
kubectl create secret generic secret-password --from-file=./password.txt
```

Par la suite, vérifions la **liste des secrets disponibles dans le cluster** :

```
kubectl get secrets
```

Les secrets peuvent être **décodés** via la commande kubectl get secret . Par exemple, pour récupérer le secret créé dans la section précédente, nous utiliserons la commande suivante :

```
kubectl get secret secret-password -o yaml
```

Utilisons la valeur fournit dans le champ **data** afin de découvrir notre mot de passe

```
echo 'MWYyZDFlMmU2N2Rm' | base64 --decode
```

lors du déploiement

```
volumes:
- name: secret-password
  secret:
    secretName: secret-password
```

Longhorn

5.5) L'objet pod

Un Pod est tout simplement un groupe d'un ou plusieurs conteneurs, avec un stockage et un réseau partagé. Par conséquence, ces conteneurs communiquent plus efficacement entre eux et assurent une localisation de la donnée. Nous utiliserons un template au format YAML afin de créer la configuration de notre pod,

```
apiVersion: v1
kind: Pod       # creation d'un pod
metadata:
    labels:
      type: web # clef type Valeur web
    name: nginx # nom du pod
spec:
  containers:
  - image: nginx #image nginx
    name: nginx #
```

```
kubectl create -f nginx-pod.yaml
```

Chaque pod possède un champ nommé **nodeName** qui prend comme valeur le nom du noeud qui accueillera le pod. Si ce champ est défini dans le spec du pod, il est alors prioritaire sur les différentes méthodes de sélection de noeuds que nous verrons plus tard.

```
apiVersion: v1
kind: Pod       # creation d'un pod
metadata:
    labels:
      type: web # clef type Valeur web
    name: nginx # nom du pod
spec:
  containers:
  - image: nginx #image nginx
    name: nginx #
    nodeName: worker-1
```

Le **nodeSelector** est une des formes de contrainte de sélection de nœud disponible dans Kubernetes. C'est un champ qu'on rajoute dans le spec des pods, et où on spécifie une paire de clé-valeur correspondant au label du noeud sur lequel on souhaite recevoir notre pod.
Créons un label

```
kubectl label node <NODE_NAME> <key>=<value>
kubectl get nodes <NODE_NAME> --show-labels
```

```
spec:
  containers:
  - name: nginx
  image: nginx
  nodeSelector:
    ntype: html
```

Vous pouvez **supprimer le label de votre noeud** en rajoutant le signe **-** (moins) à la fin de la commande kubectl label sans spécifier la valeur de la clé. Par exemple si je souhaite supprimer le label crée récédemment, je vais utiliser la commande suivante :

```
kubectl label node worker-1 ntype-
```

5.6) L'objet déploiement

Un Déploiement est un des objets permettant de lancer des Pods et à les déployer dans un système distribué complet, fiable et évolutif. Dans les bonnes pratiques de Kubernetes il est encouragé d'utiliser des Déploiements pour les applications stateless.
Lors de la création d'un Déploiement le controller associé va créer un ReplicaSet à partir de votre configuration. Le controller associé au ReplicaSet va créer une série de Pod à partir de la configuration du Replicaset.
Les avantages d'utiliser un Déploiement au lieu de créer directement un ReplicaSet sont : Historisation de l'objet : chaque changement dans l'objet (via un "apply" ou un "edit") va créer une sauvegarde de la version précédente.
Gestion du rollout et du rollback : en lien avec le point précédent, vous pourrez revenir en arrière sur une configuration.

Sans un template YAML

Vous pouvez **créer un Deployment sans utiliser de template au format YAML**, avec la commande kubectl run . Dans cet exemple, nous allons créer trois répliques de Pods utilisant l'image **alpine** , possédant le label **app: alpine** et exécutant la commande sleep 1d afin que les Pods soient actifs pendant 1 jour :

kubectl run alpine --image=alpine -r=3 -l="app=alpine" -- sleep 1d

Le Déploiement est un objet stable depuis la version 1.9 de Kubernetes, disponible sur 'l'api "apps/v1".

```
apiVersion: apps/v1
kind: Deployment # le type d'objet k8s que vous comptez créer.
metadata: # ici les metadata lié au Deployment
#donnée de type cléf valeur permettant d'identifer de manière unique l'objet
  # Nom de l'objet, il doit être unique dans le namespace
  name: deployment-example
  namespace: default
spec: # Ici les spécifications du Deployment
  # Lance 3 Pods
  replicas: 3
 selector:
    matchLabels:
      app: nginx
  template:
    metadata: # ici les metadata lié aux Pods (ne pas confondre avec les précédents)
      labels:
        app: nginx
    spec: # Ici les spécifications des Pods
      containers: # un Pod peut contenir plusieurs conteneurs
      - name: web
        image: nginx:1.10
```

5.7) L'objet sonde ("probes")

Il existe deux types de sondes dans un Pod :

- readinessProbe : savoir quand votre application a fini de démarrer
- livenessProbe : savoir si votre application est fonctionnelle

Ces deux sondes sont essentielles. La première permet de ne pas diriger le trafic réseau vers un Pod encore en cours de démarrage, ce qui vous évitera des erreurs 500 lors de la scalabilité. La deuxième sonde permet à Kubernetes de redémarrer votre Pod en cas d'erreur applicative. En effet, votre application peut être en erreur sans que le conteneur soit arrêté.

Il y a plusieurs moyens de configurer vos sondes, via des appels HTTP ou TCP ou même en exécutant une commande régulièrement dans votre conteneur.

```
livenessProbe:
  initialDelaySeconds: 150
  tcpSocket:
    port: 5901
  timeoutSeconds: 10
name: jdk-mvn-py3-vnc
readinessProbe:
  httpGet:
    path: /
    port: 6901
    scheme: HTTP
  initialDelaySeconds: 150
  timeoutSeconds: 10
```

5.8) L'objet service

Un service peut être défini comme une abstraction par-dessus les Pods, qui attribue aux pods leurs propres adresses IP et un nom DNS unique, et peut aussi équilibrer la charge entre eux.
En effet, les pods ont une durée de vie très limité et sont à maintes reprises créés et détruits, ce qui par conséquent altère continuellement leurs adresses IP.
Interviennent alors les services qui vont quant à eux permettre aux clients d'échanger de manière plus fiable avec les conteneurs s'exécutant dans le Pod à l'aide d'une adresse IP virtuelle statique grâce au travail du composant **kube-proxy**
.
Il existe quatre types de services pour une utilisation particulière :

ClusterIP : C'est le type par défaut. Il expose le Service sur une adresse IP interne du cluster. De ce fait, le service n'est accessible que depuis l'intérieur du cluster.
NodePort : Il expose le service vers l'extérieur du cluster à l'aide du NAT (la plage de ports autorisés est entre 30000 et 32767).
LoadBalancer : Il utilise l'équilibreur de charge des fournisseurs de cloud. Ainsi, les services NodePort et ClusterIP sont créés automatiquement et sont acheminés par l'équilibreur de charge externe.
ExternalName : Ce service effectue une simple redirection CNAME (par exemple rediriger le trafic vers le nom de domaine "example.com").

Créons d'abord notre template sous le format YAML de manière à **construire notrepropre service**, avec le contenu suivant :

```
apiVersion: v1
kind: Service
metadata:
name: flask-service
spec:
type: ClusterIP #type de notre service.
app: flask # flask-service utilisé que pour les apps flask
ports:
- port: 5000 #port utilisé par notre service afin de communiquer en interne
  targetPort: 5000 #port qui est utilisé par le ou les Pods
  name: flask-cluster-ip


kubectl get svc
```

Imaginons, que nous avons besoin cette fois-ci d'accéder à notre Pod depuis l'extérieur de notre cluster Kubernetes. Nous exploiterons ainsi cette le type **NodePort** .
Sans la nécessité de détruire notre ancien service, nous allons modifier et appliquer les changements de notre ancien template YAML, en suivant les étapes suivantes :
Premièrement modifions notre ancien template YAML, comme suit :

```
apiVersion: v1
kind: Service
metadata:
name: flask-service
spec:
type: NodePort
selector:
app: flask
ports:
  - port: 5000
  targetPort: 5000
  nodePort: 30001#translation d'adresse vers le port 30001
  name: flask-np
```

la plage d'IP dans un service de type NodePort se situe entre 30000 et 32767. On ne peut donc pas dépasser cette limite dans le champ **nodePort** .

nous créerons un pod qui exécute un conteneur basé sur l'image alpine contenant une variable d'environnement nommé **DEMO_ENV** et sa valeur "Coucou". Voici donc à quoi va ressembler le fichier de configuration du pod :

```
apiVersion: v1
kind: Pod
metadata:
name: alpine
spec:
containers:
- name: alpine
image: alpine
env:
- name: DEMO_ENV
value: "Coucou"
command: ["/bin/sh", "-c"]
args:
- while true; do
sleep 1;
```

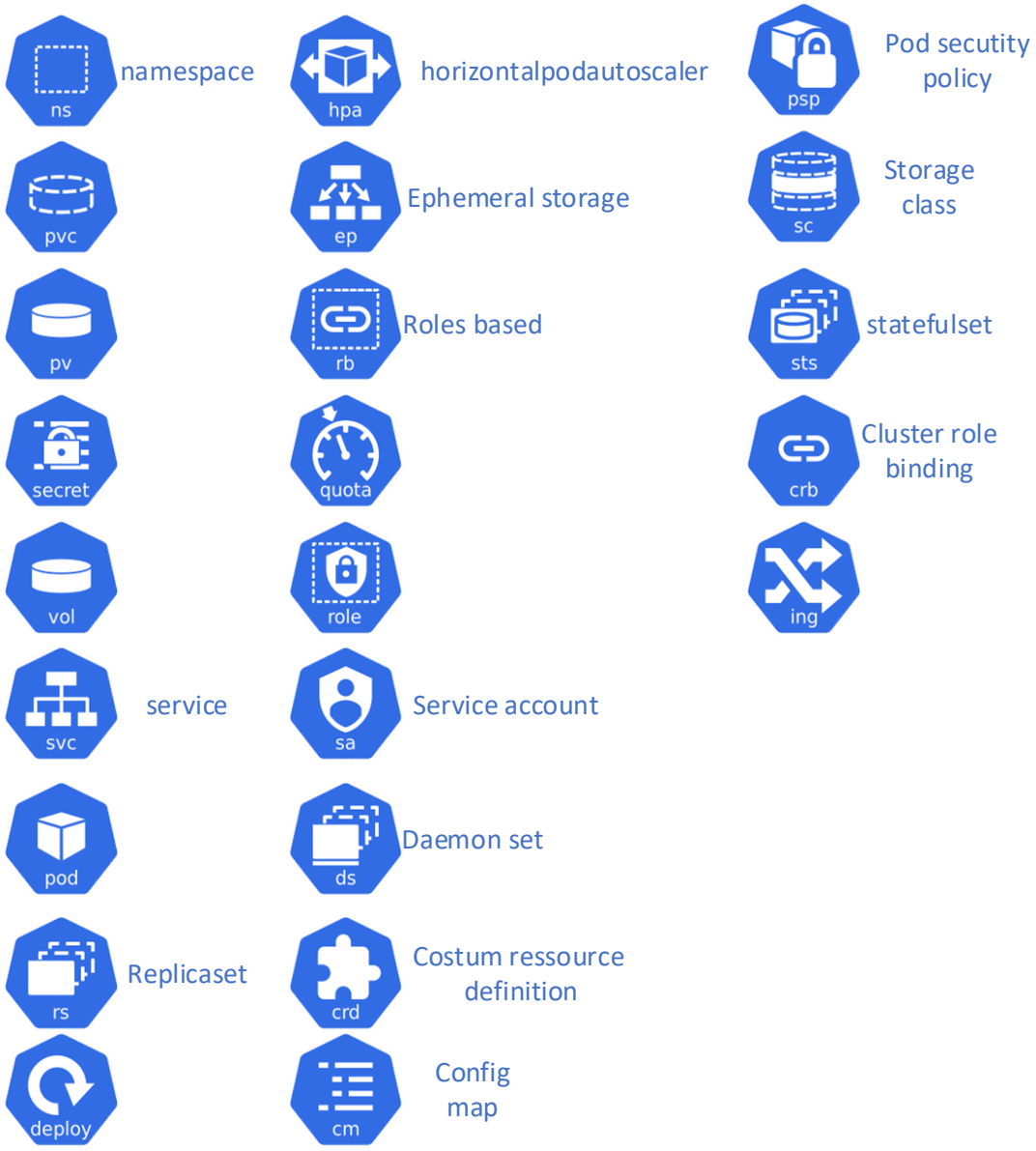
```
done
```

Créez ensuite le pod basé sur le fichier de configuration YAML:

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

Par la suite, nous allons exécuter la commande printenv afin de **répertorier les variables d'environnement disponible dans le conteneur de notre pod** :

```
kubectl exec -it alpine -- printenv | grep DEMO_ENV
```


ns
namespace
hpa
horizontalpodautoscaler
psp
Pod secutity policy
pvc
ep
Ephemeral storage
sc
Storage class
pv
rb
Roles based
sts
statefulset
secret
quota
crb
Cluster role binding
vol
role
ing
svc
service
sa
Service account
pod
ds
Daemon set
rs
Replicaset
crd
Costum ressource definition
deploy
cm
Config map