



# Bienvenue

## Les containers , la révolution ?

### **Le modèle containers**

Les Namespaces et les Control groups

### **Docker.io**

Docker

Historique

Pourquoi Docker

Mise en oeuvre

Exploitation des containers

Orchestrations avec Docker Compose

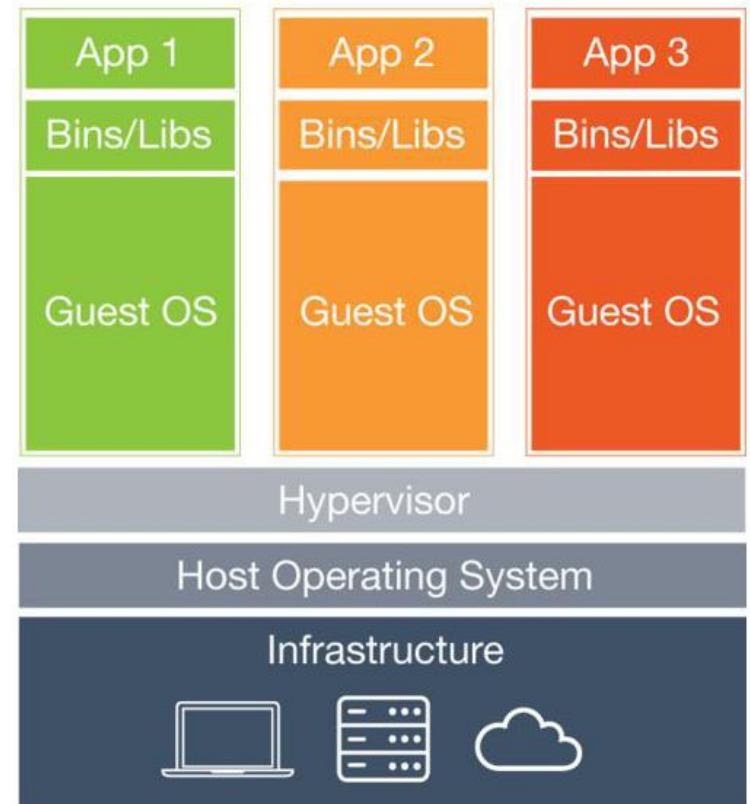
Cluster avec Docker Swarm

### **Autres solutions**

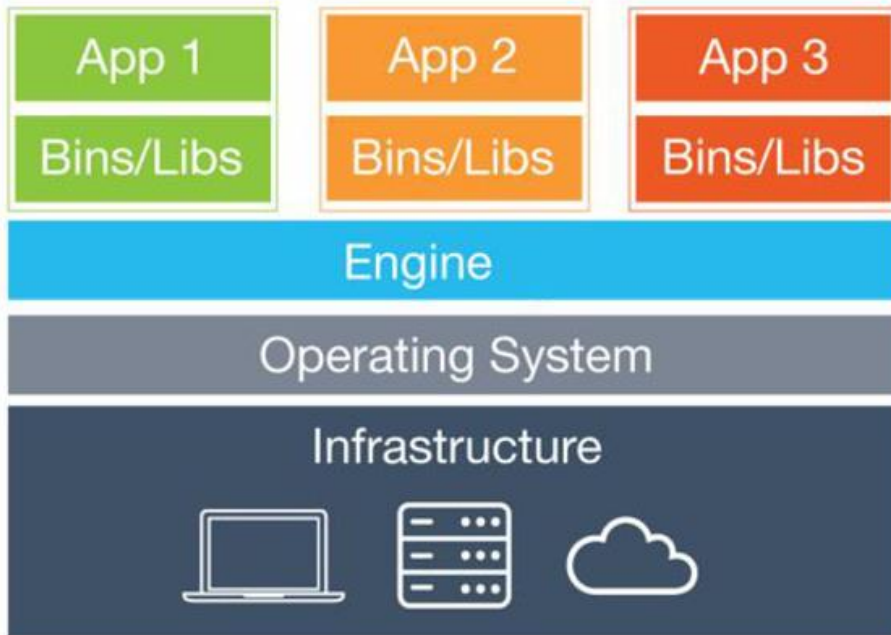
# Le modèle containers

Une machine virtuelle c'est :

- Une représentation logicielle d'une machine physique
- Une émulation du CPU, mémoire, nic, contrôleur de disque, disque, etc..
- Un OS installé dans la VM – Guest OS ou OS invité
- Une forte empreinte sur l'hyperviseur



# Le modèle containers



Un container, c'est :

- Pas d'émulation de matériel
- Pas de noyau
- Un accès direct au matériel
- Très léger, déploiement rapide, gain de performances

# Les Namespaces et les Control groups

**LXC et Docker** sont des “outils” qui permettent d’isoler très simplement plusieurs petits systèmes d’environnement ou applications sur une machine physique, appelée hôte

Ces environnements exécutent des applications (web, BDD, DHCP, ..) qui n’interfèrent ni avec le système installé sur la machine, ni entre eux.

Ils sont donc **ISOLÉS**

Les containers utilisent des fonctions du noyau Linux afin d’isoler ces environnements

**Les Namespaces et les Control groups**

# Les Namespaces et les Control groups

Les namespaces sont une fonctionnalité du noyau Linux :

## **Process Namespace**

Isole les processus, le conteneur dispose de sa liste de processus

## **Network Namespace**

Isole les interfaces réseaux, le conteneur dispose de ses propres interfaces

## **Mount Namespace**

Isole les systèmes de fichiers

## **UTS Namespace**

Permet au conteneur de disposer de ses noms d'hôtes et de domaines

## **IPC Namespace**

Ses propres processus Inter Communications

## **User Namespace**

Permet aux conteneurs de disposer de ses utilisateurs, l'utilisateur root dans l'espace de nom possède l'id et le gid 0

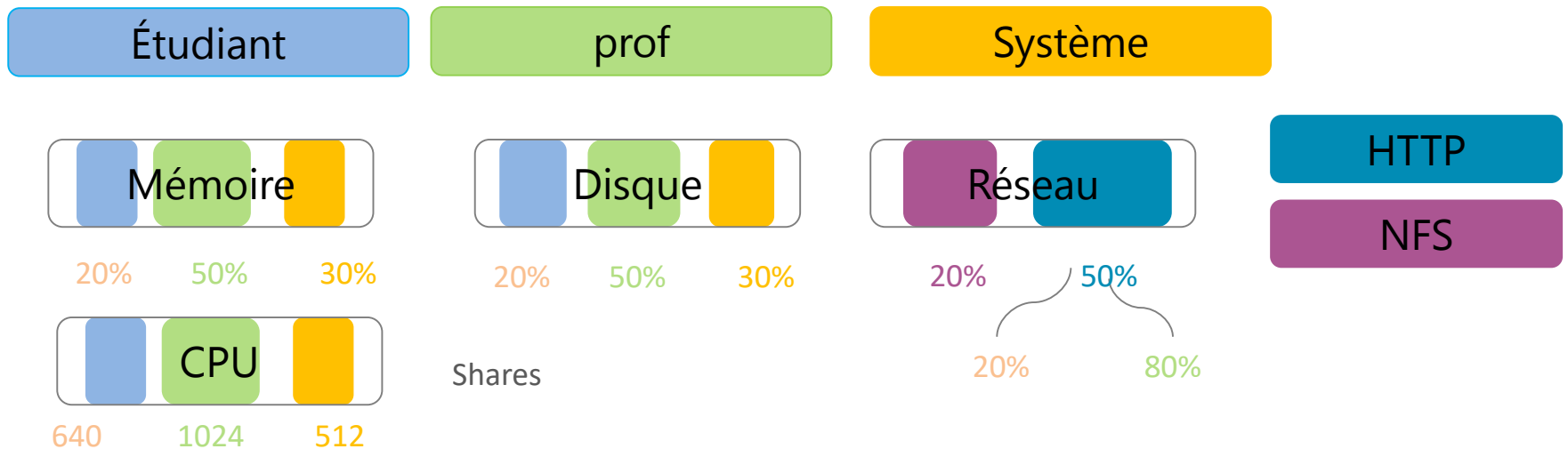
# Les Namespaces et les Control groups

Les **cgroups** ou **Control Groups**- Groupes de contrôle est une fonctionnalité du noyau Linux qui a pour but de :

Contrôler les ressources

Allocation, l'interdiction, priorisation, figer

Surveiller et mesurer les quantités de ressources consommées



# Les Namespaces et les Control groups

Les **cgroups** sont organisés en sous-systèmes ou modules

Un sous-système est un contrôleur de ressources :

**Blkio** : Surveille et contrôle l'accès des tâches aux entrées/sorties sur des périphériques block

**Cpu** : Planifie l'accès de la CPU

**Cpuacct** : Rapports sur les CPU utilisées, **cpu.shares** : Part relative du temps CPU disponible pour les tâches

**Cpuset** : Assigne des CPU à des tâches

**Devices** : Autorise ou refuse l'accès aux périphériques

**Freezer** : Suspend ou réactive les tâches

**memory** : Utilisation mémoire



# Docker.io

Docker

Historique

Pourquoi Docker

Mise en oeuvre

Exploitation des containers

Orchestrations avec Kubernetes

# Docker.io



Docker Hub



Kitematic



Docker Compose



Docker Registry



Docker Engine



Docker Machine



Docker Swarm



Docker Datacenter



Docker Cloud

# Build, Ship, Run

**Docker** permet construire des applications conteneurisées avec ses dépendances,

de déployer rapidement,

d'exécuter avec fiabilité quel que soit l'environnement Linux



**Build**



**Ship**



**Run**



# Historique

Docker a été développé par Solomon Hykes pour un projet interne de dotCloud, une société proposant une plate-forme en tant que service, avec les contributions d'Andrea Luzzardi et Francois-Xavier Bourlet, également employés de dotCloud.

Docker est une évolution basée sur les technologies propriétaires de dotCloud, elles-mêmes construites sur des projets open source.

# Historique

Docker a été distribué en tant que projet open source à partir de mars 2013<sup>3</sup>.

Au 18 novembre 2013, le projet a été mis en favoris plus de 7 300 fois sur [GitHub](#) (14<sup>e</sup> projet le plus populaire), avec plus de 900 forks et 200 contributeurs<sup>6</sup>.

# Historique

En octobre 2015, le projet a été mis en favoris plus de 25 000 fois sur [GitHub](#), avec plus de 6 500 forks et 1 100 contributeurs

En septembre 2016, le projet a été mis en favoris plus de 34 000 fois sur [GitHub](#), avec plus de 10 000 forks et 1 400 contributeurs<sup>6</sup>.

# Pourquoi Docker

OpenSource

beaucoup plus léger qu'une machine virtuelle

beaucoup plus rapide qu'une machine virtuelle, bootable en quelques secondes

Performance native

# Pourquoi Docker

OpenSource

portables de cloud en cloud

Portable d'une infrastructure a une autre

Architecture micro-services



# Pourquoi Docker

Des outils pour construire, créer et déployer facilement

Docker Engine,

Docker registry, Docker Hub

Des outils pour Orchestrer, clusteriser, ....

Docker cloud,

Docker Machine,

Docker compose,

Docker swarm

# Pourquoi Docker

Des outils pour déployer facilement

- Docker Engine,

- Docker compose

- Docker registry, Docker Hub

Des outils pour clusteriser un DataCenter

- Docker cloud,

- Docker Machine,

- Docker swarm

- Docker DataCenter

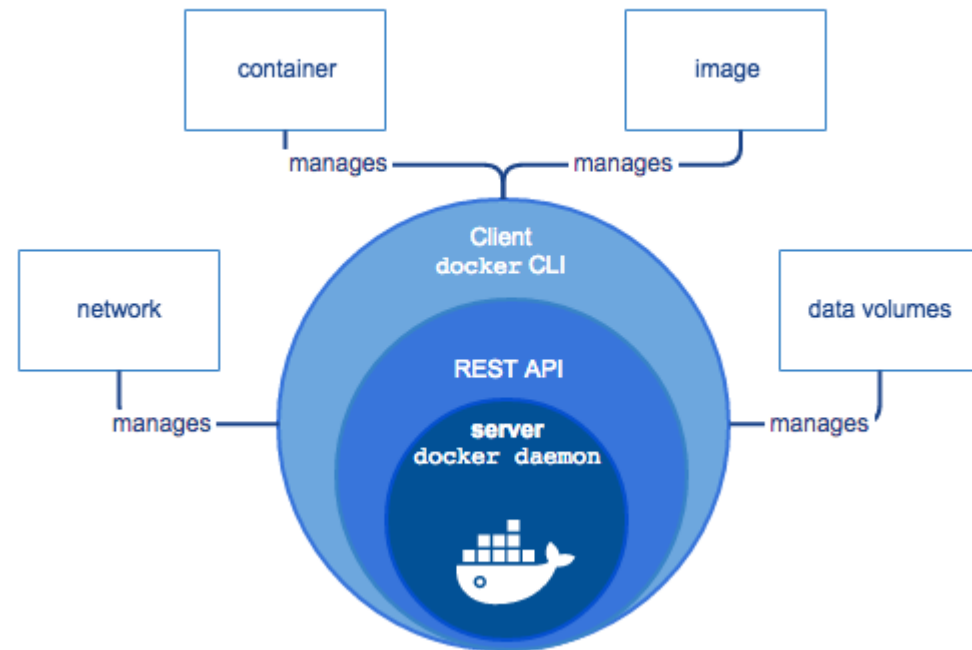
# Architecture

La plateforme Docker est composée de **trois éléments** :

**Le démon Docker** qui s'exécute en arrière-plan et qui s'occupe de gérer vos conteneurs. Le démon crée et gère les objets Docker. Les objets Docker comprennent *des images, des conteneurs, des réseaux, des volumes de données...*

**Le client Docker** permet d'interagir avec le démon par l'intermédiaire d'un outil en ligne de commande

**Une API REST** qui permet d'interagir à distance avec le démon



# Architecture

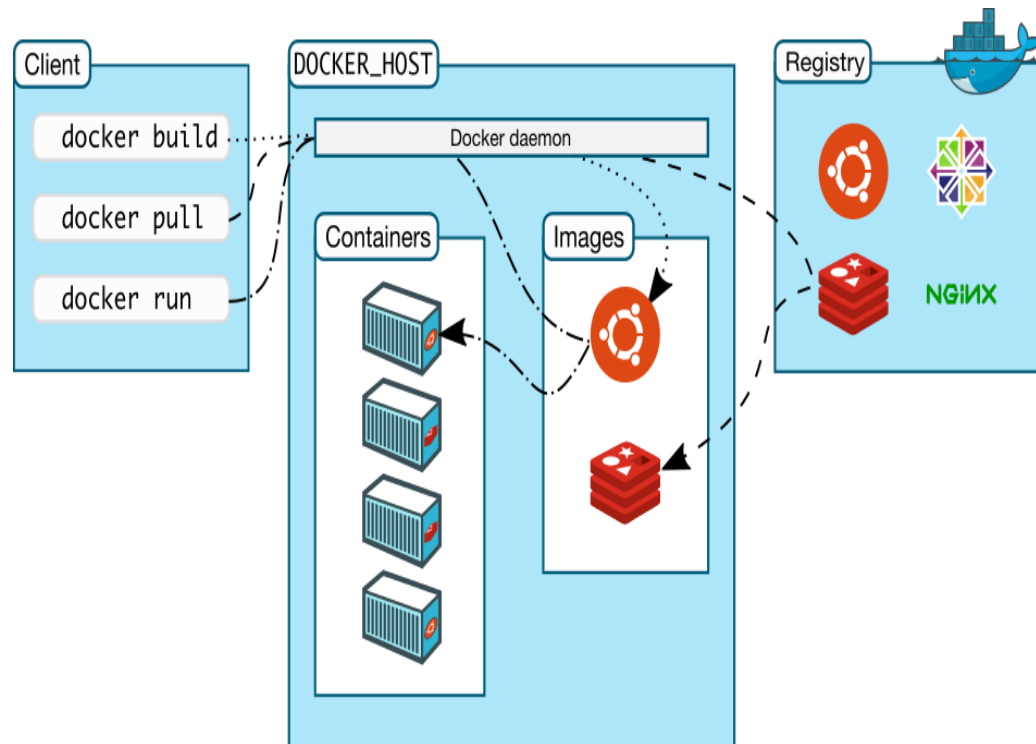
La plateforme Docker est composée de **trois éléments** :

**Le client Docker** permet d'interagir avec le démon par l'intermédiaire d'un outil en ligne de commande



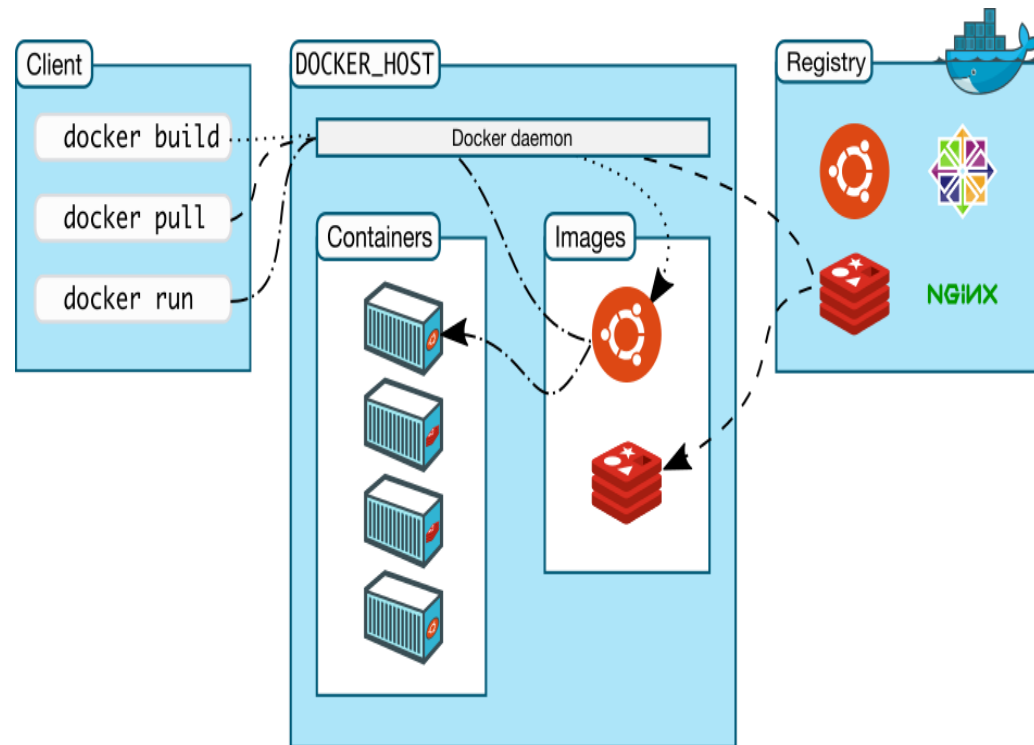
# Architecture

- Docker utilise une architecture **client-serveur**.
- Le client discute avec le démon Docker
- Le démon fait le gros de la construction, la distribution des conteneurs et gère les conteneurs en cours d'exécution.
- Le client Docker et le démon peuvent fonctionner sur le même système, ou connecter un client Docker à un démon Docker distant.
- Le client Docker et démon communiquent via les sockets ou via une API RESTfull.



# Architecture

- Pour bien comprendre Docker
- On trouve trois ressources:
  - **Docker Images**
  - **Docker Registry**
  - **Docker Container**



# Docker Registry

- Le registre Docker est une bibliothèque, un magasin d'images, un repository
- Le Docker Hub est la bibliothèque officielle Docker.
- Des images sont disponibles de toutes les distributions, d'énormément d'applications
- Ces images sont téléchargeables, mais également « chargeables » sur le Hub.
- On “Pull” et “push” des images
- Création d'un Registre privé

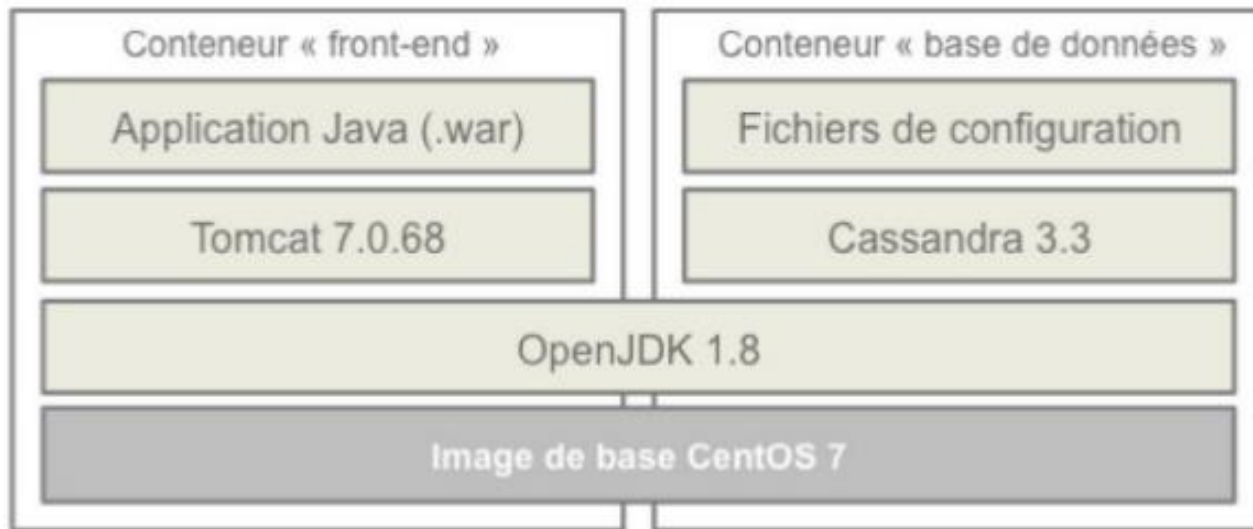
# Docker images

- Composants de base de Docker
- Des archives en lecture seule qui peuvent être échangées entre plusieurs hôtes
- Peut contenir un « système d'exploitation » Débian avec Apache ou un OS tout simplement.
- Sont utilisées pour créer des conteneurs Docker.
- Fonctionnent sous un modèle d'héritage en couches
- Peuvent avoir été construites à partir d'une autre image qui elle-même a pu être construite à partir d'une autre image.
- Docker fournit un moyen simple de construire de nouvelles images ou mettre à jour des images existantes



# Docker images

- Les images Docker sont des **modèles en lecture seule** à partir desquels les conteneurs sont créés.
- Chaque image est constituée d'une **série de couches**. Pour ce faire Docker utilise le système de fichiers **UnionFS** afin de combiner ces couches en une seule image.




# Docker Images

- Les images Docker sont construites à partir d'images de base
- Ces instructions sont stockées dans un fichier appelé **Dockerfile**.
- Un Dockerfile est un script texte qui contient des instructions et des commandes pour la construction à partir de l'image de base.
- Docker lit ce Dockerfile et construit l'image.
- On dispose donc d'une nouvelle image.

# Docker Registry


- Le registre Docker est une bibliothèque, un magasin d'images, un repository
- Le Docker Hub est la bibliothèque officielle Docker.
- Des images sont disponibles de toutes les distributions, d'énormément d'applications
- Ces images sont téléchargeables, mais également « chargeables » sur le Hub.
- On “Pull” et “push” des images
- Création d'un Registre privé

# Docker Registry

 Dashboard Explore Organizations

Q Search

Create

 pycloux

pycloux


Repositories Stars Contributed

Private Repositories: Using 0 of 1 [Get more](#)

## Repositories

Create Repository +

Type to filter repositories by name

	pycloux/test public	0 STARS	0 PULLS	> DETAILS
--	------------------------	------------	------------	--------------

### Docker Trusted Registry

Need an on-premise registry? [Get a 30-day free trial](#)

# Docker Hub

OFFICIAL REPOSITORY

centos ☆

Last pushed: 10 hours ago

Repo Info Tags

## Short Description

The official build of CentOS

## Docker Pull Command



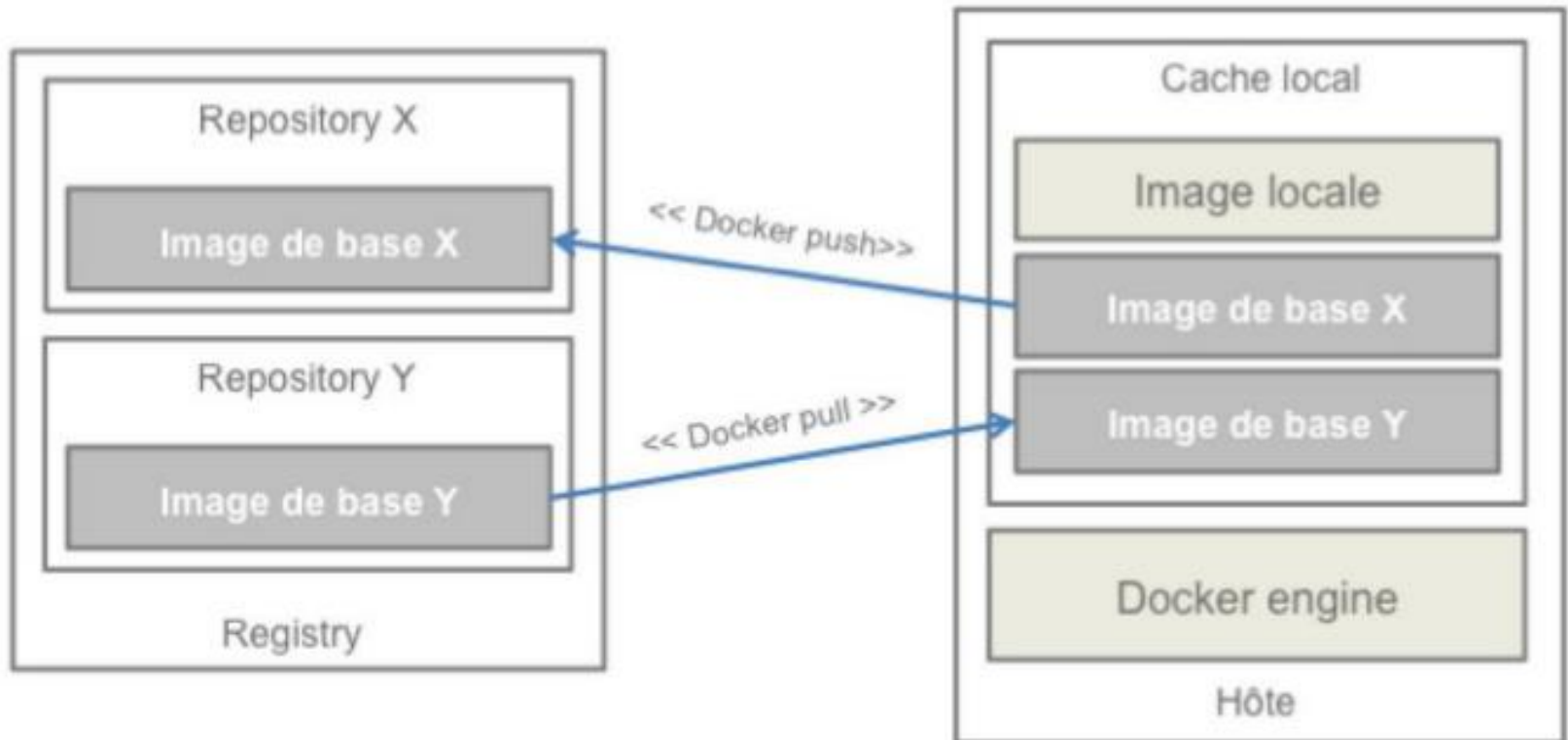
```
docker pull centos
```

## Full Description

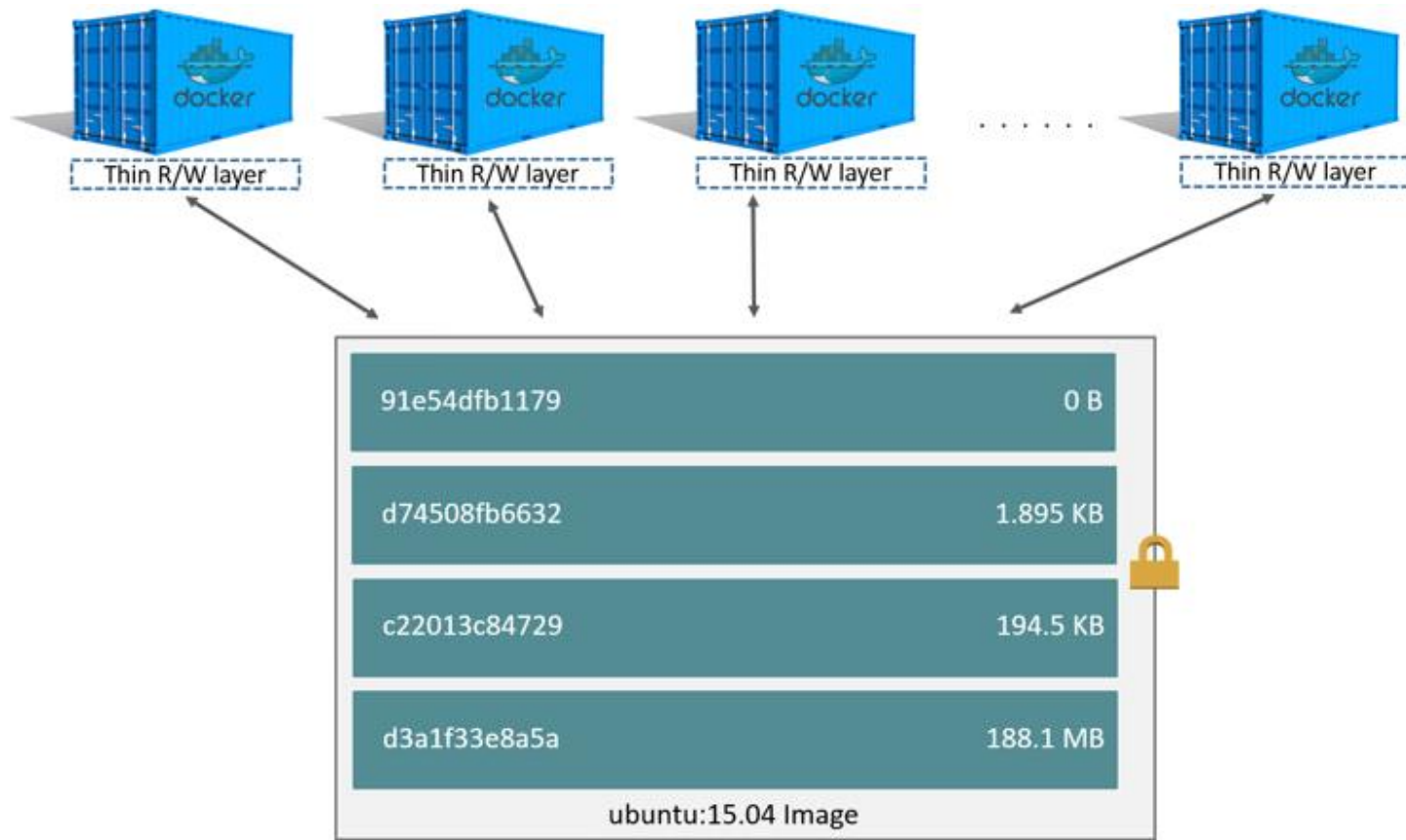
## Supported tags and respective Dockerfile links

- latest, centos7, 7 ([docker/Dockerfile](#))
- centos6, 6 ([docker/Dockerfile](#))
- centos5, 5 ([docker/Dockerfile](#))
- centos7.2.1511, 7.2.1511 ([docker/Dockerfile](#))

# Docker Hub

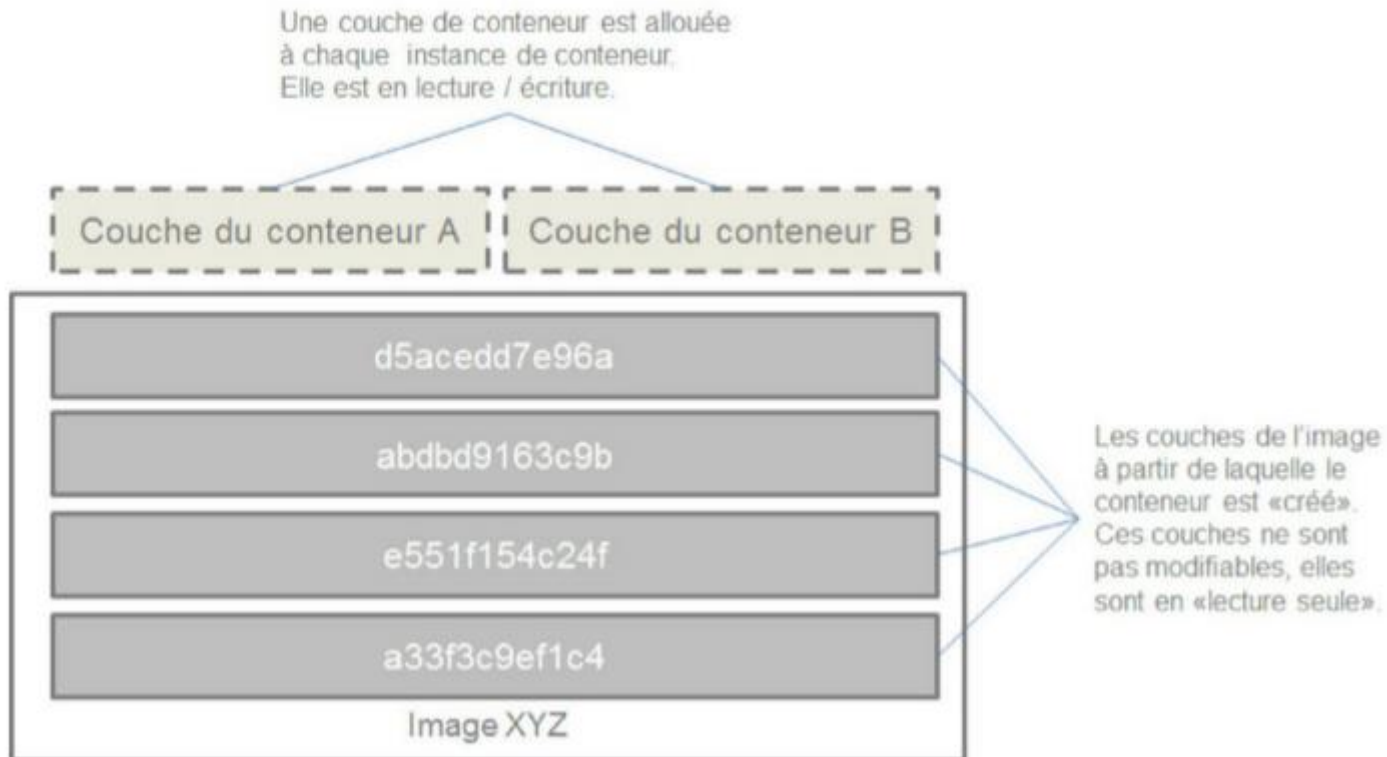


# Docker containers



# Docker containers

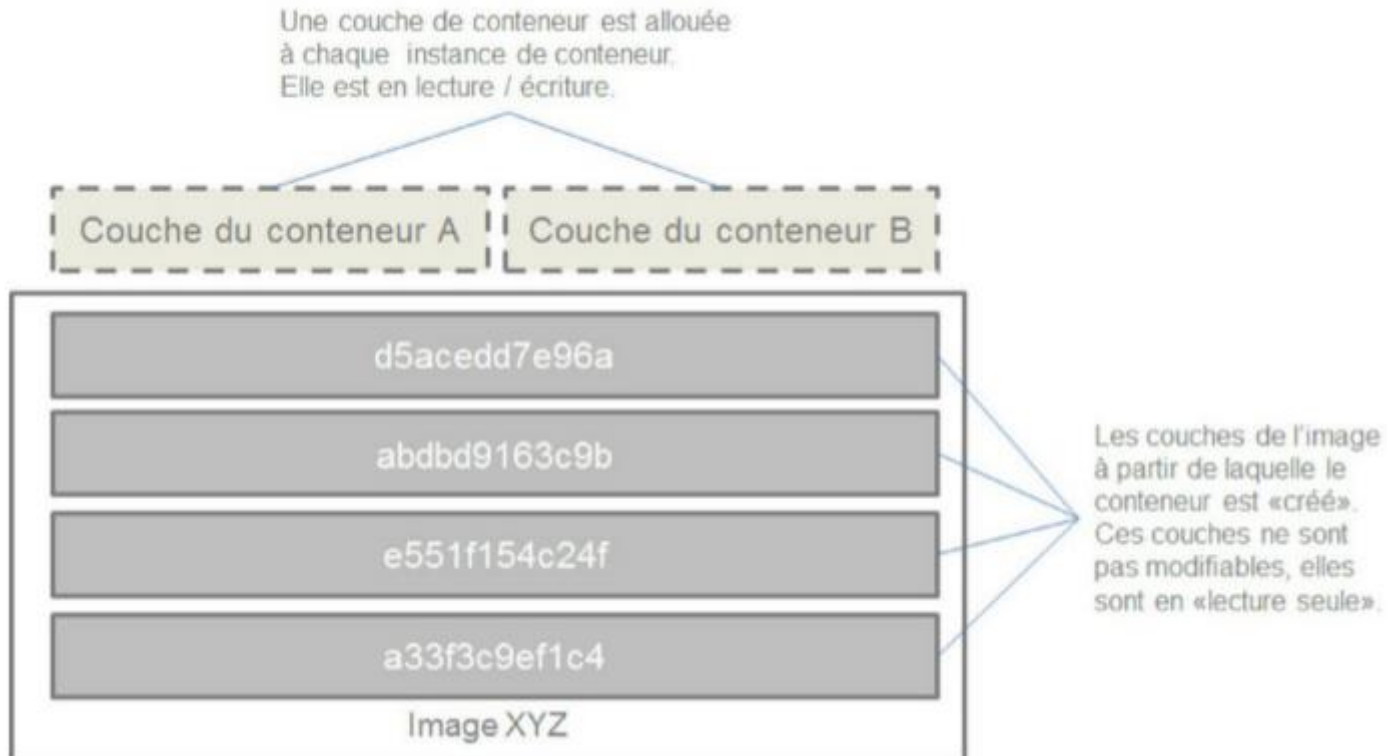
- Les Conteneurs Docker sont construits à partir d'images de base ou des images « personnalisées »





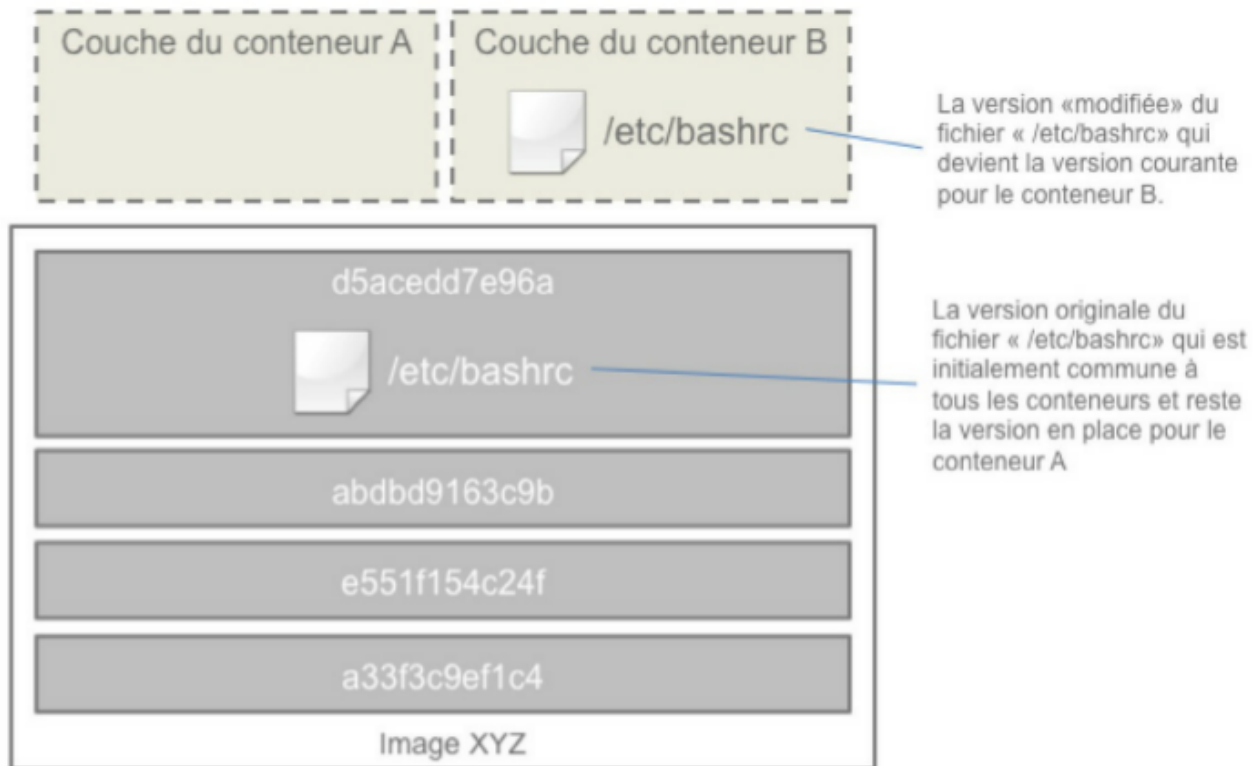
# Docker containers

- Le conteneurs est mise en exécution, exécute des processus, alloue de la mémoire et écrire des données. Ces données sont produites dans une couche au-dessus de toutes les autres.



# Docker containers

- Le Copy On Write, COW. Le fichier original est copié et la nouvelle version surcharge la version originale. Le conteneur ne contient que le différentiel par rapport à l'image



# Docker containers

- Une des raisons pour laquelle, Docker est si léger.
- Lorsque l'on modifie une image pour Docker, mettre à jour une application par exemple. Une nouvelle couche est construite. Pas de remplacement d'image ou reconstruction, une seule cette couche est ajoutée.

# Mise en œuvre de Docker Engine

- Docker fournit plusieurs versions du moteur.
- Pour Microsoft Windows, pour Mac OSX et bien sur pour Linux
- La version Open source devient le projet Moby
- Les versions Docker CE et Docker EE
- La version Docker Community Edition
- La version Enterprise Edition

# Mise en oeuvre

- Installation Docker Centos 7

```
[root@docker00 ~]# yum -y install docker
```

- Démarrage et activation du service

```
[root@docker00 ~]# systemctl start docker
```

```
[root@docker00 ~]# systemctl enable docker
```

# Exploitation des images

- Recherche d'images

```
[root@docker00 ~]# docker search centos
```

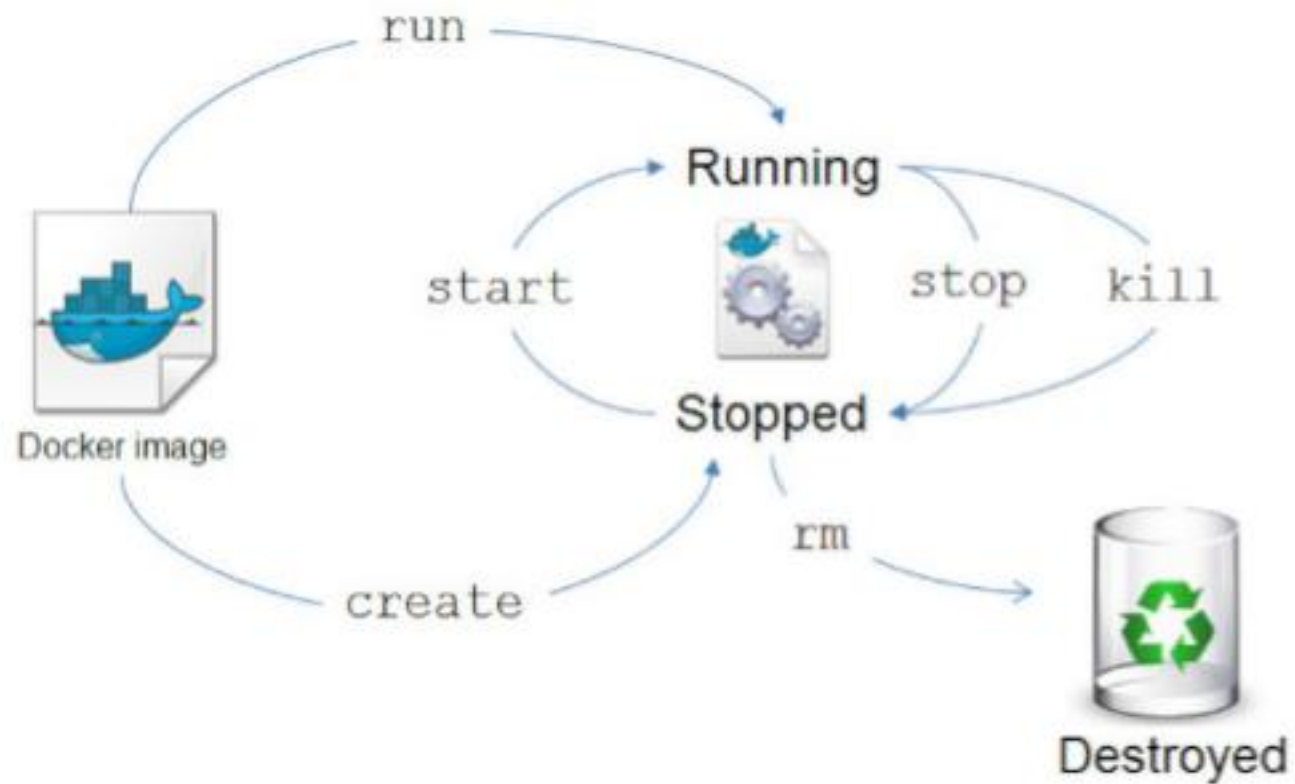
- Téléchargement d'une image centos

```
[root@docker00 ~]# docker pull centos
```

```
[root@docker00 ~]# docker image list
```

```
[root@docker00 ~]# docker rmi ....
```

# Exploitation des containers



# Exploitation des containers

- Création d'un container

```
[root@docker00 ~]# docker run centos /bin/echo "Welcome to the Docker World"
Welcome to the Docker World
```

- Création

```
[root@docker00 ~]# docker run -it centos /bin/bash
# Ctrl+p, Ctrl+q pour sortir de la console
```

```
[root@docker00 ~]# docker ps
```

6bfd366dba99	centos	"/bin/bash"	8 seconds ago	Up 6 seconds	elated_spence
--------------	--------	-------------	---------------	--------------	---------------



# Exploitation des containers

- Création d'un container avec installation d'Apache

```
[root@docker00 ~]# docker run centos /bin/bash -c "yum -y update; yum -y install httpd"
```

- Liste les containers

```
[root@docker00 ~]# docker ps
```

695d7674948f	centos	"/bin/bash -c 'yum -y'"	9 minutes ago	Exited
(0)	7 minutes ago	sick_mccarthy		

- Création d'une image Centos avec Apache

```
[root@docker00 ~]# docker commit 695d7674948f local/centos_httpd
```

# Exploitation des containers

- Liste les images

```
[root@docker00 ~]# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
image/centos_httpd	latest	9072383f22ba	About a minute ago	288.5 MB

- Création d'un containers avec la nouvelle image httpd

```
[root@docker00 docker run my_image/centos_httpd /usr/bin/which httpd  
/usr/sbin/httpd
```

# Exploitation des containers

## Les ports

- Création d'un conteneur avec redirection des ports

```
[root@docker00 ~]# docker run -it -p 8081:80 my_image/centos_httpd /bin/bash
[root@821bc61cb2e6 /]# /usr/sbin/httpd &
[root@821bc61cb2e6 /]# echo "httpd on Docker Container" >
/var/www/html/index.html
```

```
[root@821bc61cb2e6 /]# # exit with Ctrl+p, Ctrl+q
```

```
[root@docker00 ~]# docker ps
1c0ee4dbf062    image/centos_httpd  "/bin/bash"        6 minutes ago    Up 6 minutes
0.0.0.0:8081->80/tcp  modest_boh
```

# Exploitation des containers

## Les ports

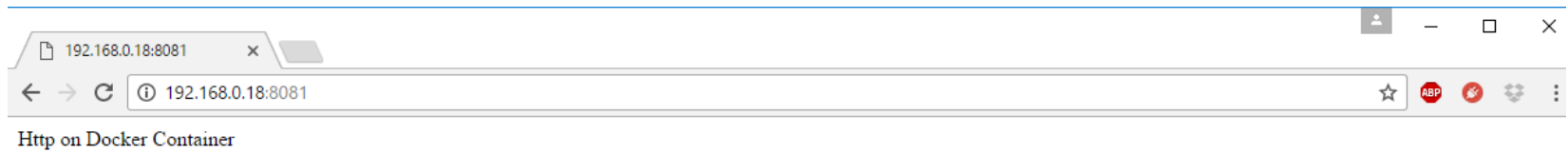
- Création d'un conteneur avec redirection des ports

```
[root@docker00 ~]# docker run -it -p 8081:80 my_image/centos_httpd /bin/bash
[root@821bc61cb2e6 /]# /usr/sbin/httpd &
[root@821bc61cb2e6 /]# echo "httpd on Docker Container" >
/var/www/html/index.html
```

```
[root@821bc61cb2e6 /]# # exit with Ctrl+p, Ctrl+q
```

```
[root@docker00 ~]# docker ps
1c0ee4dbf062    image/centos_httpd  "/bin/bash"        6 minutes ago    Up 6 minutes
0.0.0.0:8081->80/tcp  modest_boh
```

# Exploitation des containers



# Exploitation des containers

## Les volume de données

- Création d'un volume de données.

```
[root@docker00 ~]# docker run -it -p 8081:80 -v /html:  
/var/www/html/index.html my_image/centos_httpd /bin/bash  
[root@821bc61cb2e6 /]# /usr/sbin/httpd &  
[root@docker00 ~]# echo "httpd on Docker Container" >/html/index.html
```

```
[root@docker00 ~]# docker ps  
1c0ee4dbf062    image/centos_httpd  "/bin/bash"        6 minutes ago    Up 6 minutes  
0.0.0.0:8081->80/tcp  modest_boh
```

# Exploitation des containers

## Le Dockerfile

- Permet la construction d'images personnalisées
- Simple fichier texte
- Portable
- Syntaxe simple à comprendre “Clé/valeur”

# Exploitation des containers

**FROM** : définit l'image de base à utiliser pour construire notre image

**MAINTAINER** : la personne qui a créé ou maintient le Dockerfile.

**RUN** : exécute une commande sur l'image courante. A chaque RUN une nouvelle image (layer) est créée

**ADD** : permet d'ajouter de nouveaux fichiers au conteneur

**CMD** : commande à exécuter lors du lancement d'un conteneur

**ENTRYPOINT** : commande par défaut exécutée au démarrage du conteneur.

**VOLUME** : Mappe des filesystems locaux aux conteneurs



# Exploitation des containers

**EXPOSE** : définit les ports pour le conteneur

**USER** : L'utilisateur qui exécute les commandes RUN et  
**ENTRYPOINT**.

# Exploitation des containers

- Création d'un Dockerfile

```
[root@docker00 ~]# vi Dockerfile
```

```
FROM centos # l'image a utiliser  
MAINTAINER ludo <admin@ipsis.local>  
RUN yum -y install httpd  
RUN echo "Hello DockerFile" > /var/www/html/index.html  
EXPOSE 80  
CMD ["-D", "FOREGROUND"]  
ENTRYPOINT ["/usr/sbin/httpd"]
```

# Exploitation des containers

- On construit l'image a partir du Dockerfile

```
[root@docker00 ~]#docker build --tag=web_server .
```

```
Sending build context to Docker daemon 14.34 kB
```

```
Step 1 : FROM centos
```

```
---> 67591570dd29
```

```
Step 2 : MAINTAINER ludo <admin@ipsis.local>
```

```
---> Running in 43ab5e42a069
```

```
---> 83a178648b52
```

```
Removing intermediate container 43ab5e42a069
```

```
Step 3 : RUN yum -y install httpd
```

```
---> Running in 35a416805660
```

```
.....
```

# Exploitation des containers

- On visualise l'image

```
[root@docker00 ~]#docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
web_server	latest	3aad8e9dde83	16 minutes ago	287.6 MB

- On teste l'image

```
[root@docker00 ~]#docker run -d -p 80:80 web_server
```

# Exploitation des containers

- Création du container Registry

```
[root@docker00 ~]#docker run -d -p 5000:5000 --name registry registry:2
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
web_server	latest	3aad8e9dde83	16 minutes ago	287.6 MB

- Téléchargement d'une image

```
[root@docker00 ~]#docker pull ubuntu
```

- On taggue et push la nouvelle image

```
[root@docker00 ~]#docker tag ubuntu localhost:5000/ubuntu_local
```

```
[root@docker00 ~]#docker push localhost:5000/ubuntu_local
```

# Docker Compose



# Docker Compose

Composer est un outil pour définir et exécuter des applications Docker multi-conteneurs

Au travers d'un simple fichier

On configure des services pour nos applications.

Avec une seule commande

On créer et démarrer tous les services à partir de votre configuration

# Docker Compose

L'utilisation de Compose est un processus en trois étapes.

Définir l'environnement de votre application avec un Dockerfile afin qu'il puisse être reproduit n'importe où.

Définir les services qui composent l'application dans un fichier docker-compose.yml afin qu'ils puissent être exécutés ensemble dans un environnement isolé.

Enfin, exécutez `docker-compose up`. Compose va démarrer et exécuter votre application entière



# Docker Compose

Compose fournit des commandes pour gérer tout le cycle de vie des applications:

Démarrer, arrêter et reconstruire des services

Afficher l'état des services en cours d'exécution

Fournir les journaux des services en cours d'exécution

# Installation Docker Compose

- Avec PIP

```
#yum install -y python-pip
```

```
#pip install docker-compose
```

# Premier service Docker Compose

- Un Hello word

```
#mkdir hello-world
```

```
#cd hello-world
```

```
#vi docker-compose.yml  
my-test:  
  image: hello-world
```

```
#docker-compose up
```

# LAMP avec Docker Compose

- Un Hello word

```
#mkdir
```

```
#cd redis
```

```
#vi docker-compose.yml
```

Merci de votre attention

# Kubernetes

Kubernetes est le système de cluster open source de Google dédié à la gestion de conteneurs Linux

Déploiement,

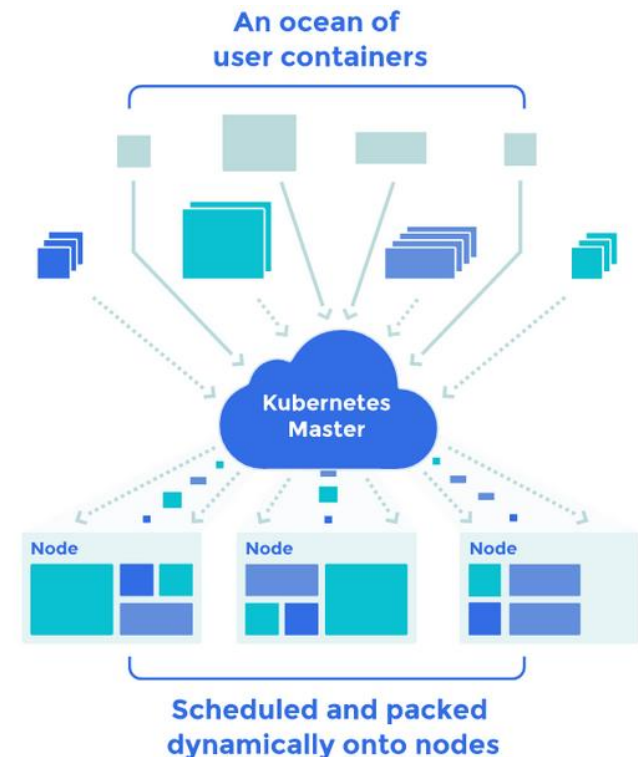
Maintenance,

Planification,

Équilibrage de charge,

Scaling - élasticité,

Réplication

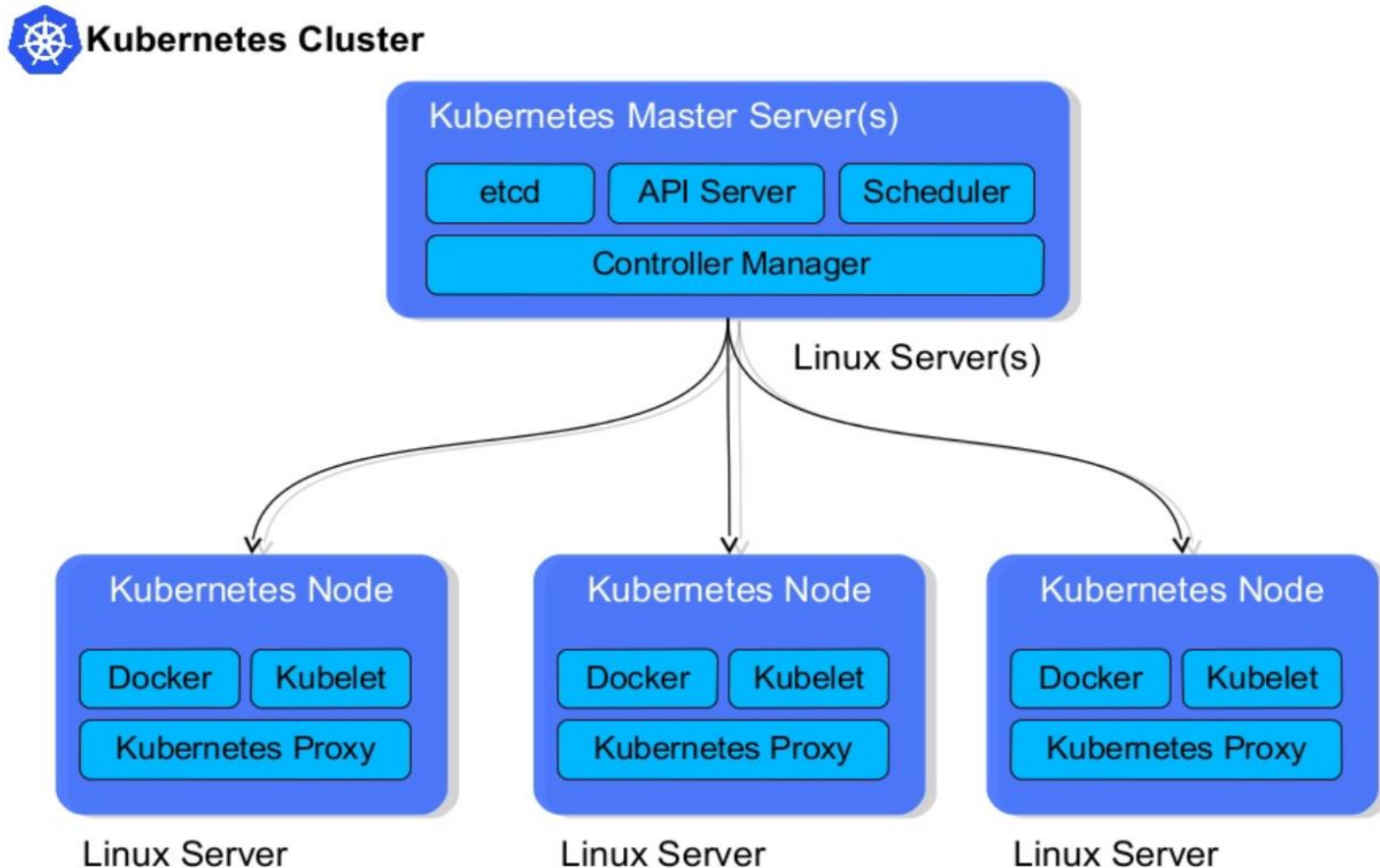


# Kubernetes

- Kubernetes exécute les conteneurs dans des « pods ».
- sont les plus petites unités déployables qui peuvent être créés, planifiés et gérés avec Kubernetes.



# Architecture Kubernetes

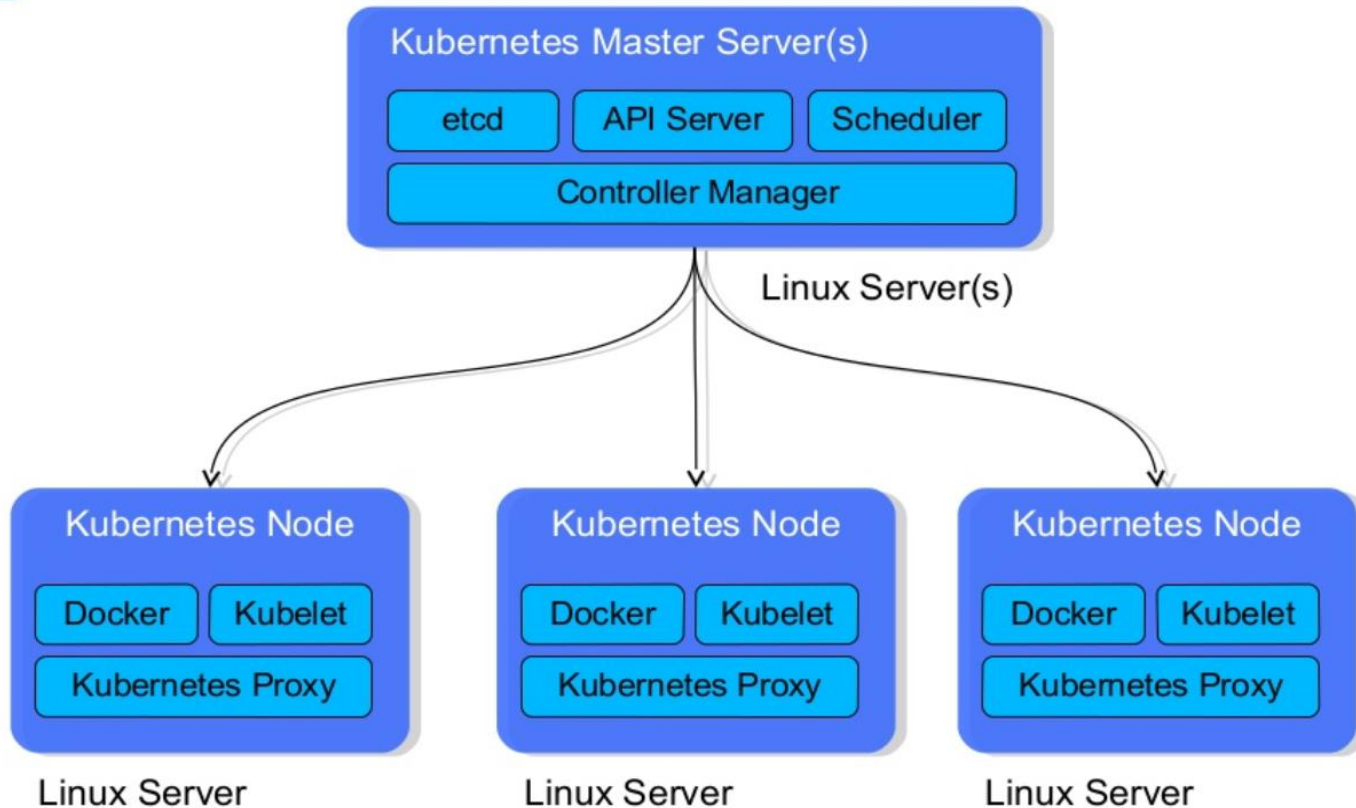




# Architecture Kubernetes



**Kubernetes Cluster**

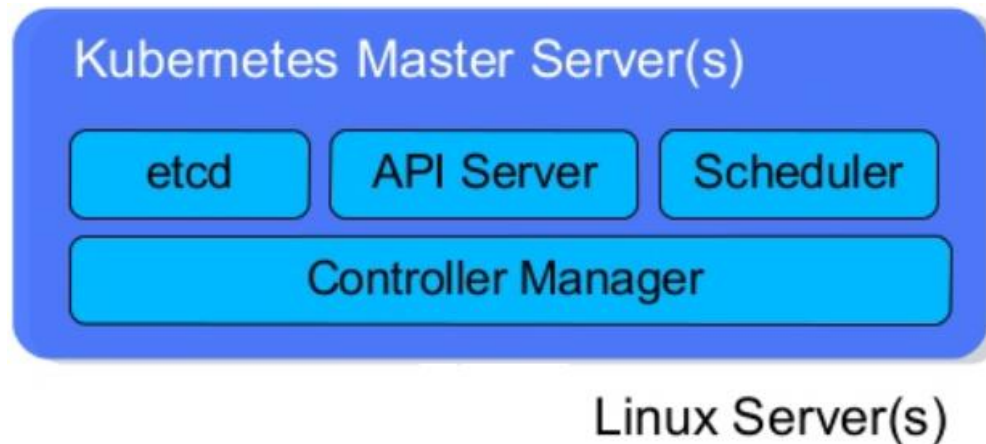


# Exploitation des containers

Est un ensemble des services contrôlés par Kubernetes.

Ces services fonctionnent sur un seul serveur.

Ils peuvent être exécutés sur un ou plusieurs serveurs derrière un équilibreur de charge dans le cadre de haute disponibilité



# Exploitation des containers

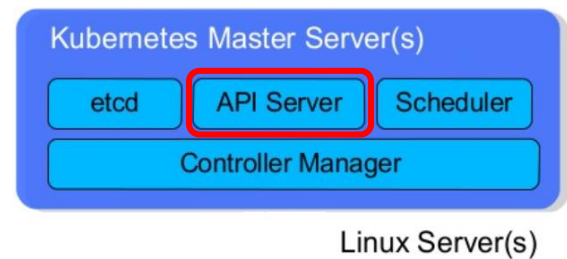
**Le serveur API** : est le point central de gestion de l'ensemble du cluster

Il permet à l'administrateur de configurer les charges de travail et les unités organisationnelles.

Il est également responsable de veiller que les services de conteneurs déployés soient validés avec ETCD

Il valide et configure les données pour les pods, les services et les contrôleurs de réplication.

Il distribue également les pods sur les différents nœuds et synchronise les pods avec les services.



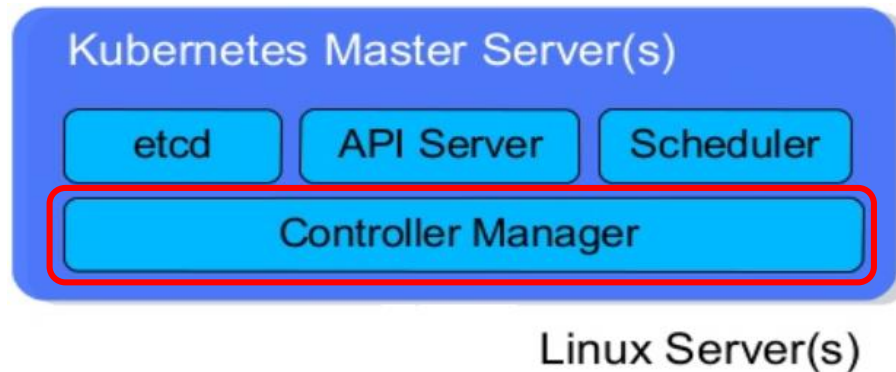
# Architecture Kubernetes

## Le service de gestion du contrôleur :

Il met en œuvre la procédure de réplication

Il gère les processus de réplication des pods.

Il fait le Scaling (mise à l'échelle) du groupe d'applications



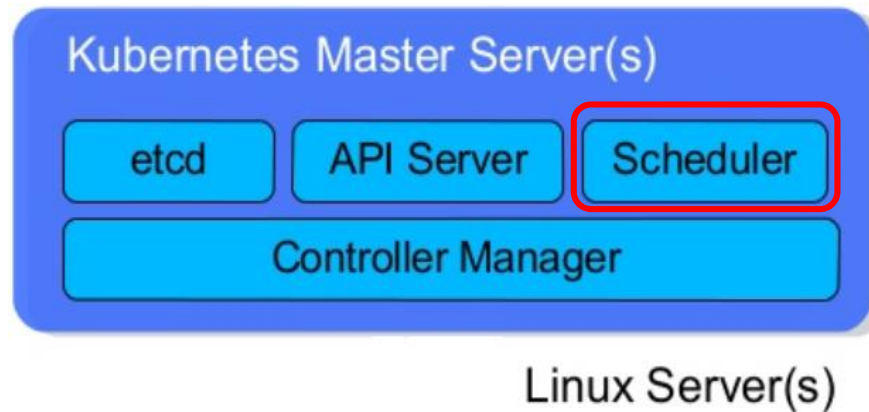
# Exploitation des containers

## L'ordonnanceur :

Il attribue la charge de travail aux nœuds du cluster.

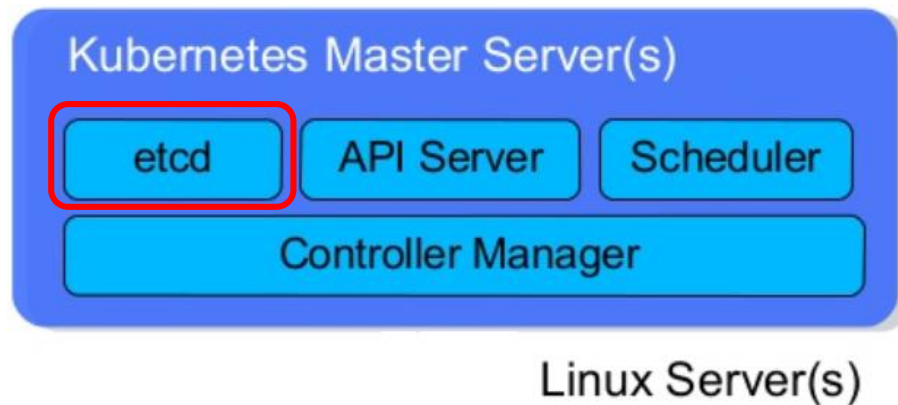
Il analyse l'environnement et la charge de travail (Workload)

Il place la charge de travail sur un nœud ou sur un autre nœud.



# Architecture Kubernetes

**ETCD** : est un stockage de paires valeur/clé distribuée,  
Fournit un moyen fiable de stocker des données dans un cluster,  
Les applications peuvent lire et écrire des données dans ETCD,  
Ces valeurs peuvent être surveillées, permettant à votre application de se reconfigurer.



# Architecture Kubernetes

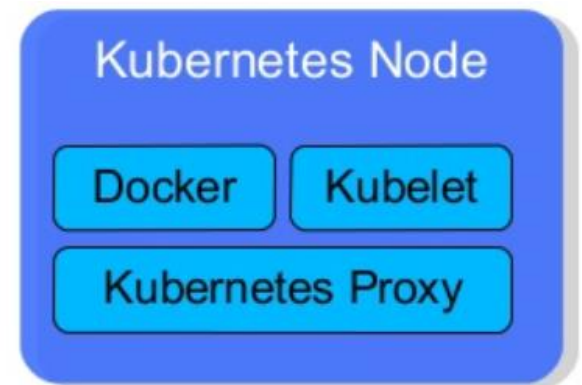
Les nœuds ou minions ou nodes comme ils sont souvent appelés

Exécute les Pods

Le nœud gère plusieurs services importants :

**kubelet**

**kube-proxy**



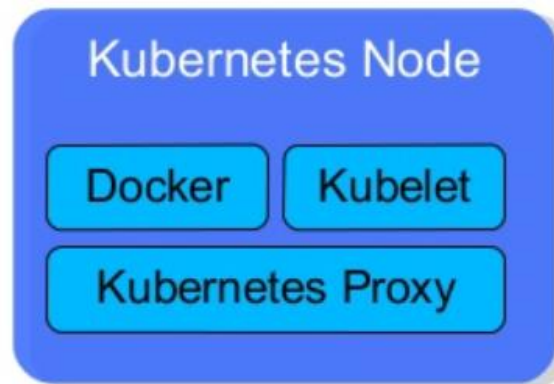
Linux Server

# Architecture Kubernetes

**Kubelet** : est responsable de la gestion des pods

Le démon écoute le serveur API maître et s'assure que les conteneurs locaux soient exécutés, restent en « bonne santé », et de leur états

**kube-proxy** : exécuté sur chaque nœud en un proxy réseau simple et d'équilibrage de charge pour les services sur ce nœud.



Linux Server



# Architecture Kubernetes

**Les Pods :** Kubernetes exécute les conteneurs dans des « pods ».

Ils sont les plus petites unités déployables qui peuvent être créées, planifiées et gérées avec Kubernetes.

Ils ne disposent pas d'un cycle de vie géré, Si ils « meurent », ils ne sont pas recréés.

Il est recommandé d'utiliser un contrôleur de réplication



# Architecture Kubernetes

## Les Pods

Les applications dans le pod utilisent toutes le même espace de noms de réseau, adresse IP, et port

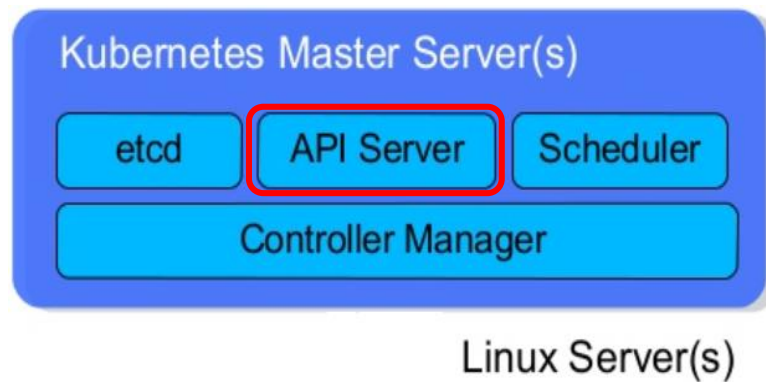
Ils disposent d'une adresse IP dans un espace réseau partagé. (Flanneld)

Ils partagent des ressources



# **Configuration du serveur maitre**

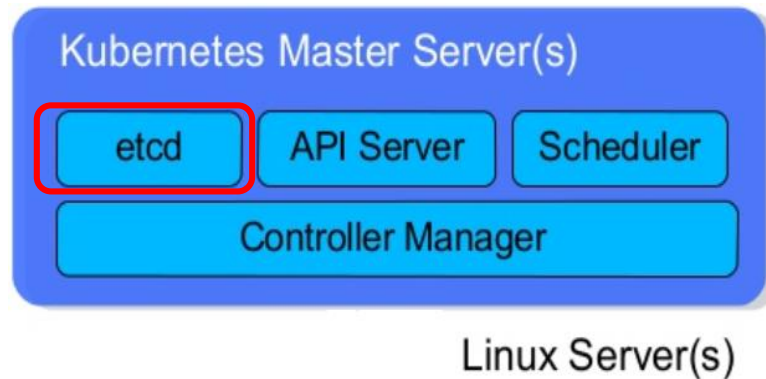
# Kube-API server



- Configuration de l'apiserver

```
#vi /etc/kubernetes/apiserver  
KUBE_API_ADDRESS="--address=0.0.0.0"  
KUBE_ETCD_SERVERS="--etcd_servers=http://master:2379"
```

# ETCD



- Configuration de ETCD

```
#vi /etc/etcd/etcd.conf  
ETCD_LISTEN_CLIENT_URLS="http://0.0.0.0:2379"  
ETCD_ADVERTISE_CLIENT_URLS="http://master.ipssi.local:2379"
```

# Kube-API server



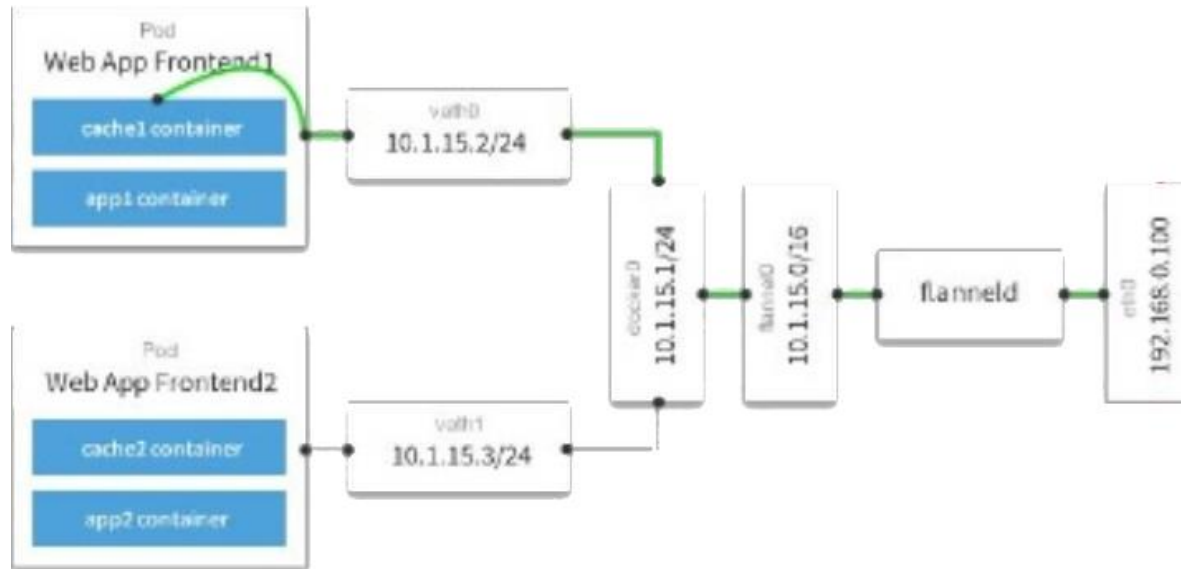
- Activation et démarrage des services

```
#for SERVICES in etcd kube-apiserver kube-controller-manager kube-scheduler; do
systemctl restart $SERVICES
systemctl enable $SERVICES
systemctl status $SERVICES
Done
```

- Activation et démarrage des services

```
#kubectl cluster-info
Kubernetes master is running at http://localhost:8080
```

# Flannel méthode 1



- Configuration flannel dans etcd

```
#etcdctl mk /atomic.io/network/config etcdctl mk /atomic.io/network/config  
'{"Network":"172.17.0.0/16","SubnetLen":"24","Backend":{"Type": "vxlan"}}'
```

```
#etcdctl get /atomic.io/network/config
```

# Flannel méthode 2

- Configuration flannel dans etcd

```
#cat flannel.json
```

```
{  
  "Network": 172.17.0.0/16,  
  "SubnetLen": 24,  
  "Backend": {  
    "Type": vxlan  
  }  
}
```

```
#curl -L http://localhost:2379/v2/keys/atomic.io/network/config -XPUT --data-urlencode  
value@flanneld.json
```

```
{"action": "set", "node":  
{"key": "/coreos.com/network/config", "value": "{\n\"Network\":
```

```
.....
```

```
#etcdctl get /atomic.io/network/config
```



# Flannel

- Configuration flanneld

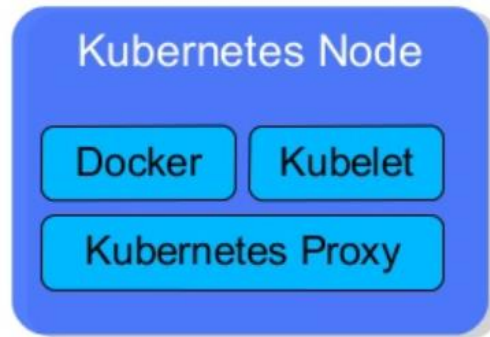
```
#cat /etc/sysconfig/flanneld
```

```
# etcd url location. Point this to the server where etcd runs  
FLANNEL_ETCD=http://master.ipssi.local:2379
```

```
#systemctl enable flanneld  
#reboot
```

# **Configuration des nodes**

# Kubelet



Linux Server

- Configuration de kubelet

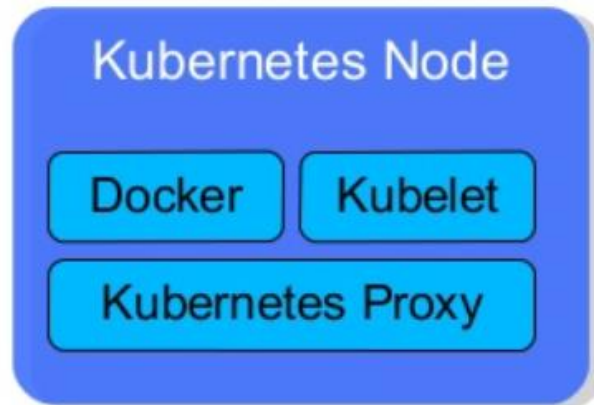
```
#cat /etc/kubernetes/kubelet
```

```
KUBELET_ADDRESS="--address=0.0.0.0"
```

```
KUBELET_HOSTNAME="--hostname-override=master"
```

```
KUBELET_API_SERVER="--api_servers=http://master:8080"
```

# Kubernetes maitre



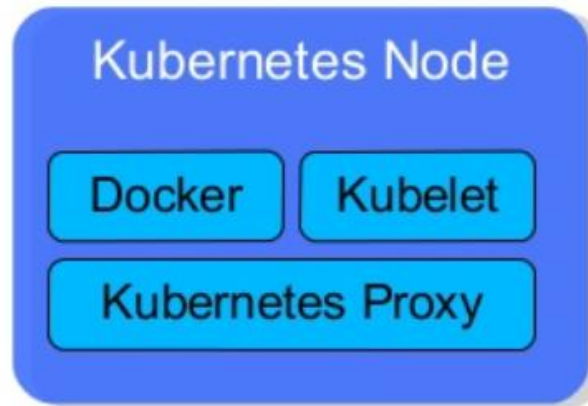
Linux Server

- Configuration de kubernetes

```
#vi /etc/kubernetes/config
```

```
KUBE_MASTER="--master=http://master.ipssi.local:8080"
```

# Kubelet

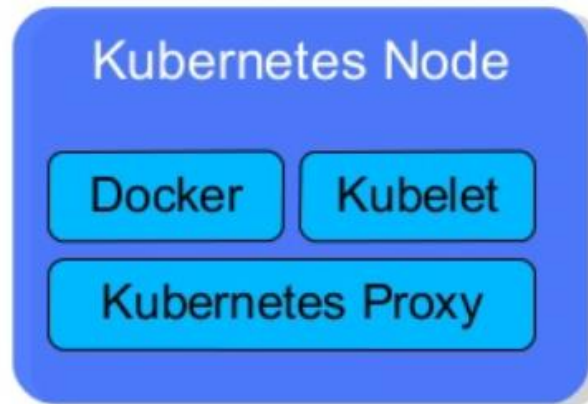


Linux Server

- Configuration de kubelet

```
#vi /etc/kubernetes/kubelet  
KUBELET_ADDRESS="--address=0.0.0.0"  
KUBELET_HOSTNAME="--hostname-override=node0"  
KUBELET_ARGS="--register-node=true"  
KUBELET_API_SERVER="--api_servers=http://master.ipssi.local:8080"
```

# Service

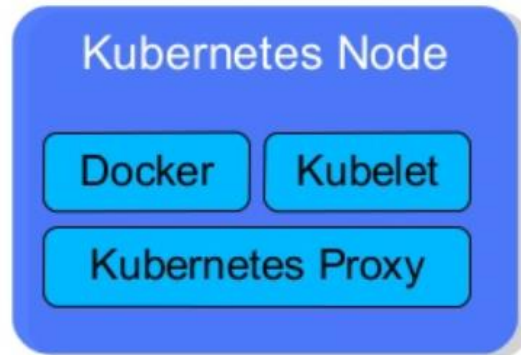


Linux Server

- Activation et restart des services

```
#for SERVICES in docker kube-proxy.service kubelet.service; do
systemctl restart $SERVICES
systemctl enable $SERVICES
systemctl status $SERVICES
done
```

# Flanneld



Linux Server

- Configuration du service flanneld

```
#vi /etc/sysconfig/flanneld
```

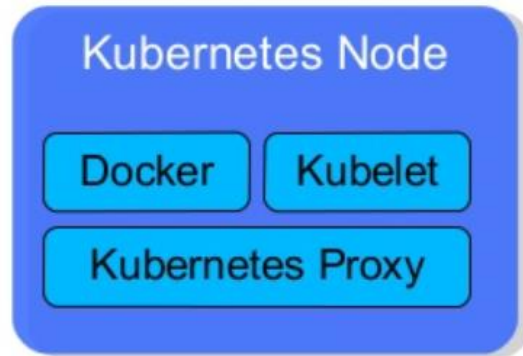
```
FLANNEL_ETCD=http://master.ipssi.local:2379
```

```
# systemctl start flanneld
```

```
# systemctl enable flanneld
```

```
# systemctl reboot
```

# Get nodes



Linux Server

- Vérification de la config

```
#kubectl get nodes
```

NAME	LABELS	STATUS
Master	kubernetes.io/hostname=master	Ready
Node1	kubernetes.io/hostname=node1	Ready
Node2	kubernetes.io/hostname=node2	Ready



# Mon premier Pod

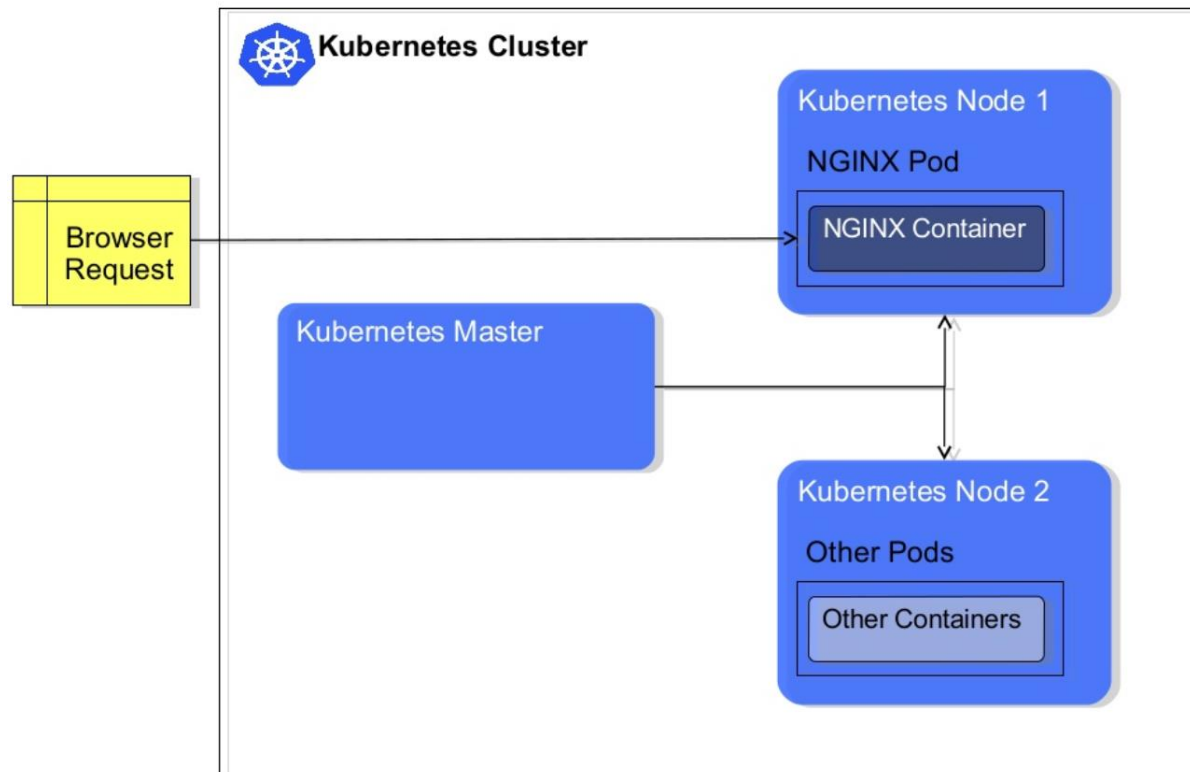
Les Pods sont les plus petites unités déployables. qui peuvent être créées, planifiées et gérées avec Kubernetes.

Les Pods peuvent être créés individuellement.

Les pods ne disposent pas d'un cycle de vie géré, s'ils meurent, ils ne seront pas recréés.

Kubernetes crée les pods avec des fichiers de configuration (appelés manifests) qui peuvent être soit YAML ou JSON.

# Mon premier Pod



# Mon premier Pod

- Création d'un Pod nginx

```
#vi nginx.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx-server
    image: nginx
    ports:
    - containerPort: 80
```

# Mon premier Pod

- Création d'un Pod nginx

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

```
#Kuberctl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx 1/1	Running	0	1m	

```
#kubectl describe pod nginx
```

Name: nginx

Namespace: default

Image(s):

nginx Node: 192.168.10.152/192.168.10.151

Labels: <none>

Status: Running

IP: 10.244.3.2

Replication Controllers: <none>

Containers: nginx:

# Mon premier Pod

- Suppression du Pod nginx

```
#kubectl delete pod nginx
```

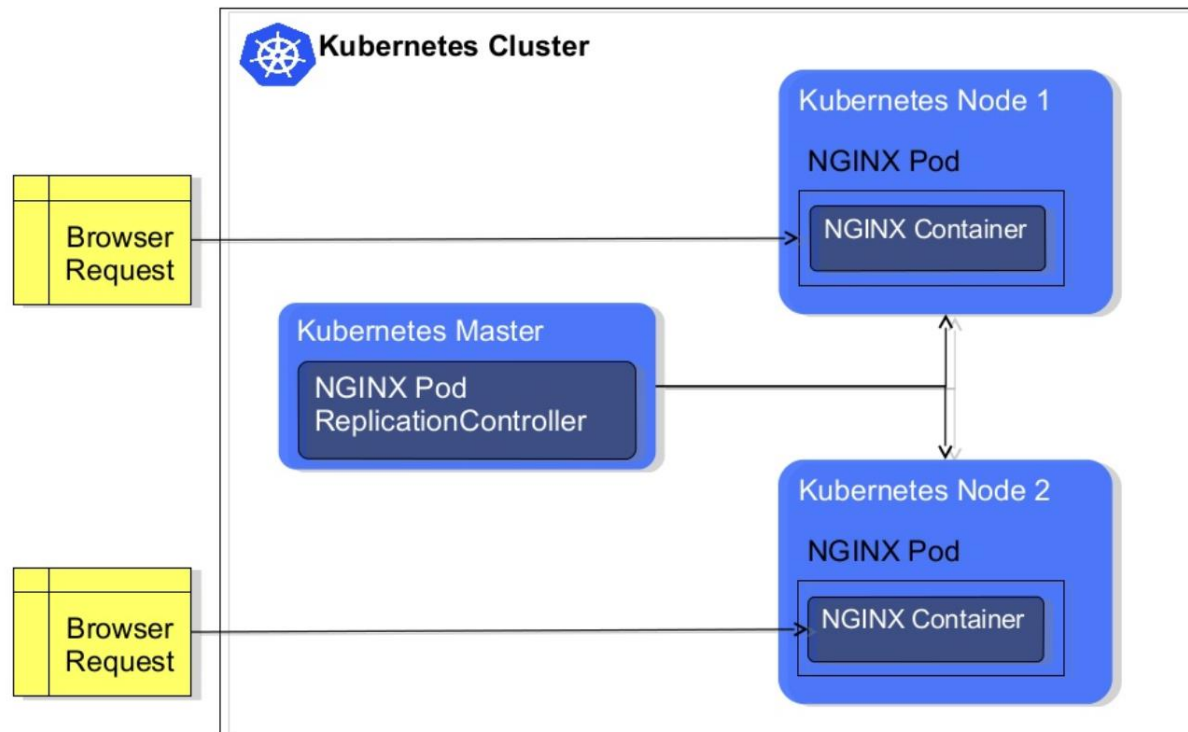
```
#Kuberctl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
------	-------	--------	----------	-----

# Réplication multi niveaux

- Les contrôleurs de réplication ou RC :
  - Ils gèrent le cycle de vie des pods.
  - Ils assurent qu'un certain nombre de pods spécifiques sont en cours d'exécution à un moment donné.
  - Ils le font en créant ou en supprimant les pods au besoin.
- Il est recommandé d'utiliser un contrôleur de réplication même si on crée un seul pod.

# Réplication multi niveaux



# Réplication multi niveaux

- Création d'un réplicateur nginx

```
#vi nginx-rc.yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```



# Réplication multi niveaux

- Création d'un Pod nginx

```
#kubectl create -f nginxrc.yaml  
replicationcontrollers/my-nginx
```

```
#Kuberctl get rc
```

CONTROLLER	CONTAINER(S)	IMAGE(S)	SELECTOR	REPLICAS
my-nginx	nginx	nginx	app=nginx	1

# Réplication multi niveaux

- Évolution du réplicateur

```
#kubectl scale --replicas=3 rc my-nginx
```

```
#Kuberctl get rc
```

CONTROLLER	CONTAINER(S)	IMAGE(S)	SELECTOR	REPLICAS
my-nginx	nginx	nginx	app=nginx	3

```
#Kuberctl get pod
```

```
...
```

# Équilibrage de charge

Les services fournissent une adresse IP et un nom unique à un ensemble de Pods .

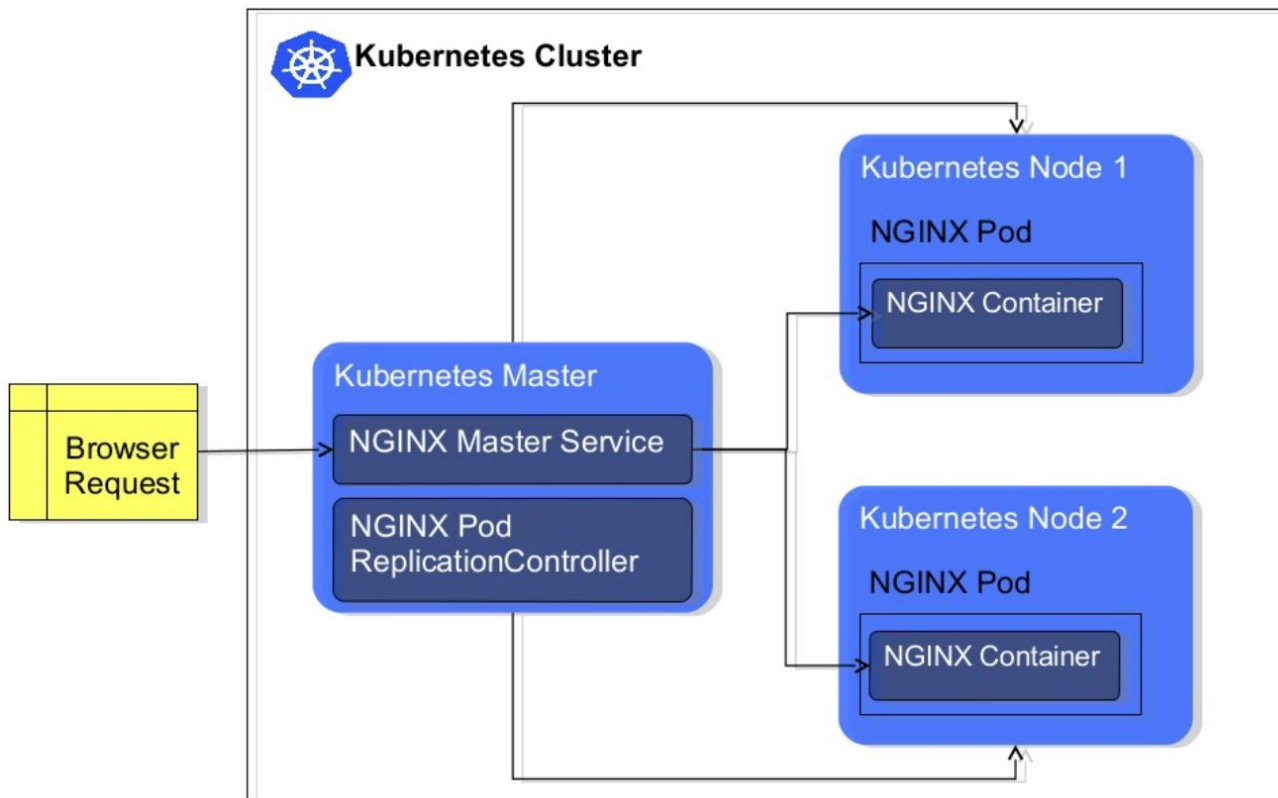
Le service agit comme un équilibreur de charge (load balancing).

Un service est lié à un contrôleur de réplication.

Chaque service se voit assigné une adresse IP virtuelle.

Le service permet de conserver l'état des pods créés par le contrôleur de réplication, et distribue les requêtes

# Équilibrage de charge



# Équilibrage de charge

- Création d'un service load blancing pour le réplicateur nginx

```
#vi nginx-rc-svc.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginxrcsvc
  labels:
    app: nginxsrcsvc
spec:
  ports:
    - port: 80
      protocol: TCP
  selector:
    app: nginx-rc
```

# Équilibrage de charge

- Création d'un Pod nginx

```
#kubectl create -f nginxrc-sc.yaml
```

```
#kubectl get svc nginxrcsvc
```

NAME	LABELS	SELECTOR	IP(S)	PORT(S)
nginxsvc	app=nginx	app=nginx	10.100.168.7	80/TCP