FORMATION CONTENEURS

CONCERNANT CES SUPPORTS DE COURS

SUPPORTS DE COURS RÉALISÉS PAR PARTICULE

- ex Osones/Alterway
- Expertise Cloud Native
- Expertise DevOps
- Nos offres de formations: https://particule.io/trainings/
- Sources : https://github.com/particuleio/formations/
- HTML/PDF : https://particule.io/formations/

COPYRIGHT

OBJECTIFS DE LA FORMATION : CLOUD

- Comprendre les principes du cloud et son intérêt
- Connaitre le vocabulaire inhérent au cloud
- Avoir une vue d'ensemble sur les solutions existantes en cloud public et privé
- Posséder les clés pour tirer parti au mieux du IaaS
- Pouvoir déterminer ce qui est compatible avec la philosophie cloud ou pas
- Adapter ses méthodes d'administration système et de développement à un environnement cloud

LE CLOUD, VUE D'ENSEMBLE

CARACTÉRISTIQUES

Fournir un (des) service(s)...

- Self service
- À travers le réseau
- Mutualisation des ressources
- Élasticité rapide
- Mesurabilité

Inspiré de la définition du NIST
https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpubl
145.pdf

SELF SERVICE

- L'utilisateur accède *directement* au service
- Pas d'intermédiaire humain
- Réponses immédiates
- Catalogue de services permettant leur découverte

À TRAVERS LE RÉSEAU

- L'utilisateur accède au service à travers le réseau
- Le *fournisseur* du service est distant du *consommateur*
- Réseau = internet ou pas
- Utilisation de protocoles réseaux standards (typiquement : HTTP)

MUTUALISATION DES RESSOURCES

- Un cloud propose ses services à de multiples utilisateurs/organisations (*multi-tenant*)
- *Tenant* ou *projet* : isolation logique des ressources
- Les ressources sont disponibles en grandes quantités (considérées illimitées)
- Le taux d'occupation du cloud n'est pas visible
- La localisation précise des ressources n'est pas visible

ÉLASTICITÉ RAPIDE

- Provisionning et suppression des ressources quasi instantané
- Permet le *scaling* (passage à l'échelle)
- Possibilité d'automatiser ces actions de *scaling*
- Virtuellement pas de limite à cette élasticité

MESURABILITÉ

- L'utilisation des ressources cloud est monitorée par le fournisseur
- Le fournisseur peut gérer son *capacity planning* et sa facturation à partir de ces informations
- L'utilisateur est ainsi facturé en fonction de son usage précis des ressources
- L'utilisateur peut tirer parti de ces informations

MODÈLES

On distingue :

- modèles de service : IaaS, PaaS, SaaS
- modèles de déploiement : public, privé, hybride

IAAS

- *Infrastructure as a Service*
- Infrastructure :
- Compute (calcul)
- Storage (stockage)
- Network (réseau)
- Utilisateurs cibles : administrateurs (système, stockage, réseau)

PAAS

- *Platform as a Service*
- Désigne deux concepts :
- Environnement permettant de développer/déployer une application (spécifique à un langage/framework - exemple : Python/Django)
- Ressources plus haut niveau que l'infrastructure, exemple : BDD
- Utilisateurs cibles : développeurs d'application

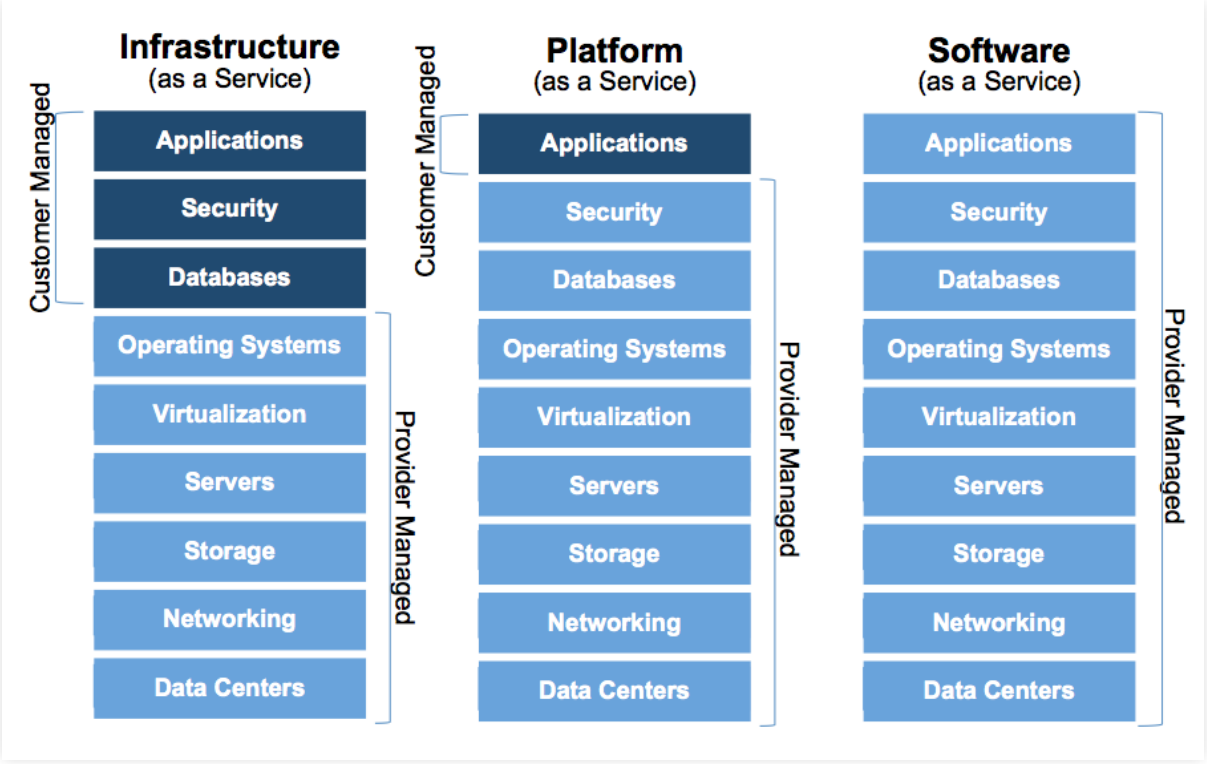
SAAS

- *Software as a Service*
- Utilisateurs cibles : utilisateurs finaux
- Ne pas confondre avec la définition *économique* du SaaS

QUELQUECHOSE AS A SERVICE ?

- Load balancing as a Service (Infra)
- Database as a Service (Platform)
- MonApplication as a Service (Software)
- etc.

LES MODÈLES DE SERVICE EN UN SCHÉMA



IaaS - PaaS - SaaS (source : Wikipedia)

CLOUD PUBLIC OU PRIVÉ ?

À qui s'adresse le cloud ?

- Public : tout le monde, disponible sur internet
- Privé : à une organisation, disponible sur son réseau

CLOUD HYBRIDE

- Utilisation mixte de multiples clouds privés et/ou publics
- Concept séduisant mais mise en œuvre a priori difficile
- Certains cas d'usages s'y prêtent très bien
- Intégration continue (CI)
- Motivations
- Éviter le *lock-in*
- Débordement (*cloud bursting*)

L'INSTANT VIRTUALISATION

Mise au point.

- La virtualisation est une technologie permettant d'implémenter la fonction *compute*
- Un cloud fournissant du compute *peut* utiliser la virtualisation
- Mais peut également utiliser :
- Du bare-metal
- Des containers (système)

LES APIS, LA CLÉ DU CLOUD

- Rappel : API pour *Application Programming Interface*
- Au sens logiciel : Interface permettant à un logiciel d'utiliser une bibliothèque
- Au sens cloud : Interface permettant à un logiciel d'utiliser un service (XaaS)
- Interface de programmation (via le réseau, souvent HTTP)
- Frontière explicite entre le fournisseur (provider) et l'utilisateur (user)
- Définit la manière dont l'utilisateur communique avec le cloud pour gérer ses ressources
- Gérer : CRUD (Create, Read, Update, Delete)

API REST

- Une ressource == une URI (*Uniform Resource Identifier*)
- Utilisation des verbes HTTP pour caractériser les opérations (CRUD)
- GET
- POST
- PUT
- DELETE
- Utilisation des codes de retour HTTP
- Représentation des ressources dans le corps des réponses HTTP

REST - EXEMPLES

```
GET http://endpoint/volumes/
GET http://endpoint/volumes/?size=10
POST http://endpoint/volumes/
DELETE http://endpoint/volumes/xyz
```

EXEMPLE CONCRET

```
GET /v2.0/networks/d32019d3-bc6e-4319-9c1d-6722fc136a22
{
 "network":{
  "status":"ACTIVE",
  "subnets":[ "54d6f61d-db07-451c-9ab3-b9609b6b6f0b" ],
  "name":"private-network",
  "provider:physical_network":null,
  "admin_state_up":true,
  "tenant_id":"4fd44f30292945e481c7b8a0c8908869",
  "provider:network_type":"local",
  "router:external":true,
  "shared":true,
  "id":"d32019d3-bc6e-4319-9c1d-6722fc136a22",
  "provider:segmentation_id":null
 }
}
```

POURQUOI LE CLOUD ? CÔTÉ ÉCONOMIQUE

- Appréhender les ressources IT comme des services “fournisseur”
- Faire glisser le budget “investissement” (Capex) vers le budget “fonctionnement” (Opex)
- Réduire les coûts en mutualisant les ressources, et éventuellement avec des économies d'échelle
- Réduire les délais
- Aligner les coûts sur la consommation réelle des ressources

POURQUOI LE CLOUD ? CÔTÉ TECHNIQUE

- Abstraire les couches basses (serveur, réseau, OS, stockage)
- S'affranchir de l'administration technique des ressources et services (BDD, pare-feux, load-balancing, etc.)
- Concevoir des infrastructures scalables à la volée
- Agir sur les ressources via des lignes de code et gérer les infrastructures "comme du code"

LE MARCHÉ

AMAZON WEB SERVICES (AWS), LE LEADER

AWS logo

- Lancement en 2006
- À l'origine : services web "e-commerce" pour développeurs
- Puis : d'autres services pour développeurs
- Et enfin : services d'infrastructure
- Récemment, SaaS

ALTERNATIVES IAAS PUBLICS À AWS

- Google Cloud Platform
- Microsoft Azure
- DigitalOcean
- Alibaba Cloud
- En France :
    - Scaleway
    - OVH
    - Outscale
    - Ikoula

FAIRE DU IAAS PRIVÉ

- OpenStack
- CloudStack
- Eucalyptus
- OpenNebula

OPENSTACK EN QUELQUES MOTS

OpenStack logo

- Naissance en 2010
- Fondation OpenStack depuis 2012...
- ...rebaptisée Open Infrastructure Foundation en 2020 (https://openinfra.dev/)
- Écrit en Python et distribué sous licence Apache 2.0
- Soutien très large de l'industrie et contributions variées

EXEMPLES DE PAAS PUBLIC

- Amazon Elastic Beanstalk (https://aws.amazon.com/fr/elasticbeanstalk)
- Google App Engine (https://cloud.google.com/appengine)
- Heroku (https://www.heroku.com)

SOLUTIONS DE PAAS PRIVÉ

- Cloud Foundry, Fondation (https://www.cloudfoundry.org)
- OKD, Red Hat (https://www.okd.io)
- Solum, OpenStack (https://wiki.openstack.org/wiki/Solum)

LES CONCEPTS INFRASTRUCTURE AS A SERVICE

LA BASE

- Infrastructure :
- Compute
- Storage
- Network

RESSOURCES *COMPUTE*

- Instance
- Image
- Flavor (gabarit)
- Paire de clé (SSH)

INSTANCE

- Dédiée au compute
- Durée de vie typiquement courte, à considérer comme éphémère
- Ne doit pas stocker de données persistantes
- Disque racine non persistant
- Basée sur une image

IMAGE CLOUD

- Image disque contenant un OS déjà installé
- Instanciable à l'infini
- Sachant parler à l'API de metadata

API ... DE METADATA

- `http://169.254.169.254`
- Accessible depuis l'instance
- Fournit des informations relatives à l'instance
- Expose les *userdata*
- L'outil `cloud-init` permet d'exploiter cette API

FLAVOR (GABARIT)

- *Instance type* chez AWS
- Définit un modèle d'instance en termes de CPU, RAM, disque (racine), disque éphémère
- Le disque éphémère a, comme le disque racine, l'avantage d'être souvent local donc rapide

PAIRE DE CLÉ

- Clé publique + clé privée SSH
- Le cloud manipule et stocke la clé publique
- Cette clé publique est utilisée pour donner un accès SSH aux instances

RESSOURCES RÉSEAU 1/2

- Réseau L2
- Port réseau
- Réseau L3
- Routeur
- IP flottante
- Groupe de sécurité

RESSOURCES RÉSEAU 2/2

- Load Balancing as a Service
- VPN as a Service
- Firewall as a Service

RESSOURCES STOCKAGE

Le cloud fournit deux types de stockage

- Block
- Objet

STOCKAGE BLOCK

- **Volumes** attachables à une instance
- Accès à des raw devices type */dev/vdb*
- Possibilité d'utiliser n'importe quel système de fichiers
- Possibilité d'utiliser du LVM, du chiffrement, etc.
- Compatible avec toutes les applications existantes
- Nécessite de *provisionner* l'espace en définissant la taille du volume

DU STOCKAGE PARTAGÉ ?

- Le stockage block n'est **pas** une solution de stockage partagé comme NFS
- NFS se situe à une couche plus haute : système de fichiers
- Un volume est *a priori* connecté à une seule machine

"BOOT FROM VOLUME"

Démarrer une instance avec un disque racine sur un **volume**

- Persistance des données du disque racine
- Se rapproche du serveur classique

STOCKAGE OBJET

- API : faire du CRUD sur les données
- Pousser et retirer des **objets** dans un **container**/bucket
- Pas de hiérarchie, pas de répertoires, pas de système de fichiers
- Accès lecture/écriture uniquement par les APIs
- Pas de *provisioning* nécessaire
- L'application doit être conçue pour tirer parti du stockage objet

ORCHESTRATION

- Orchestrer la création et la gestion des ressources dans le cloud
- Définition de l'architecture dans un **template**
- Les ressources créées à partir du **template** forment la **stack**
- Il existe également des *outils* d'orchestration (plutôt que des *services*)

BONNES PRATIQUES D'UTILISATION

POURQUOI DES BONNES PRATIQUES ?

Deux approches :

- Ne pas évoluer
- Risquer de ne pas répondre aux attentes
- Se contenter d'un cas d'usage *test & dev*
- Adapter ses pratiques au cloud pour en tirer parti pleinement

HAUTE DISPONIBILITÉ (HA)

- Le control plane (les APIs) du cloud est HA
- Les ressources provisionnées ne le sont pas forcément

PET VS CATTLE

Comment considérer ses instances ?

- Pet
- Cattle

INFRASTRUCTURE AS CODE

Avec du code

- Provisionner les ressources d'infrastructure
- Configurer les dites ressources, notamment les instances

Le métier évolue : Infrastructure Developer

SCALING, PASSAGE À L'ÉCHELLE

- Scale out plutôt que Scale up
- Scale out : passage à l'échelle horizontal
- Scale up : passage à l'échelle vertical
- Auto-scaling
- Géré par le cloud
- Géré par un composant extérieur

APPLICATIONS CLOUD READY

- Stockent leurs données au bon endroit
- Sont architecturées pour tolérer les pannes
- Etc.

Cf. https://12factor.net/

DERRIÈRE LE CLOUD

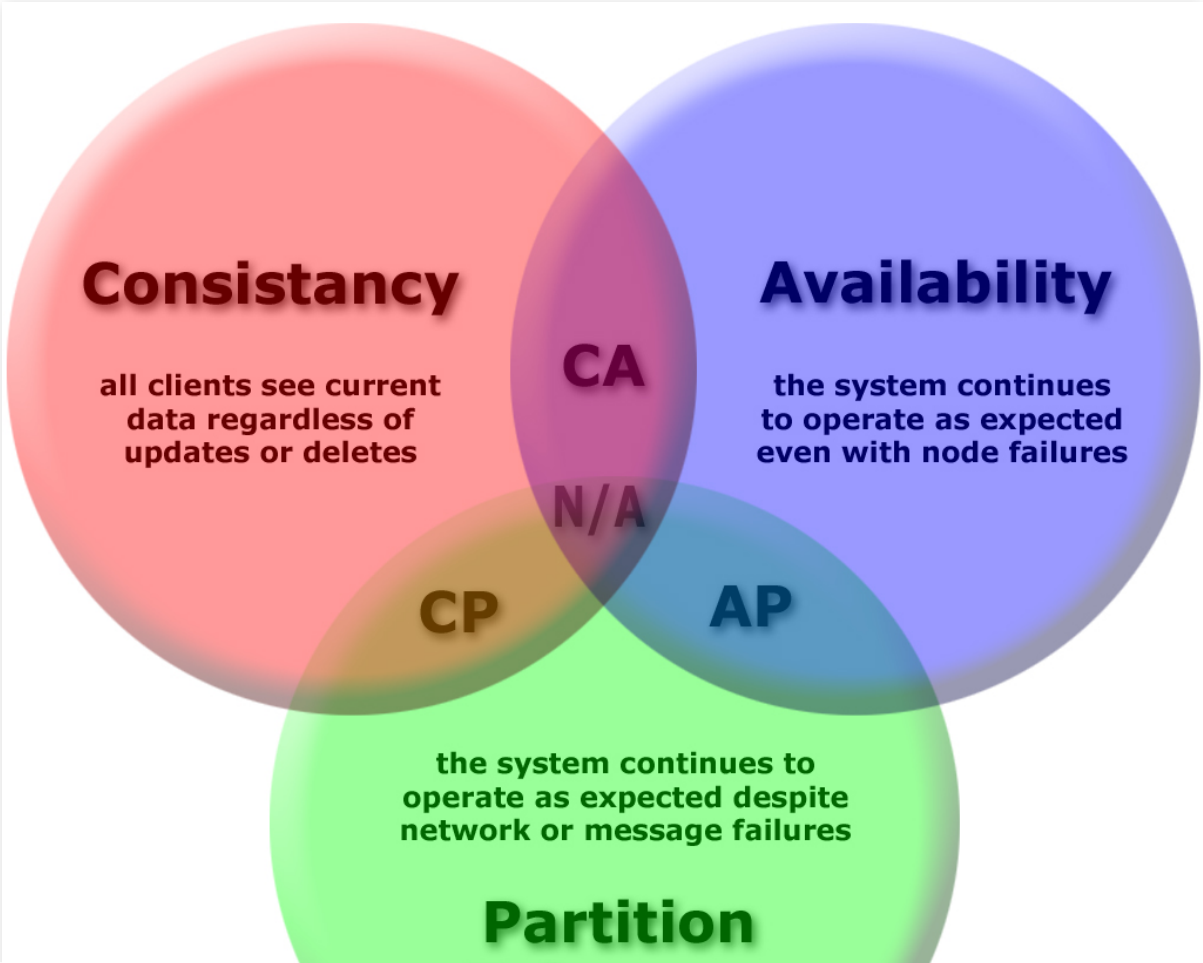
COMMENT IMPLÉMENTER UN SERVICE DE COMPUTE

- Virtualisation
- Containers (système)
- Bare metal

IMPLÉMENTATION DU STOCKAGE : (SOFTWARE DEFINED STORAGE) SDS

- **Attention** : ne pas confondre avec le sujet block vs objet
- Utilisation de commodity hardware
- Pas de RAID matériel
- Le logiciel est responsable de garantir les données
- Les pannes matérielles sont prises en compte et gérées
- Le projet **Ceph** et le composant **OpenStack Swift** implémentent du SDS
- Voir aussi Scality, OpenIO, OpenSDS,...

SDS - THÉORÈME CAP



Consistency - Availability - Partition tolerance

LE CLOUD : VUE D'ENSEMBLE

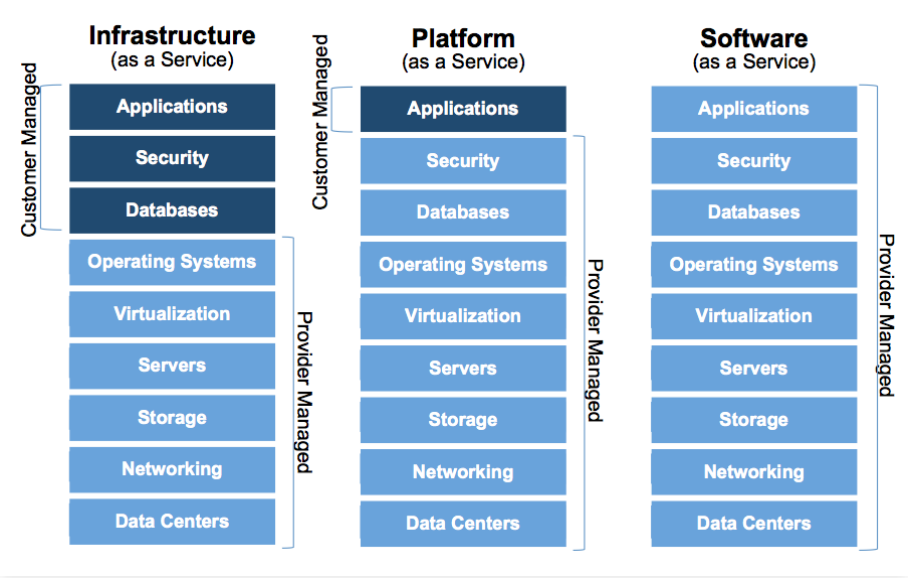
LE CLOUD, C'EST LARGE !

- Stockage/calcul distant (on oublie, cf. externalisation)
- Virtualisation++
- Abstraction du matériel (voire plus)
- Accès normalisé par des APIs
- Service et facturation à la demande
- Flexibilité, élasticité

WAAS : WHATEVER AS A SERVICE

- IaaS : Infrastructure as a Service
- PaaS : Platform as a Service
- SaaS : Software as a Service

LE CLOUD EN UN SCHÉMA



POURQUOI DU CLOUD ? CÔTÉ TECHNIQUE

- Abstraction des couches basses
- On peut tout programmer à son gré (API)
- Permet la mise en place d’architectures scalables

VIRTUALISATION DANS LE CLOUD

- Le cloud IaaS repose souvent sur la virtualisation
- Ressources compute : virtualisation
- Virtualisation complète : KVM, Xen
- Virtualisation conteneurs : OpenVZ, LXC, Docker, RKT

NOTIONS ET VOCABULAIRE IAAS

- L’instance est par définition éphémère
- Elle doit être utilisée comme ressource de calcul
- Séparer les données des instances

ORCHESTRATION DES RESSOURCES ?

- Groupement fonctionnel de ressources : micro services
- Infrastructure as Code : Définir toute une infrastructure dans un seul fichier texte de manière déclarative
- Scalabilité : passer à l'échelle son infrastructure en fonction de différentes métriques.

POSITIONNEMENT DES CONTENEURS DANS L'ÉCOSYSTÈME CLOUD ?

- Facilitent la mise en place de PaaS
- Fonctionnent sur du IaaS ou sur du bare-metal
- Simplifient la décomposition d'applications en micro services

LES CONTENEURS

DÉFINITION

- Les conteneurs fournissent un environnement isolé sur un système hôte, semblable à un chroot sous Linux ou une jail sous BSD, mais en proposant plus de fonctionnalités en matière d'isolation et de configuration. Ces fonctionnalités sont dépendantes du système hôte et notamment du kernel.

LE KERNEL LINUX

- Namespaces
- Cgroups (control groups)

LES NAMESPACES

MOUNT NAMESPACES (LINUX 2.4.19)

- Permet de créer un arbre des points de montage indépendants de celui du système hôte.

UTS NAMESPACES (LINUX 2.6.19)

- Unix Time Sharing : Permet à un conteneur de disposer de son propre nom de domaine et d'identité NIS sur laquelle certains protocoles tel que LDAP peuvent se baser.

IPC NAMESPACES (LINUX 2.6.19)

- Inter Process Communication : Permet d'isoler les bus de communication entre les processus d'un conteneur.

PID NAMESPACES (LINUX 2.6.24)

- Isole l’arbre d’exécution des processus et permet donc à chaque conteneur de disposer de son propre processus maître (PID 0) qui pourra ensuite exécuter et manager d'autres processus avec des droits illimités tout en étant un processus restreint au sein du système hôte.

USER NAMESPACES (LINUX 2.6.23-3.8)

- Permet l'isolation des utilisateurs et des groupes au sein d'un conteneur. Cela permet notamment de gérer des utilisateurs tels que l'UID 0 et GID 0, le root qui aurait des permissions absolues au sein d'un namespace mais pas au sein du système hôte.

NETWORK NAMESPACES (LINUX 2.6.29)

- Permet l'isolation des ressources associées au réseau, chaque namespace dispose de ses propres cartes réseaux, plan IP, table de routage, etc.

CGROUPS : CONTROL CROUPS

```
CGroup: /
           |--docker
           |   |--7a977a50f48f2970b6ede780d687e72c0416d9ab6e0b02030698c1633fdde95
           |   |--6807 nginx: master process ngin
           |   |   |--6847 nginx: worker proces
```

CGROUPS : LIMITATION DE RESSOURCES

- Limitation des ressources : des groupes peuvent être mis en place afin de ne pas dépasser une limite de mémoire.

CGROUPS : PRIORISATION

- Priorisation : certains groupes peuvent obtenir une plus grande part de ressources processeur ou de bande passante d'entrée-sortie.

CGROUPS : COMPTABILITÉ

- Comptabilité : permet de mesurer la quantité de ressources consommées par certains systèmes, en vue de leur facturation par exemple.

CGROUPS : ISOLATION

- Isolation : séparation par espace de nommage pour les groupes, afin qu'ils ne puissent pas voir les processus des autres, leurs connexions réseaux ou leurs fichiers.

CGROUPS : CONTRÔLE

- Contrôle : figer les groupes ou créer un point de sauvegarde et redémarrer.

DEUX PHILOSOPHIES DE CONTENEURS

- *Systeme* : simule une séquence de boot complète avec un init process ainsi que plusieurs processus (LXC, OpenVZ).
- *Process* : un conteneur exécute un ou plusieurs processus directement, en fonction de l'application conteneurisée (Docker, Rkt).

ENCORE PLUS “CLOUD” QU’UNE INSTANCE

- Partage du kernel
- Un seul processus par conteneur
- Le conteneur est encore plus éphémère que l’instance
- Le turnover des conteneurs est élevé : orchestration

CONTAINER RUNTIME

Permettent d'exécuter des conteneurs sur un système

- docker: historique
- containerd: implémentation de référence
- cri-o: implémentation Open Source développée par RedHat
- kata containers: Conteneurs dans des VMs

LES CONTENEURS: CONCLUSION

- Fonctionnalités offertes par le Kernel
- Les conteneurs engine fournissent des interfaces d'abstraction
- Plusieurs types de conteneurs pour différents besoins

LES CONCEPTS

UN ENSEMBLE DE CONCEPTS ET DE COMPOSANTS

- Layers
- Stockage
- Volumes
- Réseau
- Publication de ports
- Service Discovery

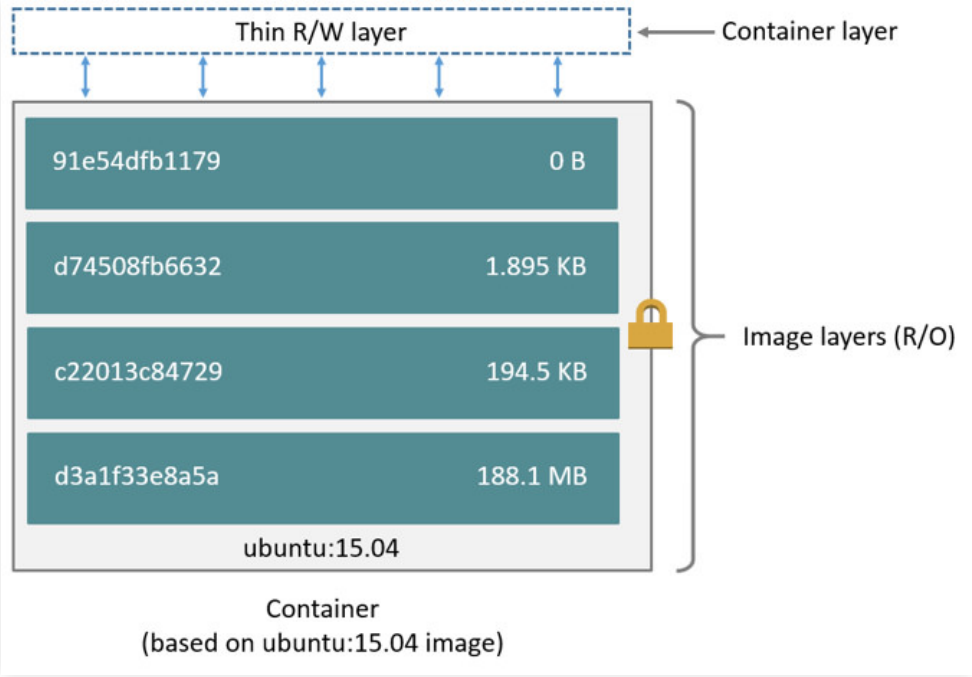
LAYERS

- Les conteneurs et leurs images sont décomposés en couches (layers)
- Les layers peuvent être réutilisés entre différents conteneurs
- Gestion optimisée de l'espace disque.

LAYERS : UNE IMAGE

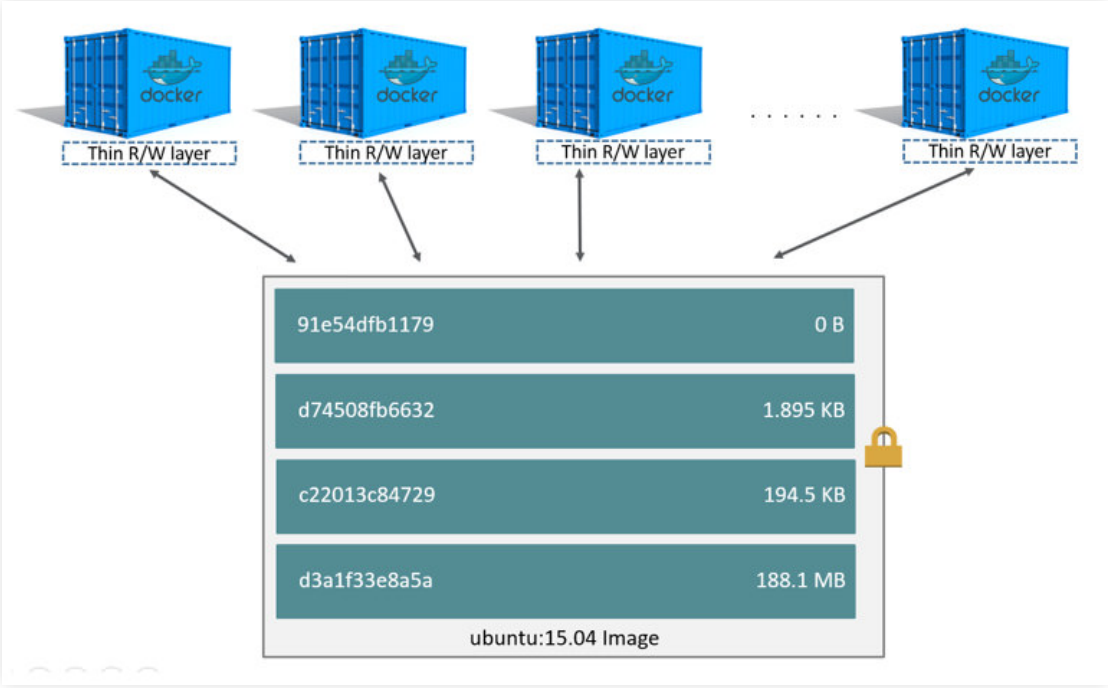
Une image se decompose en layers

LAYERS : UN CONTENEUR



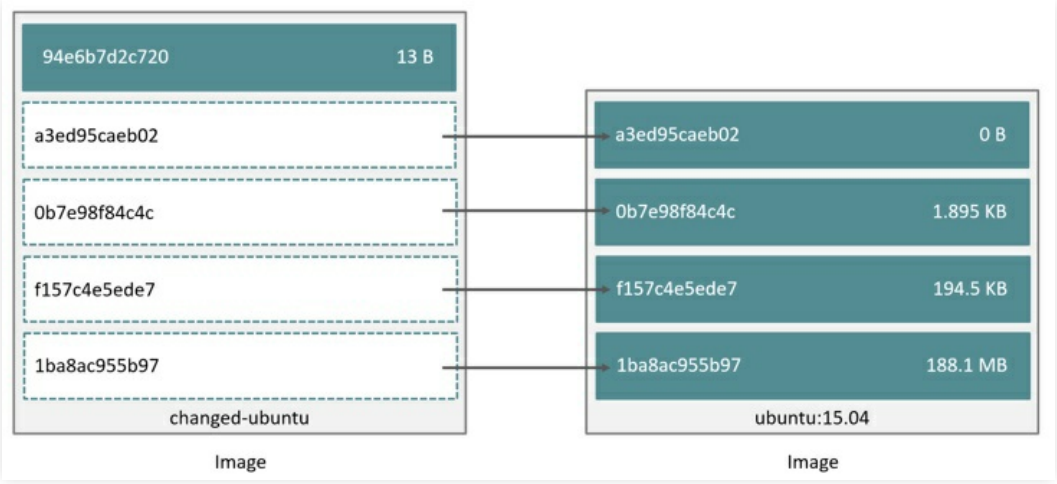
Une conteneur = une image + un layer r/w

LAYERS : PLUSIEURS CONTENEURS



Une image, plusieurs conteneurs

LAYERS : RÉPÉTITION DES LAYERS



Les layers sont indépendants de l'image

STOCKAGE

- Images Docker, données des conteneurs, volumes
- Multiples backends (extensibles via plugins):
  - AUFS
  - DeviceMapper
  - OverlayFS
  - NFS (via plugin Convoy)
  - GlusterFS (via plugin Convoy)

STOCKAGE : AUFS

- A unification filesystem
- Stable : performance écriture moyenne
- Non intégré dans le Kernel Linux (mainline)

STOCKAGE : DEVICE MAPPER

- Basé sur LVM
- Thin Provisionning et snapshot
- Intégré dans le Kernel Linux (mainline)
- Stable : performance écriture moyenne

STOCKAGE : OVERLAYFS

- Successeur de AUFS
- Performances accrues
- Relativement stable mais moins éprouvé que AUFS ou Device Mapper

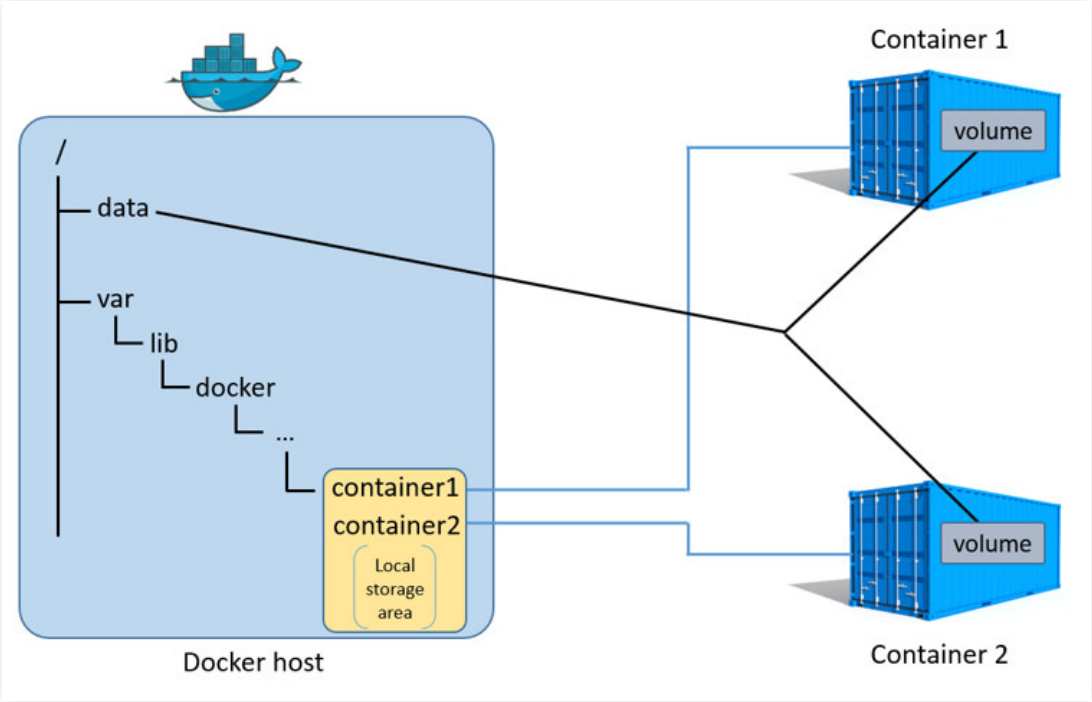
STOCKAGE : PLUGINS

- Étendre les drivers de stockages disponibles
- Utiliser des systèmes de fichier distribués (GlusterFS)
- Partager des volumes entre plusieurs hôtes docker

VOLUMES

- Assurent une persistance des données
- Indépendance du volume par rapport au conteneur et aux layers
- Deux types de volumes :
  - Conteneur : Les données sont stockées dans ce que l'on appelle un data conteneur
  - Hôte : Montage d'un dossier de l'hôte docker dans le conteneur
- Partage de volumes entre plusieurs conteneurs

VOLUMES : EXEMPLE



Un volume monté sur deux conteneurs

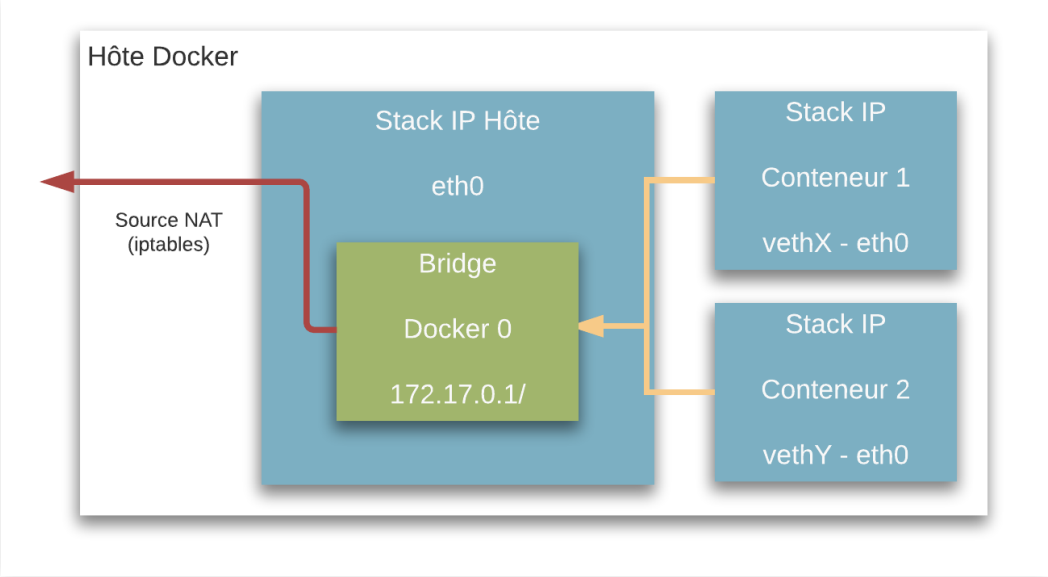
VOLUMES : PLUGINS

- Permettre le partage de volumes entre differents hôtes
- Fonctionnalité avancées : snapshot, migration, restauration
- Quelques exemples:
  - Convoy : multi-host et multi-backend (NFS, GlusterFS)
  - Flocker : migration de volumes dans un cluster

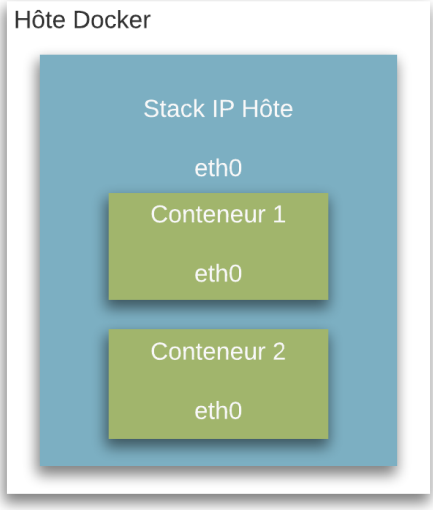
RÉSEAU : A LA BASE, PAS GRAND CHOSE...

```
NETWORK ID      NAME      DRIVER
42f7f9558b7a    bridge    bridge
6ebf7b150515    none      null
0509391a1fbd    host      host
```

RÉSEAU : BRIDGE



RÉSEAU : HOST


Hôte Docker
Stack IP Hôte
eth0
Conteneur 1
eth0
Conteneur 2
eth0

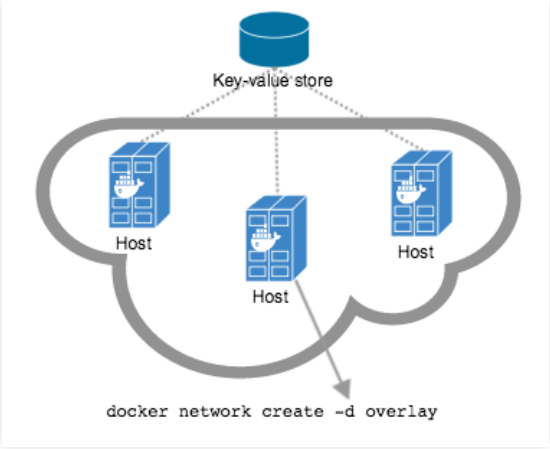
RÉSEAU : NONE

- Explicite

RÉSEAU : LES ÉVOLUTIONS

- Refactorisation des composants réseau (*libnetwork*)
- Système de plugins : multi host et multi backend (overlay network)
- Quelques exemples d'overlay :
  - Flannel : UDP et VXLAN
  - Weave : VXLAN

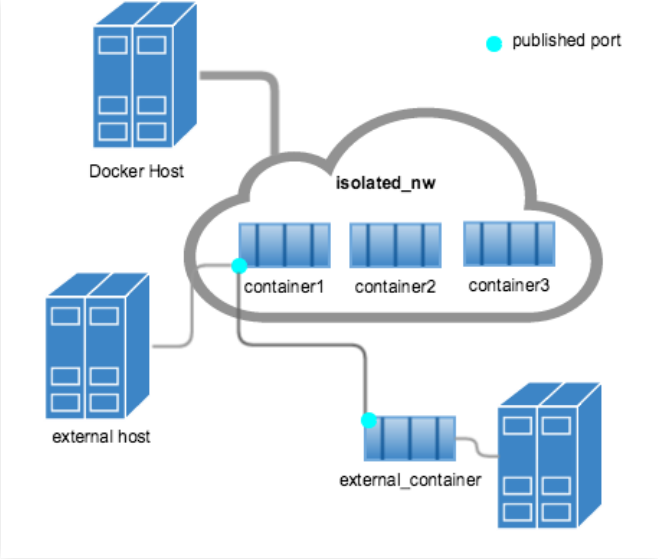
RÉSEAU : MULTIHOST OVERLAY



PUBLICATION DE PORTS

- Dans le cas d'un réseau diffèrent de l'hôte
- Les conteneurs ne sont pas accessible depuis l'extérieur
- Possibilité de publier des ports depuis l'hôte vers le conteneur (*iptables*)
- L'hôte sert de proxy au service

PUBLICATION DE PORTS



LINKS

- Les conteneurs d'un même réseau peuvent communiquer via IP
- Les liens permettent de lier deux conteneurs par nom
- Système de DNS rudimentaire (`/etc/hosts`)
- Remplacés par les *discovery services*

LES CONCEPTS: CONCLUSION

- Les composants de Docker sont modulaires
- Nombreuses possibilités offertes par défaut.
- Possibilité d'étendre les fonctionnalités via des plugins

BUILD, SHIP AND RUN !

BUILD

LE CONTENEUR ET SON IMAGE

- Flexibilité et élasticité
- Format standard de facto
- Instanciation illimitée

CONSTRUCTION D'UNE IMAGE

- Possibilité de construire son image à la main (long et source d'erreurs)
- Suivi de version et construction d'images de manière automatisée
- Utilisation de *Dockerfile* afin de garantir l'idempotence des images

DOCKERFILE

- Suite d'instruction qui définit une image
- Permet de vérifier le contenu d'une image

```
FROM alpine:3.4
MAINTAINER Particule <admin@particule.io>
RUN apk -U add nginx
EXPOSE 80 443
CMD ["nginx"]
```

DOCKERFILE : PRINCIPALES INSTRUCTIONS

- `FROM` : baseimage utilisée
- `RUN` : Commandes effectuées lors du build de l'image
- `EXPOSE` : Ports exposées lors du run (si `-P` est précisé)
- `ENV` : Variables d'environnement du conteneur à l'instanciation
- `CMD` : Commande unique lancée par le conteneur
- `ENTRYPOINT` : "Préfixe" de la commande unique lancée par le conteneur

DOCKERFILE : BEST PRACTICES

- Bien choisir sa baseimage
- Chaque commande Dockerfile génère un nouveau layer
- Comptez vos layers !

DOCKERFILE : BAD LAYERING

```
RUN apk --update add \
    git \
    tzdata \
    python \
    unrar \
    zip \
    libxslt \
    py-pip \

RUN rm -rf /var/cache/apk/*

VOLUME /config /downloads

EXPOSE 8081

CMD ["--datadir=/config", "--nolaunch"]

ENTRYPOINT ["/usr/bin/env","python2","/sickrage/SickBeard.py"]
```

DOCKERFILE : GOOD LAYERING

```
RUN apk --update add \
    git \
    tzdata \
    python \
    unrar \
    zip \
    libxslt \
    py-pip \
    && rm -rf /var/cache/apk/*

VOLUME /config /downloads

EXPOSE 8081

CMD ["--datadir=/config", "--nolaunch"]

ENTRYPOINT ["/usr/bin/env","python2","/sickrage/SickBeard.py"]
```

DOCKERFILE : DOCKERHUB

- Build automatisée d'images Docker
- Intégration GitHub / DockerHub
- Plateforme de stockage et de distribution d'images Docker

SHIP

SHIP : LES CONTENEURS SONT MANIPULABLES

- Sauvegarder un conteneur :

```
docker commit mon-conteneur backup/mon-conteneur
```

```
docker run -it backup/mon-conteneur
```

- Exporter une image :

```
docker save -o mon-image.tar backup/mon-conteneur
```

- Importer un conteneur :

```
docker import mon-image.tar backup/mon-conteneur
```

SHIP : DOCKER REGISTRY

- DockerHub n’est qu’au Docker registry ce que GitHub est à git
- Pull and Push
- Image officielle : `registry`

RUN

RUN : LANCER UN CONTENEUR

- docker run
- `-d` (detach)
- `-i` (interactive)
- `-t` (pseudo tty)

RUN : BEAUCOUP D'OPTIONS...

- `-v` /directory/host:/directory/container
- `-p` portHost:portContainer
- `-P`
- `-e` “VARIABLE=valeur”
- `--restart=always`
- `--name=mon-conteneur`

RUN : ...DONT CERTAINES UN PEU DANGEREUSES

- `--privileged` (Accès à tous les devices)
- `--pid=host` (Accès aux PID de l'host)
- `--net=host` (Accès à la stack IP de l'host)

RUN : SE “CONNECTER” À UN CONTENEUR

- docker exec
- docker attach

RUN : DÉTRUIRE UN CONTENEUR

- docker kill (SIGKILL)
- docker stop (SIGTERM puis SIGKILL)
- docker rm (détruit complètement)

BUILD, SHIP AND RUN : CONCLUSIONS

- Écosystème de gestion d'images
- Construction automatisée d'images
- Contrôle au niveau conteneurs

ÉCOSYSTÈME DOCKER

DOCKER INC.

- Docker Inc != Docker Project
- Docker Inc est le principal développeur du Docker Engine, Compose, Machine, Kitematic, Swarm etc.
- Ces projets restent OpenSource et les contributions sont possibles

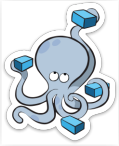
OCI

- Créé sous la Linux Fondation
- But : Créer des standards OpenSource concernant la manière de "runner" et le format des conteneurs
- Non lié à des produits commerciaux
- Non lié à des orchestrateurs ou des clients particuliers
- runC a été donné par Docker à l'OCI comme base pour leurs travaux

LES AUTRES PRODUITS DOCKER

DOCKER-COMPOSE

- Concept de stack
- Infrastructure as a code
- Scalabilité



DOCKER-COMPOSE

docker-compose.yml

```
nginx:
  image: nginx
  ports:
    - "80:80"
    - "443:443"
  volumes:
    - "/srv/nginx/etc/sites-enabled:/etc/nginx/sites-enabled"
    - "/srv/nginx/etc/certs:/etc/nginx/certs"
    - "/srv/nginx/etc/log:/var/log/nginx"
    - "/srv/nginx/data:/var/www"
```

DOCKER-MACHINE

- "Metal" as a Service
- Provisionne des hôtes Docker
- Abstraction du Cloud Provider

DOCKER SWARM

- Clustering : Mutualisation d'hôtes Docker
- Orchestration : placement intelligent des conteneurs au sein du cluster

AUTOUR DE DOCKER

- Plugins : étendre les fonctionnalités notamment réseau et volumes



  - Systèmes de gestion de conteneurs (COE)
  - Docker as a Service :
    - Docker Cloud
    - Tutum

ÉCOSYSTÈME DOCKER : CONCLUSION

- Le projet Docker est Open Source et n'appartient plus à Docker
- Des outils permettent d'étendre les fonctionnalités de Docker
- Docker Inc. construit des offres commerciales autour de Docker

DOCKER HOSTS

LES DISTRIBUTIONS TRADITIONNELLES

- Debian, CentOS, Ubuntu
- Supportent tous Docker
- Paquets DEB et RPM disponibles

UN PEU "TOO MUCH" NON ?

- Paquets inutiles ou out of date
- Services par défaut/inutiles alourdissent les distributions
- Cycle de release figé

OS ORIENTÉS CONTENEURS

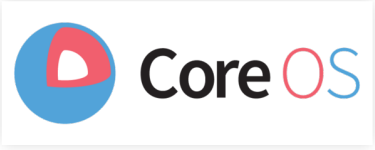
- Faire tourner un conteneur engine
- Optimisé pour les conteneurs : pas de services superflus
- Fonctionnalités avancées liées aux conteneurs (clustering, network, etc.)
- Sécurité accrue de part le minimalisme

OS ORIENTÉS CONTENEURS : EXEMPLES

- CoreOS (CoreOS)
- Atomic (Red Hat)
- RancherOS (Rancher)
- Photon (VMware)

COREOS : PHILOSOPHIE

- Trois "channels" : stable, beta, alpha
- Dual root : facilité de mise à jour
- Système de fichier en read only
- Pas de gestionnaire de paquets : tout est conteneurisé
- Forte intégration de *systemd*



COREOS : FONCTIONNALITÉS ORIENTÉES CONTENEURS

- Inclus :
  - Docker
  - Rkt
  - Etcd (base de données clé/valeur)
  - Flannel (overlay network)
  - Kubernetes Kubelet
- Permet nativement d'avoir un cluster complet
- Stable et éprouvé en production
- Idéal pour faire tourner Kubernetes (Tectonic)

COREOS : ETCD

- Système de stockage simple : clé = valeur
- Hautement disponible (quorum)
- Accessible via API

COREOS : FLANNEL

- Communication multi-hosts
- UDP ou VXLAN
- S'appuie sur un système clé/valeur comme etcd

RANCHEROS : PHILOSOPHIE

- Docker et juste Docker : Docker est le process de PID 1)
- Docker in Docker : Daemon User qui tourne dans le Daemon System
- Pas de processus tiers, pas d'init, juste Docker
- Encore en beta

FEDORA ATOMIC : PHILOSOPHIE

- Équivalent à CoreOS mais basée sur Fedora
- Utilise *systemd*
- Update Atomic (incrémentielles pour rollback)

PROJECT PHOTON

- Développé par VMware mais Open Source
- Optimisé pour vSphere
- Supporte Docker, Rkt et Pivotal Garden (Cloud Foundry)

DOCKER HOSTS : CONCLUSION

- Répond à un besoin différent des distributions classiques
- Fourni des outils et une logique adaptée aux environnements full conteneurs
- Sécurité accrue (mise à jour)

OBJECTIFS DE LA FORMATION : TIRER AVANTAGE DU CLOUD PAR LES BONNES PRATIQUES

- Comprendre les bonnes pratiques en matière d'infrastructures hébergées dans le Cloud
- Etre capable de discerner ce qui est cloud-compatible et ce qui ne l'est pas
- Comprendre les bonnes pratiques en matière de conception d'applications destinées au Cloud
- Adapter au Cloud ses méthodes d'administration système
- Anticiper l'impact sur les métiers et prévoir les changements dans l'organisation

DES BONNES PRATIQUES : POUR QUOI FAIRE ?

- Déployer les applications sur n'importe quel cloud
- Utiliser pleinement les services du cloud
- Minimiser le coût et le temps d'intégration des nouveaux équipiers
- Mettre l'application à l'échelle sans remise en cause lourde
- Entrer dans le cercle vertueux des MEP agiles
- Optimiser la facture

CONCEVOIR UNE APPLICATION POUR LE CLOUD

LA RÉFÉRENCE : LES 12 FACTEURS

"The Twelve-Factor App" https://12factor.net/fr/

- Publié par Heroku https://www.heroku.com/what/
- Ecrit par des développeurs, des architectes et des ops
- Recueil de préconisations techniques issues d'expériences terrain
- Destiné aux développeurs et aux personnes en charge du déploiement et du Run
- Applicable quel que soit le langage de programmation

LES 12 FACTEURS EN DÉTAILS (1/2)

1. Base de code : unique, suivie dans un VCS, plusieurs déploiements
2. Dépendances : les isoler et les déclarer explicitement
3. Configuration : différencier les environnements via les variables de conf.
4. Services externes : les traiter comme des ressources attachées
5. Build, release, run : séparer strictement les étapes d'assemblage et d'exécution
6. Processus : exécuter l'application comme un ou plusieurs processus sans état

LES 12 FACTEURS EN DÉTAILS (2/2)

7. Ports : exporter les services de l'application via des ports TCP
8. Mise à l'échelle : utiliser le modèle de processus
9. Jetable : maximisez la robustesse avec des démarrages rapides et des arrêts en douceur
10. Parité dev/prod : gardez les différents environnements aussi proches que possible
11. Logs : les traiter comme des flux d’évènements
12. Processus d’administration et de maintenance : les lancer comme des processus *one-off*

PENSER SON APPLICATION “CLOUD READY” 1/3

- Une base de code unique suivie dans un VCS (Git,...)
- Une configuration par environnement
- Architecture distribuée plutôt que monolithique
    - Facilite le passage à l’échelle
    - Limite les domaines de *failure*
- Couplage faible entre les composants

PENSER SON APPLICATION “CLOUD READY” 2/3

- Bus de messages pour les communications inter-composants
- Stateless : permet de multiplier les routes d’accès à l’application
- Dynamicité : l’application doit s’adapter à son environnement et se reconfigurer lorsque nécessaire
- Permettre le déploiement et l’exploitation par des outils d’automatisation

PENSER SON APPLICATION “CLOUD READY” 3/3

- Limiter autant que possible les dépendances à du matériel ou du logiciel spécifique qui pourrait ne pas fonctionner dans un cloud
- Tolérance aux pannes (*fault tolerance*) intégrée
- Ne pas stocker les données en local, mais plutôt :
  - Base de données
  - Stockage bloc
  - Stockage objet
- Utiliser des outils de journalisation standards

MODULARITÉ

- Philosophie Unix (Keep It Simple Stupid)
- Multiples composants de taille raisonnable
- Couplage faible et interface documentée

PASSAGE À L'ÉCHELLE

- Pets versus Cattle
- Vertical vs Horizontal
- Scale up/down vs Scale out/in
- Plusieurs petites instances plutôt qu'une seule grosse

STATEFUL VS STATELESS

- Beaucoup de stateful dans les applications legacy
- Nécessite de partager l'information d'état lorsque plusieurs workers
- Le stateless élimine cette contrainte

TOLÉRANCE AUX PANNES

- Les APIs du cloud sont hautement disponibles
- Le cloud ne garantit pas la haute disponibilté de l'application
- L’application prend en charge sa propre tolérance aux pannes

MODÈLES DE DÉPLOIEMENT

- Blue-Green *(attention aux quotas)*
- Rolling
- Canary

STOCKAGE DES DONNÉES

- Base de données relationnelle
- Base de données NoSQL
- Stockage bloc
- Stockage objet
- Stockage éphémère

GESTION DES LOGS

- Rester "applicatif"
- Enrichir les logs
- Ne pas présupposer le backend de traitement -> dans la conf

EXEMPLE EN PYTHON

appLog.conf:

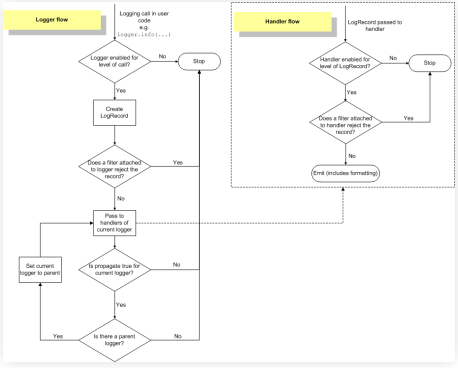
```
[logger_myapp]
qualname=mycompany.myapp
level=INFO
handlers=console
propagate=0
```

app.py:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import logging
log = logging.getLogger('mycompany.myapp.maintask')
log.info('Main worker started')
```

```
2018-12-24 22:20:02 INFO appuser Main worker started
```

LOGGING FLOW



MIGRATION DES APPLICATIONS LEGACY

- Rappel des enjeux
- Migrer ou non : critères de décision

CONCEVOIR UNE INFRA POUR LE CLOUD

L'INFRA AU SERVICE DE SON APPLICATION

- Souplesse
- Résilience
- Performance
- Opérabilité

UNE INFRA, ÇA ÉVOLUE !

- Dimensionnement des clusters
- Maintenance des O.S. « guest » et du middleware
- Règles SSI : segmentation réseau, filtrage de flux, proxys, bastions, annuaires
- Ajout de nouveaux services

MÉCANISER, AUTOMATISER, INDUSTRIALISER

- *Le niveau d'anxiété des comités face à la décision de déployer est inversement proportionnelle à la fréquence des MEP => cercle vicieux*
- (re)Construire, faire évoluer et maintenir les infras hébergées dans le cloud
- Reconstruction totale ou partielle à la demande
- Reproductibilité
- C'est Automagique !

AUTOMATISATION

- Mécaniser la gestion de l'infrastructure : indispensable
- Automatiser la gestion de l'infrastructure : un plus !
- Création des ressources
- Configuration des ressources

INFRASTRUCTURE AS CODE

- L'infra s'appréhende comme du code
- Travailler comme un développeur
- Décrire son infrastructure sous forme de code (Heat, Terraform)
- Suivre les changements dans un VCS (git), qui devient la référence
- Mettre en oeuvre un système d'intégration et de déploiement continus (CI/CD)

APPROCHE DE HEAT

- Un service <-> une API OpenStack
- Notions de stack et description YAML
- Précautions d'usage (stack update)
- Cas d'usage type

EXEMPLE HEAT

```
---
heat_template_version: 2018-08-31
description: A single nova instance
parameters:
  flavorName:
    type: string
resources:
  instance:
    type: OS::Nova::Server
    properties:
      name: MonInstance
      image: debianStretchOfTheDay
      flavor: {get_param: flavorName}
      key_name: MaCleSSH
outputs:
  instanceId:
    value: {get_resource: instance}
```

APPROCHE DE TERRAFORM

- L'aspect "cloud agnostique"
- Le DSL de Terraform
- L'exigence Infra as Code (terraform.tfstate)
- Cas d'usage type

EXEMPLE TERRAFORM

```
#  A single nova instance

# Configure the OpenStack Provider
provider "openstack" {
  user_name   = "MyName"
  tenant_name = "MyTenant"
  password    = "MyPwd"
  auth_url    = "http://myauthurl:5000/v2.0"
  region      = "RegionOne"
}

resource "openstack_compute_instance_v2" "MonInstance" {
  name            = "MonInstance"
  count           = "1"
  image_name      = ""
  flavor_name     = "${var.flavor}"
  key_pair        = "MaCleSSh"
}
```

AGILITÉ ET CI/CD APPLIQUÉES À L'INFRA

- Style de code
- Vérification de la syntaxe
- Tests unitaires
- Tests d'intégration
- Tests fonctionnels de bout en bout

TOLÉRANCE AUX PANNES

- Notion de résilience
- Load balancers
- Floating IPs
- Groupes de serveurs stateless
- Healthchecks

MISE À L'ÉCHELLE / ÉLASTICITÉ HORIZONTALE

- Groupe d’instances similaires, autoscaling group
- Nombre d’instances variable
- Scaling automatique en fonction de métriques

SUPERVISION

- Prendre en compte le cycle de vie des instances : DOWN != ALERT
- Monitorer les services plutôt que les serveurs
- Oublier les adresses IP ! Exposer un web service
- Prévoir un healthcheck fonctionnel (use case "métier")

BACKUP, PCA

- Infrastructure : être capable de reconstruire n'importe quel environnement à tout moment
- Données (de l'application, logs) : utiliser les modes de stockage bloc (volumes) et objet (dossiers)

GÉRER SES IMAGES

- Utilisation d'images génériques et personnalisation à l'instanciation (cloud-init)
- Création d'images offline :
  - *from scratch* : diskimage-builder (TripleO)
  - *from scratch* avec des outils spécifiques aux distributions (`openstack-debian-images` pour Debian)
  - modifiées avec libguestfs, virt-builder, virt-sysprep
- Création d'images via une instance :
  - automatisation possible avec Packer, Terraform, le CLI ou les API du IaaS
  - Golden images

ADAPTER LE MÉTIER ET L'ORGANISATION

IMPACTS SUR LES MÉTIERS

- l'Architecte
- l'Intégrateur
- l'Administrateur système
- l'Administrateur de base de données

RETOURS D'EXPÉRIENCES

- Convergence de métiers existants : architecte // intégrateur
    - doivent se comprendre et collaborer
- Apparition d'un nouveau métier : *développeur d'infra*
- Pilotage projet : doit être traité comme un développeur d'application en termes de budget, de compétences requises et d'organisation.

REPENSER L'ORGANISATION

- Adapter l'organisation sans tout chambouler
- Une dose d'agilité
- Amélioration continue et approche itérative

KUBERNETES : PROJET, GOUVERNANCE ET COMMUNAUTÉ

KUBERNETES

- COE développé par Google, devenu open source en 2014
- Adapté à tout type d'environnement
- Devenu populaire en très peu de temps
- Premier projet de la CNCF

CNCF

The Foundation's mission is to create and drive the adoption of a new computing paradigm that is optimized for modern distributed systems environments capable of scaling to tens of thousands of self healing multi-tenant nodes.

CNCF : PRÉREQUIS

- Distribuer sous forme de conteneurs
- Gestion dynamique de la configuration
- Orienté micro services

CNCF : LES RÔLES

- Intendance des projets
- Faire grossir et évoluer l'écosystème
- Rendre la technologie accessible
- Promouvoir la technologie

OCI

- Créé sous la Linux Fondation
- But : Créer un standard Open Source concernant la manière de "runner" et le format des conteneurs et images
- Non lié à des produits
- Non lié à des COE
- runC a été donné par Docker à l'OCI comme implémention de base

KUBERNETES : PROJET

- Docs : https://kubernetes.io/docs/
- Slack : http://slack.k8s.io/
- Discuss : https://discuss.kubernetes.io
- Stack Overflow : https://stackoverflow.com/questions/tagged/kubernete
- Serverfault https://stackoverflow.com/questions/tagged/kubernete

KUBERNETES : PROJET

- Hébergé sur Github : https://github.com/kubernetes/kubernetes :
  - Issues : https://github.com/kubernetes/kubernetes/issues
  - Pull Requests https://github.com/kubernetes/kubernetes/pulls
  - Releases : https://github.com/kubernetes/kubernetes/release
- Projets en incubation :
  - https://github.com/kubernetes-incubator/ (Deprécié)
  - https://github.com/kubernetes-sigs/

KUBERNETES : CYCLE DE DÉVELOPPEMENT

- Chaque *release* a son propre planning, pour exemple : https://github.com/kubernetes/sig-release/tree/master/releases/release-1.12#timeline
- Chaque cycle de développement dure 12 semaines et peut être étendu si nécessaire
- Features freeze
- Code Freeze
- Alpha Release
- Beta Releases
- Release Candidates

KUBERNETES : COMMUNAUTÉ

- Contributor and community guide : https://github.com/kubernetes/community/blob/maste community
- Décomposée en [Special Interest Groups] : https://github.com/kubernetes/community/blob/maste
- Les SIG sont des projets, centres d'intérêts ou Working
  - Network
  - Docs
  - AWS
  - etc
- Chaque SIG peut avoir des guidelines différentes.

KUBERNETES : KUBECON

La CNCF organise trois KubeCon par an :

- Amérique du Nord (San Diego, Seattle, Boston, etc)
- Europe (Berlin, Barcelone, Amsterdam, etc)
- Chine

KUBERNETES : ARCHITECTURE

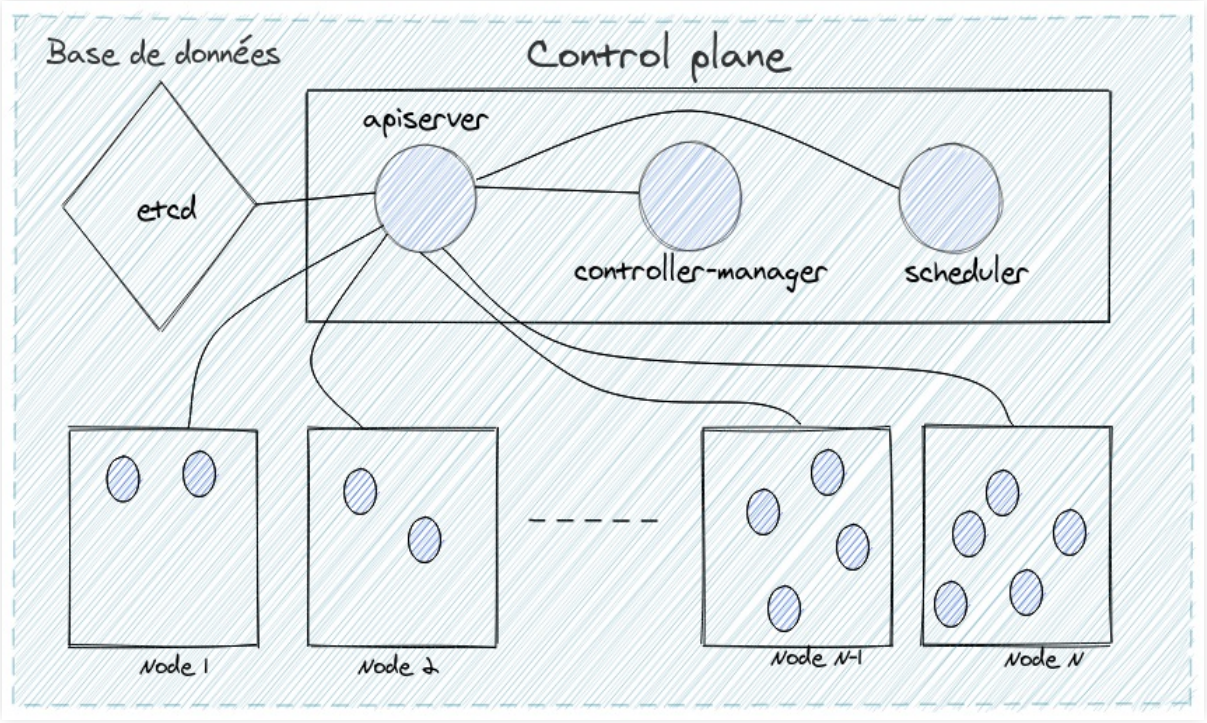
KUBERNETES : COMPOSANTS

- Kubernetes est écrit en Go, compilé statiquement.
- Un ensemble de binaires sans dépendance
- Faciles à conteneuriser et à packager
- Peut se déployer uniquement avec des conteneurs sans dépendance d'OS

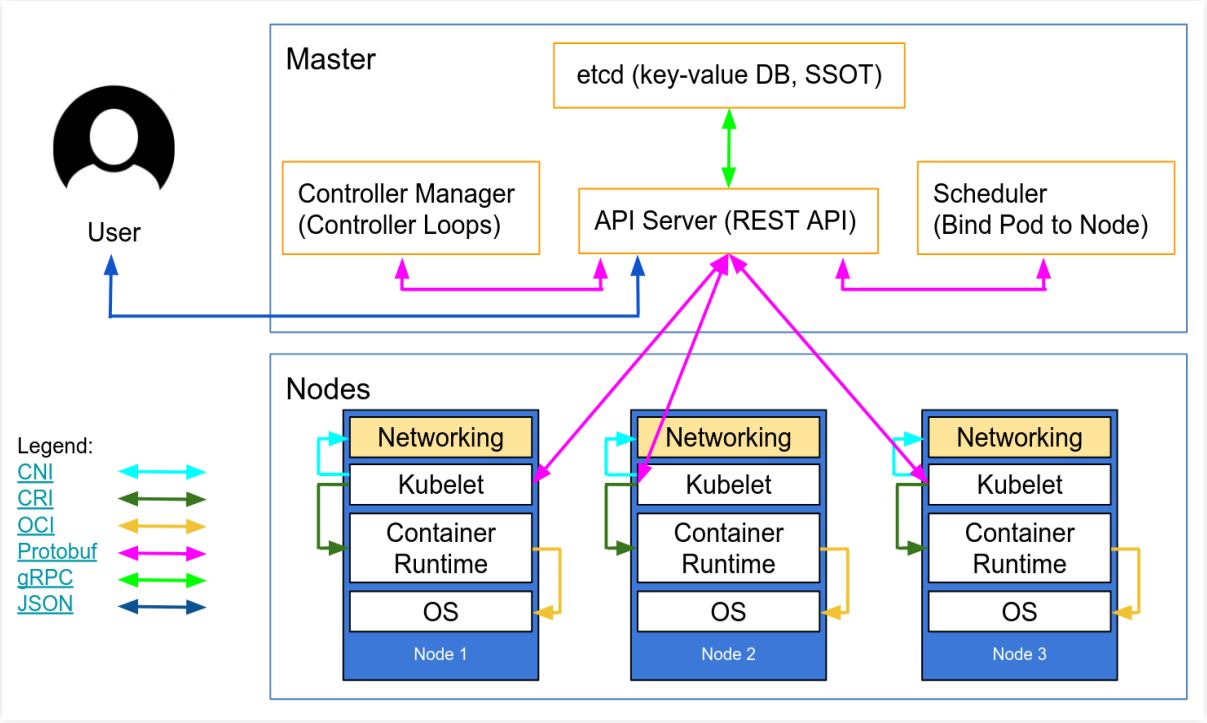
KUBERNETES : COMPOSANTS DU CONTROL PLANE

- etcd: Base de données
- kube-apiserver : API server qui permet la configuration d'objets Kubernetes (Pod, Service, Deployment, etc.)
- kube-proxy : Permet le forwarding TCP/UDP et le load balancing entre les services et les backends (Pods)
- kube-scheduler : Implémente les fonctionnalités de scheduling
- kube-controller-manager : Responsable de l'état du cluster, boucle infinie qui régule l'état du cluster afin d'atteindre un état désiré

KUBERNETES : COMPOSANTS DU CONTROL PLANE



KUBERNETES : COMPOSANTS DU CONTROL PLANE



KUBERNETES : ETCD

- Base de données de type Clé/Valeur (*Key Value Store*)
- Stocke l'état d'un cluster Kubernetes
- Point sensible (stateful) d'un cluster Kubernetes
- Projet intégré à la CNCF

KUBERNETES : KUBE-APISERVER

- Les configurations d'objets (Pods, Service, RC, etc.) se font via l'API server
- Un point d'accès à l'état du cluster aux autres composants via une API REST
- Tous les composants sont reliés à l'API server

KUBERNETES : KUBE-SCHEDULER

- Planifie les ressources sur le cluster
- En fonction de règles implicites (CPU, RAM, stockage disponible, etc.)
- En fonction de règles explicites (règles d'affinité et anti-affinité, labels, etc.)

KUBERNETES : KUBE-PROXY

- Responsable de la publication des Services
- Utilise *iptables*
- Route les paquets à destination des conteneurs et réalise le load balancing TCP/UDP

KUBERNETES : KUBE-CONTROLLER-MANAGER

- Boucle infinie qui contrôle l'état d'un cluster
- Effectue des opérations pour atteindre un état donné
- De base dans Kubernetes : replication controller, endpoints controller, namespace controller et serviceaccounts controller

KUBERNETES : AUTRES COMPOSANTS

- kubelet : Service "agent" fonctionnant sur tous les nœuds et assure le fonctionnement des autres services
- kubectl : Ligne de commande permettant de piloter un cluster Kubernetes

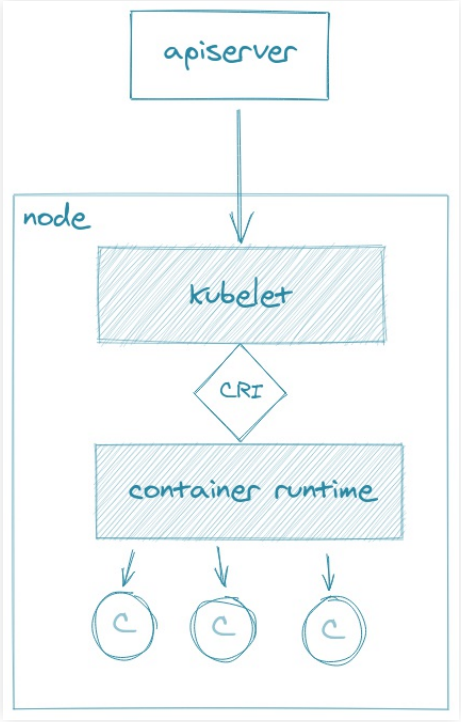
KUBERNETES : KUBELET

- Service principal de Kubernetes
- Permet à Kubernetes de s'auto configurer :
  - Surveille un dossier contenant les *manifests* (fichiers YAML des différents composant de Kubernetes).
  - Applique les modifications si besoin (upgrade, rollback).
- Surveille l'état des services du cluster via l'API server (*kube-apiserver*).

KUBERNETES : KUBELET

- Assure la communication entre les nodes et l'apiserver
- En charge de créer les conteneurs au travers de l'interface Container Runtime Interface (CRI)
- Peut fonctionner avec différentes container runtimes

KUBERNETES : KUBELET



KUBERNETES: NETWORK

Kubernetes n'implémente pas de solution réseau par défaut, mais s'appuie sur des solutions tierces qui implémentent les fonctionnalités suivantes :

- Chaque pods reçoit sa propre adresse IP
- Les pods peuvent communiquer directement sans NAT

KUBERNETES : CONCEPTS ET OBJETS

KUBERNETES : API RESOURCES

- Namespaces
- Pods
- Deployments
- DaemonSets
- StatefulSets
- Jobs
- Cronjobs

KUBERNETES : NAMESPACES

- Fournissent une séparation logique des ressources :
  - Par utilisateurs
  - Par projet / applications
  - Autres...
- Les objets existent uniquement au sein d'un namespace donné
- Évitent la collision de nom d'objets

KUBERNETES : LABELS

- Système de clé/valeur
- Organisent les différents objets de Kubernetes (Pods, RC, Services, etc.) d'une manière cohérente qui reflète la structure de l'application
- Corrèlent des éléments de Kubernetes : par exemple un service vers des Pods

KUBERNETES : LABELS

- Exemple de label :

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

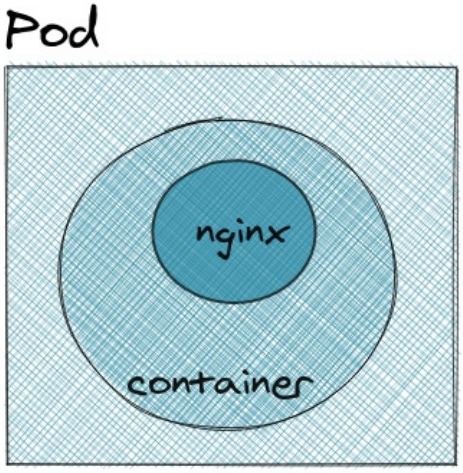
KUBERNETES : POD

- Ensemble logique composé de un ou plusieurs conteneurs
- Les conteneurs d'un pod fonctionnent ensemble (instanciation et destruction) et sont orchestrés sur un même hôte
- Les conteneurs partagent certaines spécifications du Pod :
    - La stack IP (network namespace)
    - Inter-process communication (PID namespace)
    - Volumes
- C'est la plus petite et la plus simple unité dans Kubernetes

KUBERNETES : POD

Les Pods sont définis en YAML comme les fichiers `docker-compose` :

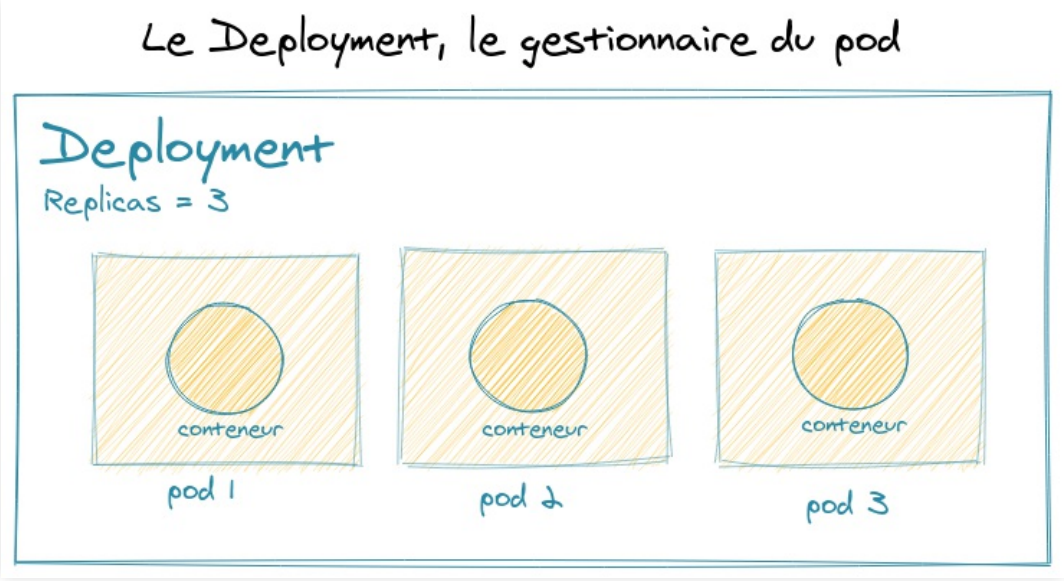
```yaml
---
apiVersion: v1
kind: Pod
metadata:
  name: mon_pod
spec:
  containers:
    - name: conteneur
      image: nginx:latest
```



KUBERNETES : DEPLOYMENT

- Permet d'assurer le fonctionnement d'un ensemble de Pods
- Version, Update et Rollback
- Anciennement appelés Replication Controllers

KUBERNETES : DEPLOYMENT



KUBERNETES : DEPLOYMENT

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
```

KUBERNETES : DAEMONSET

- Assure que tous les noeuds exécutent une copie du pod
- Ne connaît pas la notion de `replicas`.
- Utilisé pour des besoins particuliers comme :
  - l'exécution d'agents de collection de logs comme `fluentd` ou `logstash`
  - l'exécution de pilotes pour du matériel comme `nvidia-plugin`
  - l'exécution d'agents de supervision comme NewRelic agent ou Prometheus node exporter

KUBERNETES : DAEMONSET

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd
spec:
  selector:
    matchLabels:
      name: fluentd
  template:
    metadata:
      labels:
        name: fluentd
    spec:
      containers:
      - name: fluentd
        image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
```

KUBERNETES : STATEFULSET

- Similaire au `Deployment`
- Les pods possèdent des identifiants uniques.
- Chaque replica de pod est créé par ordre d'index
- Nécessite un `Persistent Volume` et un `Storage Class`.
- Supprimer un StatefulSet ne supprime pas le PV associé

KUBERNETES : STATEFULSET

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx
  serviceName: "nginx"
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: k8s.gcr.io/nginx-slim:0.8
        ports:
```

KUBERNETES : JOB

- Crée des pods et s'assurent qu'un certain nombre d'entre eux se terminent avec succès.
- Peut exécuter plusieurs pods en parallèle
- Si un noeud du cluster est en panne, les pods sont reschedulés vers un autre noeud.

KUBERNETES : JOB

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  parallelism: 1
  completions: 1
  template:
    metadata:
      name: pi
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: OnFailure
```

KUBERNETES: CRON JOB

- Un CronJob permet de lancer des Jobs de manière planifiée.
- la programmation des Jobs se définit au format `Cron`
- le champ `jobTemplate` contient la définition de l'application à lancer comme `Job`.

KUBERNETES : CRONJOB

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: batch-job-every-fifteen-minutes
spec:
  schedule: "0,15,30,45 * * * *"
  jobTemplate:
    spec:
      template:
        metadata:
          labels:
            app: periodic-batch-job
        spec:
          restartPolicy: OnFailure
          containers:
          -  name: pi
             image: perl
             command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
```

KUBERNETES : NETWORKING

KUBERNETES : NETWORK PLUGINS

- Kubernetes n'implémente pas de solution de gestion de réseau par défaut
- Le réseau est implémenté par des solutions tierces :
- Calico : IPinIP + BGP
- Cilium : eBPF
- Weave : VXLAN
- Bien d'autres

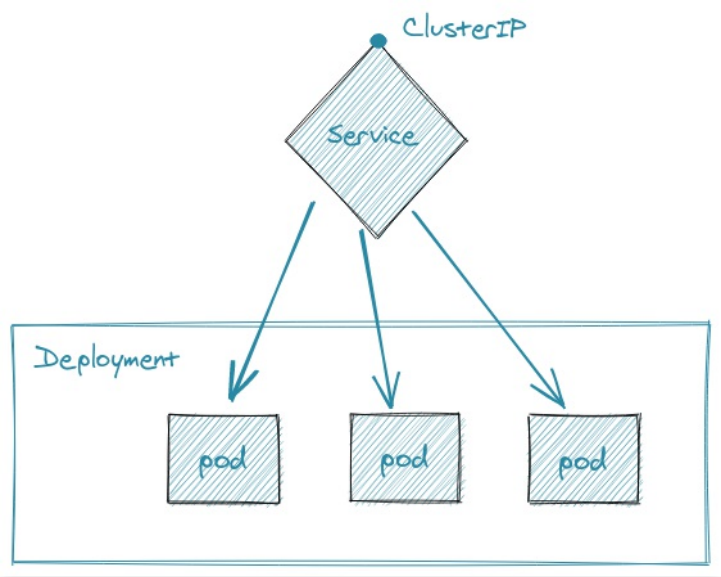
KUBERNETES : CNI

- Container Network Interface
- Projet dans la CNCF
- Standard pour la gestion du réseau en environnement conteneurisé
- Les solutions précédentes s'appuient sur CNI

KUBERNETES : SERVICES

- Abstraction des Pods sous forme d'une IP virtuelle de Service
- Rendre un ensemble de Pods accessibles depuis l'extérieur ou l'intérieur du cluster
- Load Balancing entre les Pods d'un même Service
- Sélection des Pods faisant parti d'un Service grâce aux labels

KUBERNETES : SERVICES



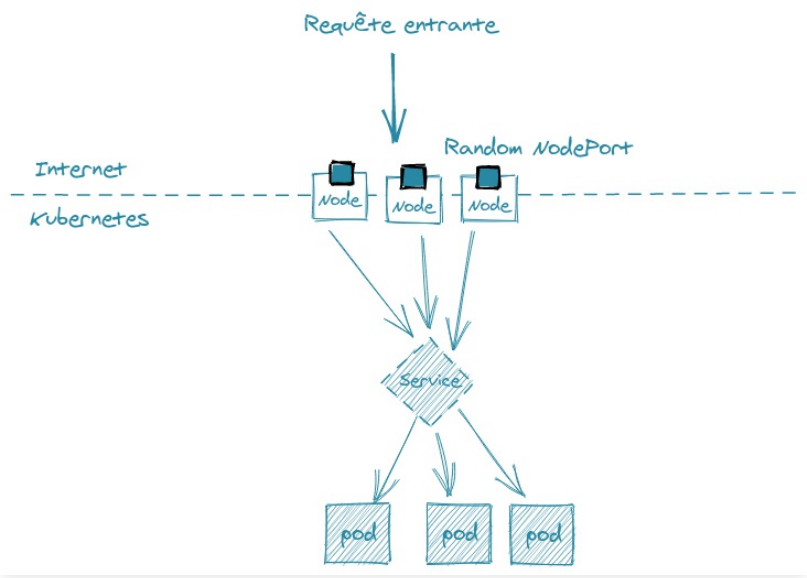
KUBERNETES : SERVICES CLUSTERIP

- Exemple de Service (on remarque la sélection sur le label et le mode d'exposition):

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
spec:
  type: ClusterIP
  ports:
  - port: 80
  selector:
    app: guestbook
```

KUBERNETES : SERVICE NODEPORT

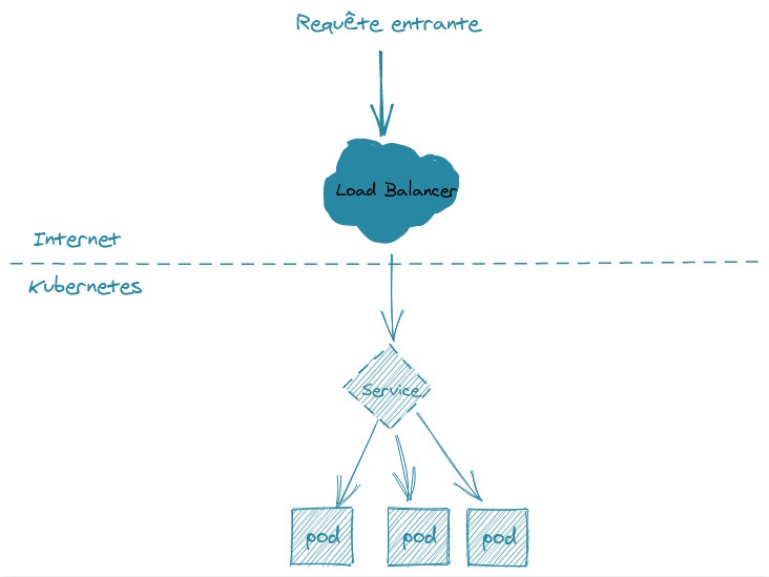
`NodePort` : chaque noeud du cluster ouvre un port statique et redirige le trafic vers le port indiqué



KUBERNETES : SERVICE LOADBALANCER

- `LoadBalancer` : expose le Service en externe en utilisant le loadbalancer d'un cloud provider
  - AWS ELB/ALB/NLB
  - GCP LoadBalancer
  - Azure Balancer
  - OpenStack Octavia

KUBERNETES : SERVICE LOADBALANCER



KUBERNETES : SERVICES

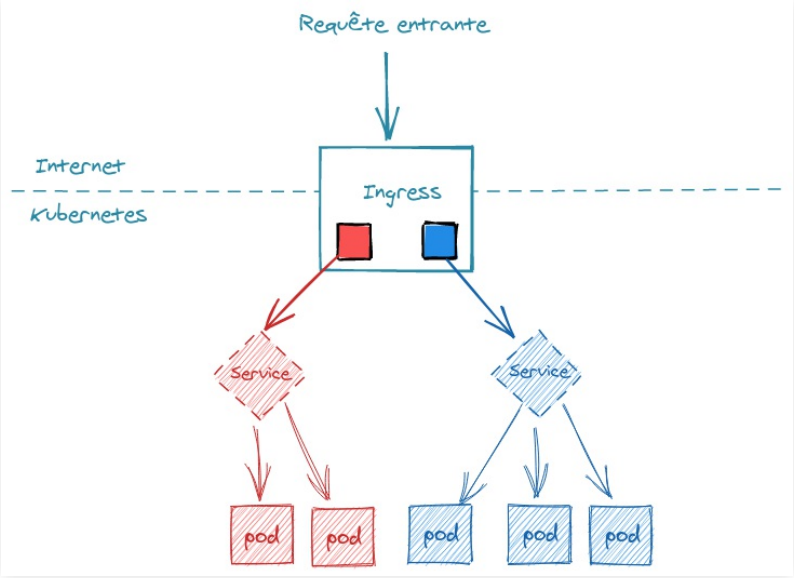
Il est aussi possible de mapper un Service avec un nom de domaine en spécifiant le paramètre `spec.externalName`.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
```

KUBERNETES: INGRESS

- L'objet `Ingress` permet d'exposer un Service à l'extérieur d'un cluster Kubernetes
- Il permet de fournir une URL visible permettant d'accéder un Service Kubernetes
- Il permet d'avoir des terminations TLS, de faire du *Load Balancing*, etc...

KUBERNETES: INGRESS



KUBERNETES : INGRESS

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: particule
spec:
  rules:
  - host: blog.particule.io
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: frontend
            port:
              number: 80
```

KUBERNETES : INGRESS CONTROLLER

Pour utiliser un `Ingress`, il faut un Ingress Controller. Un `Ingress` permet de configurer une règle de reverse proxy sur l'Ingress Controller.

- Nginx Controller : https://github.com/kubernetes/ingress-nginx
- Traefik : https://github.com/containous/traefik
- Istio : https://github.com/istio/istio
- Linkerd : https://github.com/linkerd/linkerd
- Contour : https://www.github.com/heptio/contour/

KUBERNETES : STOCKAGE

KUBERNETES : VOLUMES

- Fournir du stockage persistent aux pods
- Fonctionnent de la même façon que les volumes Docker pour les volumes hôte :
  - EmptyDir ~= volumes docker
  - HostPath ~= volumes hôte
- Support de multiples backend de stockage :
  - GCE : PD
  - AWS : EBS
  - GlusterFS / NFS
  - Ceph
  - iSCSI

KUBERNETES : VOLUMES

- On déclare d'abord le volume et on l'affecte à un pod :

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: redis
    volumeMounts:
    - name: redis-persistent-storage
      mountPath: /data/redis
  volumes:
  - name: redis-persistent-storage
    emptyDir: {}
```

KUBERNETES : STORAGE CLASS

- Permet de définir les différents types de stockage disponibles
- Utilisé par les `Persistent Volumes` pour solliciter un espace de stockage au travers des `Persistent Volume Claims`

KUBERNETES : STORAGE CLASS

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: slow
provisioner: kubernetes.io/aws-ebs
parameters:
  type: io1
  zones: us-east-1d, us-east-1c
  iopsPerGB: "10"
```

KUBERNETES : PERSISTENTVOLUMECLAIMS

- Ressource utilisée et vue comme une requête pour solliciter du stockage persistant
- Offre aux PV une variété d'options en fonction du cas d'utilisation
- Utilisé par les `StatefulSets` pour solliciter du stockage (Utilisation du champ `volumeClaimTemplates`)

KUBERNETES : PERSISTENTVOLUMECLAIMS

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: storage-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
  storageClassName: "slow"
```

KUBERNETES : PERSISTENTVOLUME

- Composant de stockage dans le cluster kubernetes
- Stockage externe aux noeuds du cluster
- Cycle de vie indépendant du pod qui le consomme
- Peut être provisionné manuellement par un administrateur ou dynamiquement grâce à une `StorageClass`

KUBERNETES : PERSISTENTVOLUME

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: persistent-volume-1
spec:
  storageClassName: slow
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/tmp/data"
```

KUBERNETES : CSI

- Container Storage Interface
- Équivalent de CNI mais pour les volumes
- Avant Kubernetes 1.13, tous les drivers de volumes étaient *in tree*
- Le but de la séparation est de sortir du code du *core* de Kubernetes
- GA depuis Kubernetes 1.13

KUBERNETES : CSI

- La plupart des volumes supportés dans Kubernetes supportent maintenant CSI :
- Amazon EBS
- Google PD
- Cinder
- GlusterFS
- La liste exhaustive est disponible ici

KUBERNETES : GESTION DE LA CONFIGURATION DES APPLICATIONS

KUBERNETES : CONFIGMAPS

- Objet Kubernetes permettant de stocker séparément les fichiers de configuration
- Il peut être créé d'un ensemble de valeurs ou d'un fichier resource Kubernetes (YAML ou JSON)
- Un `ConfigMap` peut sollicité par plusieurs `pods`

KUBERNETES : CONFIGMAP ENVIRONNEMENT (1/2)

```
apiVersion: v1
data:
  username: admin
  url: https://api.particule.io
kind: ConfigMap
metadata:
  name: web-config
```

KUBERNETES : CONFIGMAP ENVIRONNEMENT (2/2)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: configmap-env
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: USERNAME
          valueFrom:
            configMapKeyRef:
              name: web-config
              key: username
        - name: URL
          valueFrom:
            configMapKeyRef:
              name: web-config
```

KUBERNETES : CONFIGMAP VOLUME (1/2)

```
apiVersion: v1
data:
  redis-config: |
    maxmemory 2mb
    maxmemory-policy allkeys-lru
kind: ConfigMap
metadata:
  name: redis-config
```

KUBERNETES : CONFIGMAP VOLUME (2/2)

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-volume
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "head -v /etc/config/*" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: redis-config
  restartPolicy: Never
```

KUBERNETES : SECRETS

- Objet Kubernetes de type `secret` utilisé pour stocker des informations sensibles comme les mots de passe, les *tokens*, les clés SSH...
- Similaire à un `ConfigMap`, à la seule différence que le contenu des entrées présentes dans le champ `data` sont encodés en base64.
- Il est possible de directement créer un `Secret` spécifique à l'authentification sur une registry Docker privée.
- Il est possible de directement créer un `Secret` à partir d'un compte utilisateur et d'un mot de passe.

KUBERNETS : SECRETS

- S'utilisent de la même façon que les `ConfigMap`
- La seule différence est le stockage en base64
- 3 types de secrets:
- `Generic`: valeurs arbitraire comme dans une `ConfigMap`
- `tls`: certificat et clé pour utilisation avec un serveur web
- `docker-registry`: utilisé en tant que `imagePullSecret` par un pod pour pouvoir pull les images d'une registry privée

KUBERNETES : SECRETS

```
kubectl create secret generic monSuperSecret --from-literal=username='monUser' -
```

KUBERNETES : SECRETS

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

Les valeurs doivent être encodées en base64.

KUBERNETES: OÙ DEPLOYER ET COMMENT ?

BARE METAL, PRIVATE ET PUBLIC CLOUDS

Managed - Kops - Kubespray - Kubeadm - Kube-aws - Symplegma

▼

Cluster Deployment

Cluster Lifecycle

INFRASTUCTURE AS CODE

IaC : Infrastructure as Code

▼

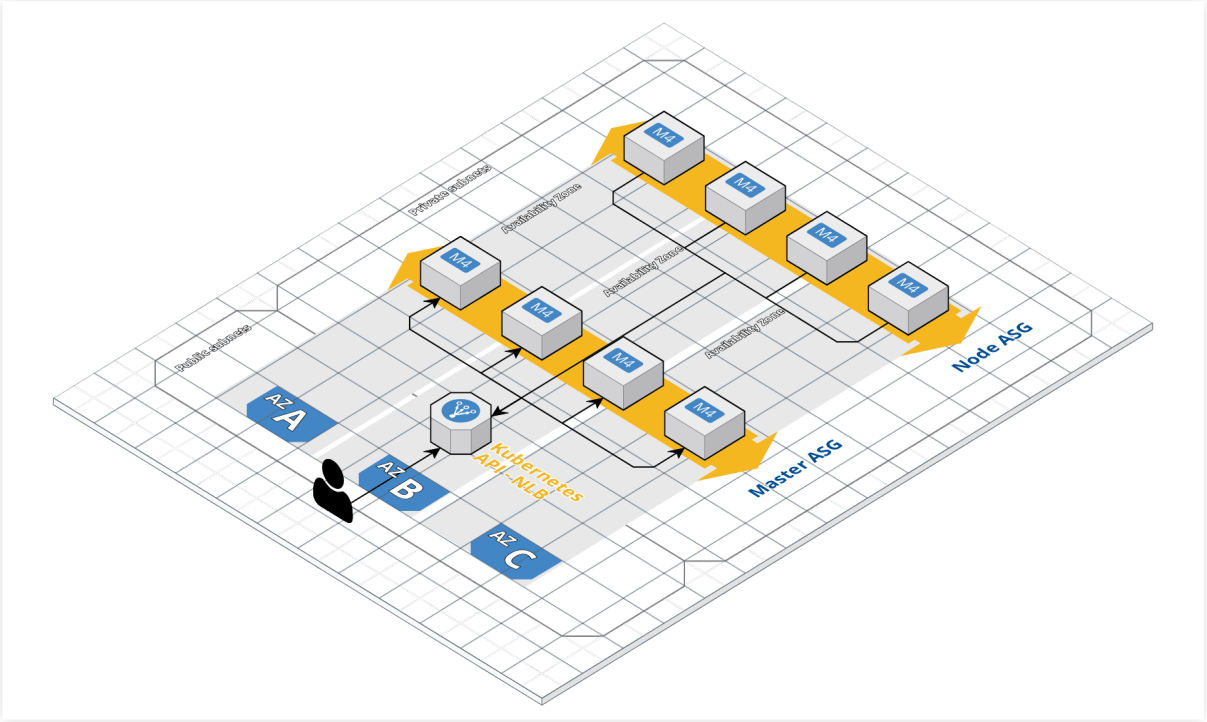
Terraform - CloudFormation - Cloud Deployment Manager - OpenStack Heat - Azure Resource Manager

▼

Infrastructure déclarée

Infrastructure immuable

IMPLÉMENTATION DE RÉFÉRENCE



QUE CHOISIR ?

Je veux utiliser Kubernetes

Cloud ?

Cloud public ou privé ?

Configuration particulière ?

Multiple cloud providers ?

Homogénéité des outils ?

LOCAL KUBERNETES

- Minikube: Machine virtuelle locale
- Kind: Kubernetes in Docker
- k3s: Kubernetes léger
- Docker for Mac/Windows

KUBERNETES MANAGÉ

AWS EKS

- Control plane managé par AWS
- Amazon Linux / Ubuntu
- CloudFormation / Terraform / eksctl

GKE

- Control plane managé par GCP
- Premier sur le marché
- COS / Ubuntu
- Terraform / Google Cloud SDK

AZURE AKS

- Control plane managé par Azure

OPENSTACK MAGNUM

- Control plane managé par OpenStack
- Basé sur OpenStack Heat
- Proposé sur certains public cloud basés sur OpenStack

OUTILS DE DÉPLOIEMENTS AGNOSTIQUES

KUBEADM

- Outil **officiel** de la communauté
- Stable depuis v1.13.0
- Ne provisionne pas de machine
- Facilement personnalisabe
- Respect des best practices
- Peut être utilisé par d'autres outils

KUBESPRAY

- Basé sur Ansible
- Dense, permet d'installer un nombre important de plugins
- Multiples OS
- Support Kubeadm

SYMPLEGMA

- Basé sur Ansible
- Inspiré de Kubespray en plus léger
- CoreOS/Ubuntu
- Full Kubeadm

OUTILS DE DÉPLOIEMENTS SPÉCIFIQUES

KUBE-AWS

- Pure AWS CloudFormation Stack
- Cycle de release lent
- Facilement personnalisable
- CoreOS ❤

KOPS

- Déploie sur AWS/GCP/OpenStack et Digital Ocean
- Cycle de release lent
- Facilement personnalisable
- Multiples OS
- Supporte Cloudformation and Terraform

KUBERNETES : HELM

QU'EST-CE QUE HELM ?

- Outil de packaging d'application Kubernetes
- Developpé en Go
- Actuellement en v3
- Projet *graduated* de la CNCF

https://github.com/helm/helm

POURQUOI HELM ?

- Applique le principe DRY (Don't Repeat Yourself)
  - Mécanisme de templating (Go templating)
  - Variabilisation des ressources générées
- Facilité de versionnement et de partage (repository Helm)
- Helm permet d'administrer les Releases
  - Rollbacks / upgrades d'applications

CONCEPTS

Concept	Description
Chart	Ensemble de ressources permettant de definir une application Kubernetes
Config	Valeurs permettant de configurer un Chart (`values.yaml`)
Release	Chart deployé avec une Config

COMPARAISON AVEC DES MANIFESTS YAML

- Permet de mettre en place le DRY (Don't Repeat Yourself)
    - customisation via fichier de configuration YAML
- Définition d'une seule source de vérité
    - Les ressources sont packagées
- Packages déployés via des Releases

STRUCTURE D'UN CHART

- Chart.yaml pour définir le chart ainsi que ses metadatas
- values.yaml sert à definir les valeurs de configuration du Chart par défaut
- `crds/`: Dossier qui recense les CRDs
- `templates/`: les templates de manifeste Kubernetes en YAML

CHART.YAML

Le fichier de configuration du Chart dans lequel sont définies ses metadatas.

```
---
apiVersion: v2
description: Hello World Chart.
name: hello-world-example
sources:
  - https://github.com/prometheus-community/helm-charts
version: 1.3.2
appVersion: 0.50.3
dependencies: []
```

STRUCTURE DU VALUES.YAML

- Chaque attribut est ensuite disponible au niveau des templates

```
---
### Provide a name in place of kube-prometheus-stack for `app:` labels
##
applicationName: ""

### Override the deployment namespace
##
namespaceOverride: ""


### Apply labels to the resources
##
commonLabels: {}
```

SURCHARGE DU VALUES.YAML

```yaml
---
# values-production.yaml
commonLabels:
  env: prod
```

```
tree
.
├── Chart.yaml
├── templates
│   ├── application.yaml
│   ├── configuration.yaml
│   └── secrets.yaml
├── values-production.yaml
├── values-staging.yaml
└── values.yaml
```

TEMPLATES

Helm permet de variabiliser les manifestes Kubernetes, permettant de créer et configurer des ressources dynamiquement. Le langage Go Template est utilisé.

```
apiVersion: apps/v1
kind: Pod
metadata:
  name: {{ .Chart.Name }}
  labels:
    app.kubernetes.io/managed-by: "Helm"
    chart: "{{ .Chart.Name }}-{{ .Chart.Version }}"
    release: {{ .Release.Name | quote }}
    version: 1.0.0
spec:
  containers:
  - image: "{{ .Values.helloworld.image.name }}:{{ .Values.helloworld.image.tag
    name: helloworld
```

GESTION DES REPOSITORIES

- Un Repository Helm permet de distribuer et versionner des Charts
- Contient un `index.yaml` listant les Charts packagés disponibles par version
- Deux méthodes de déploiement possibles
  - Via HTTP en tant que fichiers statiques
  - Via OCI en utilisant une Registry (depuis Helm v3)

COMMANDES COMMUNES

```
$ helm repo add stable https://charts.helm.sh/stable
"stable" has been added to your repositories
$ helm repo update
$ helm install stable/airflow --generate-name
helm install stable/airflow --generate-name
NAME: airflow-1616524477
NAMESPACE: defaul
...
$ helm upgrade airflow-1616524477 stable/airflow
helm upgrade airflow-1616524477 stable/airflow
Release "airflow-1616524477" has been upgraded. Happy Helming!
$ helm rollback airflow-1616524477
Rollback was a success! Happy Helming!
$ helm uninstall airflow-1616524477
release "airflow-1616524477" uninstalled
```

KUBERNETES : SECURITÉ ET CONTROLE D'ACCÈS

AUTHENTICATION & AUTORISATION

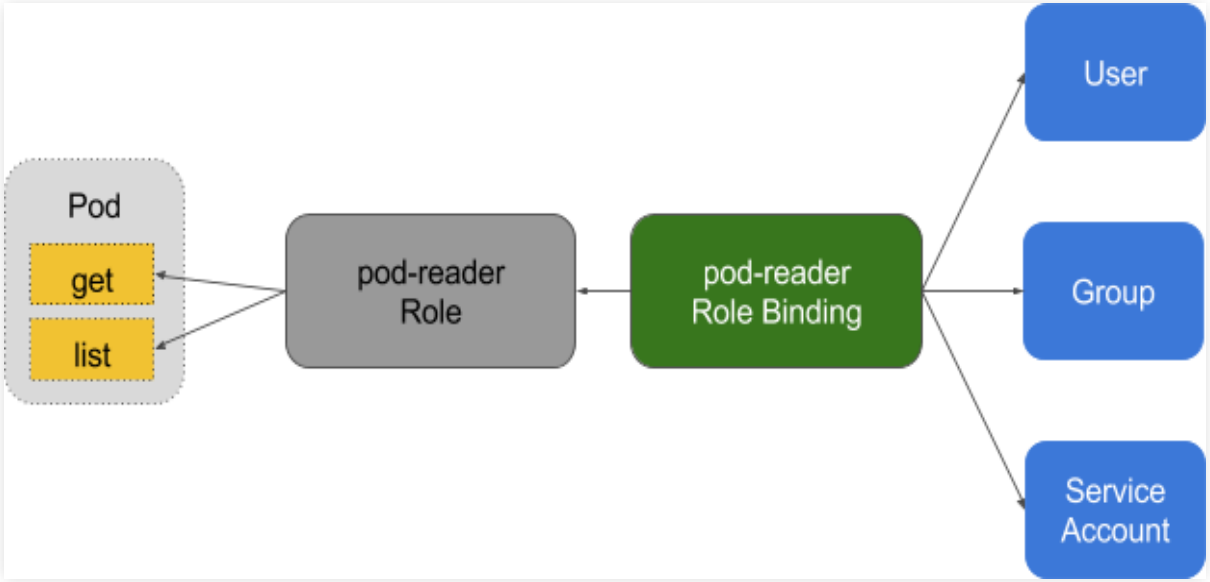
- RBAC (Role Based Access Control)
- ABAC (Attribute-based access control)
- WebHook
- Certificates
- Token

RBAC

3 entités sont utilisées :

- Utilisateurs représentés par les `Users` ou les `ServiceAccounts`
- Ressources représentées par les `Deployments`, `Pods`, `Services`, etc...
- les différentes opérations possibles : `create, list, get, delete, watch, patch`

RBAC


Pod
get
list
pod-reader
Role
pod-reader
Role Binding
User
Group
Service
Account

SERVICE ACCOUNTS

- Objet Kubernetes permettant d'identifier une application s'éxecutant dans un pod
- Par défaut, un `ServiceAccount` par `namespace`
- Le `ServiceAccount` est formatté ainsi :

```
system:serviceaccount:<namespace>:<service_account_name>
```

SERVICE ACCOUNTS

```
apiVersion: v1
kind: ServiceAccount
metadata:
    name: default
    namespace: default
```

ROLE

- L'objet `Role` est un ensemble de règles permettant de définir quelle opération (ou *verbe*) peut être effectuée et sur quelle ressource
- Le `Role` ne s'applique qu'à un seul `namespace` et les ressources liées à ce `namespace`

ROLE

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

ROLEBINDING

- L'objet `RoleBinding` va allouer à un `User`, `ServiceAccount` ou un groupe les permissions dans l'objet `Role` associé
- Un objet `RoleBinding` doit référencer un `Role` dans le même `namespace`.
- L'objet `roleRef` spécifié dans le `RoleBinding` est celui qui crée le liaison

ROLEBINDING

```yaml
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: jane
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

CLUSTERROLE

- L'objet `ClusterRole` est similaire au `Role` à la différence qu'il n'est pas limité à un seul `namespace`
- Il permet d'accéder à des ressources non limitées à un `namespace` comme les `nodes`

CLUSTERROLE

```yaml
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: secret-reader
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

CLUSTERROLEBINDING

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: salme-reads-all-pods
subjects:
- kind: User
  name: jsalmeron
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: secret-reader
  apiGroup: rbac.authorization.k8s.io
```

RBAC

```
kubectl auth can-i get pods /
--namespace=default /
--as=spesnova@example.com
```

NETWORKPOLICIES

- La ressource `NetworkPolicy` est une spécification permettant de définir comment un ensemble de `pods` communiquent entre eux ou avec d'autres endpoints
- Le `NetworkPolicy` utilisent les labels pour sélectionner les pods sur lesquels s'appliquent les règles qui définissent le trafic alloué sur les pods sélectionnés
- Le `NetworkPolicy` est générique et fait partie de l'API Kubernetes. Il est nécessaire que le plugin réseau déployé supporte cette spécification

NETWORKPOLICIES

- DENY tout le trafic sur une application
- LIMIT le trafic sur une application
- DENY le trafic all non alloué dans un namespace
- DENY tout le trafic venant d'autres namespaces
- exemples de Network Policies : https://github.com/ahmetb/kubernetes-network-policy-recipes

NETWORKPOLICIES

- Exemple de `NetworkPolicy` permettant de blocker le trafic entrant :

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: web-deny-all
spec:
  podSelector:
    matchLabels:
      app: web
  ingress: []
```

PODSECURITYPOLICIES

- Permet de contrôler les privilèges d'un pod
- Permet de définir ce qui est autorisé pendant l'exécution du pod
- A utiliser dans un contexte multi-tenant et quand les pods ne viennent pas d'un tiers de confiance
- Peut-être combiné avec le RBAC
- Attention: Activer cette fonctionnalité peut endommager votre environnement
- Il faut une PSP par défaut

PODSECURITYPOLICIES

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restricted
spec:
  privileged: false
  allowPrivilegeEscalation: false
  requiredDropCapabilities:
    - ALL
  hostNetwork: false
  hostIPC: false
  hostPID: false
  runAsUser:
    rule: 'MustRunAsNonRoot'
  readOnlyRootFilesystem: false
```

ADMISSION CONTROLLERS

- Interceptent les requêtes sur l'API Kubernetes
- Peut effectuer des modifications si nécessaires
- Conception personnalisée possible

ADMISSION CONTROLLERS

- DenyEscalatingExec
- ImagePolicyWebhook
- NodeRestriction
- PodSecurityPolicy
- SecurityContextDeny
- ServiceAccount

CONCLUSION