

# Exporter tout le contenu

205-265 minutes

---

Supports de formation : Elie Gavoty, Alexandre Aubin et Hadrien Pélissier  
Sous [licence CC-BY-NC-SA](#) - Formations Uptime

## Docker

## Introduction DevOps

### A propos de moi

### A propos de vous

- Attentes ?
- Début du cursus :
- Est-ce que ça vous plait ?
- Quels modules avez vous déjà fait ?

## Le mouvement DevOps

Le DevOps est avant tout le nom d'un mouvement de transformation professionnelle et technique de l'informatique.

Ce mouvement se structure autour des solutions **humaines** (organisation de l'entreprise et des équipes) et **techniques** (nouvelles technologies de rupture) apportées pour répondre aux défis que sont:

- L'agrandissement rapide face à la demande des services logiciels et infrastructures les supportant.
- La célérité de déploiement demandée par le développement agile (cycles journaliers de développement).
- Difficultés à organiser des équipes hétérogènes de grande taille et qui s'agrandissent très vite selon le modèle des startups.

Il y a de nombreuses versions de ce que qui caractérise le DevOps mais pour résumer:

Du côté humain:

- Application des processus de management agile aux opérations et la gestion des infrastructures (pour les synchroniser avec le développement).
- Remplacement des procédés d'opérations

humaines complexes et spécifiques par des opérations automatiques et mieux standardisées.

- Réconciliation de deux cultures divergentes (Dev et Ops) rapprochant en pratique les deux métiers du développeur et de l'administrateur système.

Du côté technique:

- L'intégration et le déploiement continus des logiciels/produits.
- L'infrastructure as code: gestion sous forme de code de l'état des infrastructures d'une façon le plus possible déclarative.
- Les conteneurs (Docker surtout mais aussi Rkt et LXC/LXD): plus léger que la virtualisation = permet d'isoler chaque service dans son “OS” virtuel sans dupliquer le noyau.
- Le cloud (Infra as a service, Plateforme as a Service, Software as a service) permet de fluidifier l'informatique en alignant chaque niveau d'abstraction d'une pile logicielle avec sa structuration économique sous forme de service.

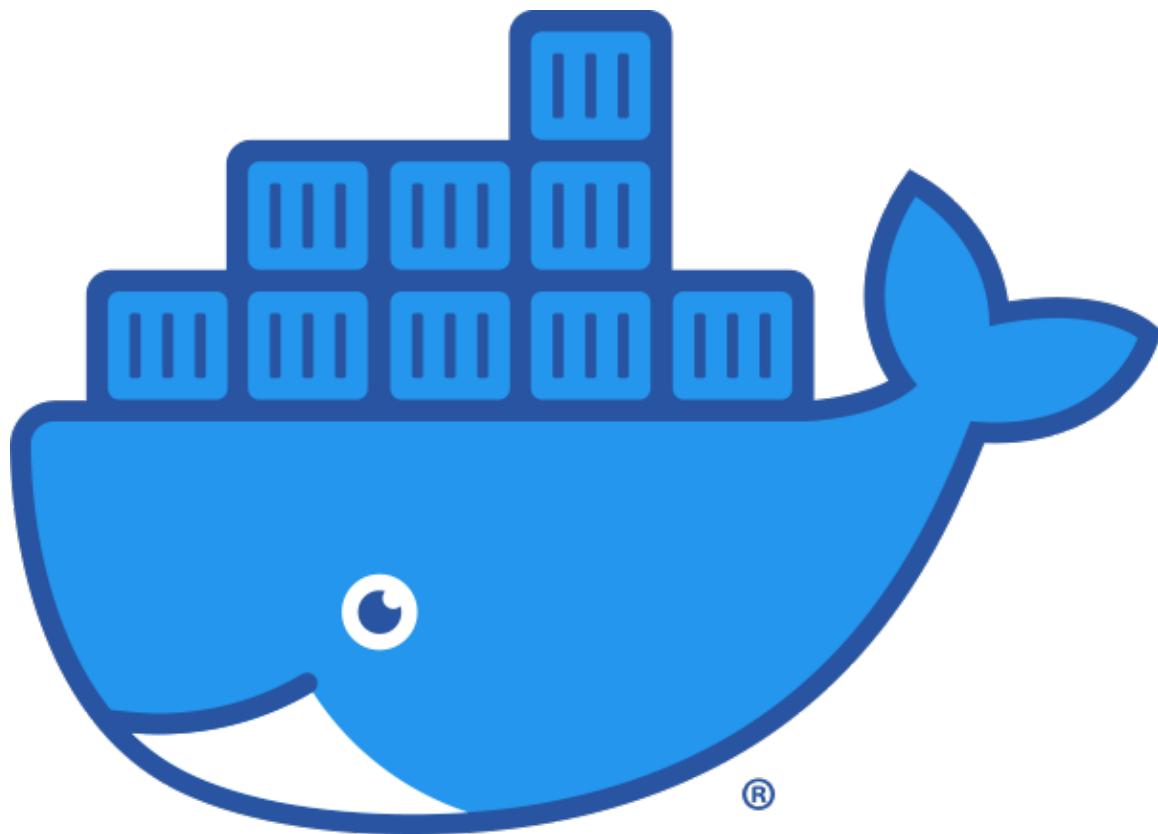
## Aller plus loin

- La DevOps roadmap: <https://github.com>

[/kamranahmedse/developer-roadmap#devops-roadmap](https://kamranahmedse/developer-roadmap#devops-roadmap)

## 0 - Introduction à Docker

***Modularisez et maîtrisez vos applications***



---

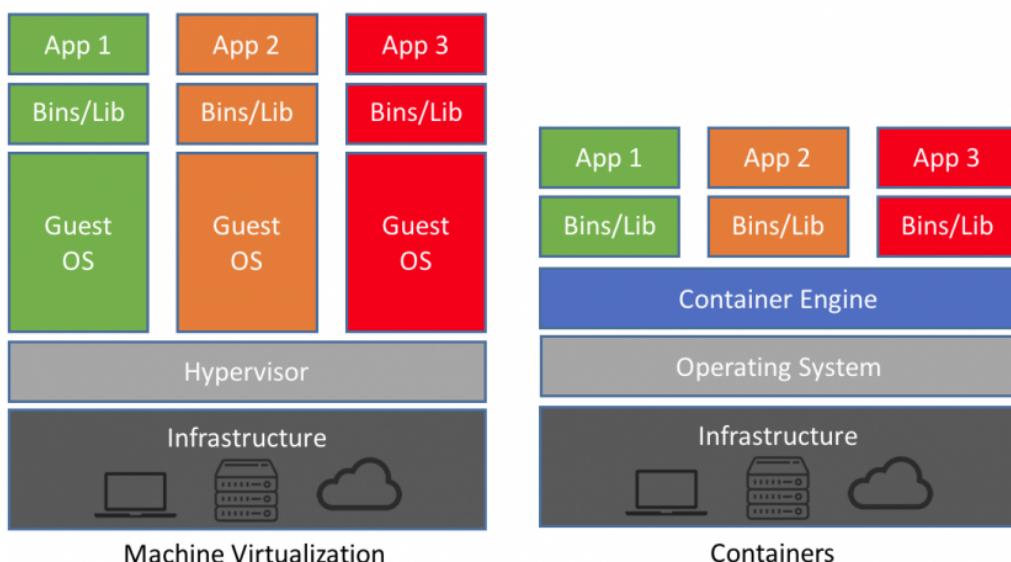
### Introduction

- **La métaphore docker : “box it, ship it”**
- Une abstraction qui ouvre de nouvelles possibilités pour la manipulation logicielle.
- Permet de standardiser et de contrôler la livraison

et le déploiement.

## Retour sur les technologies de virtualisation

On compare souvent les conteneurs aux machines virtuelles. Mais ce sont de grosses simplifications parce qu'on en a un usage similaire : isoler des programmes dans des “contextes”. Une chose essentielle à retenir sur la différence technique : **les conteneurs utilisent les mécanismes internes du \_kernel de l’OS Linux\_ tandis que les VM tentent de communiquer avec l’OS (quel qu’il soit) pour directement avoir accès au matériel de l’ordinateur.**



- **VM** : une abstraction complète pour simuler des machines

- un processeur, mémoire, appels systèmes, carte réseau, carte graphique, etc.
- **conteneur** : un découpage dans Linux pour séparer des ressources (accès à des dossiers spécifiques sur le disque, accès réseau).

Les deux technologies peuvent utiliser un système de quotas pour l'accès aux ressources matérielles (accès en lecture/écriture sur le disque, sollicitation de la carte réseau, du processeur)

Si l'on cherche la définition d'un conteneur :

**C'est un groupe de *processus* associé à un ensemble de permissions.**

L'imaginer comme une “boîte” est donc une allégorie un peu trompeuse, car ce n'est pas de la virtualisation (= isolation au niveau matériel).

---

## **Docker Origins : genèse du concept de conteneur**

Les conteneurs mettent en œuvre un vieux concept d'isolation des processus permis par la philosophie Unix du “tout est fichier”.

**chroot, jail, les 6 namespaces et les**

# cgroups

## chroot

- Implémenté principalement par le programme `chroot` [*change root* : changer de racine], présent dans les systèmes UNIX depuis longtemps (1979 !)

:

“Comme tout est fichier, changer la racine d'un processus, c'est comme le faire changer de système”.

## jail

- `jail` est introduit par FreeBSD en 2002 pour compléter `chroot` et qui permet pour la première fois une **isolation réelle (et sécurisée) des processus**.
- `chroot` ne s'occupait que de l'isolation d'un process par rapport au système de fichiers :
- ce n'était pas suffisant, l'idée de "tout-est-fichier" possède en réalité plusieurs exceptions
- un process *chrooté* n'est pas isolé du reste des process et peut agir de façon non contrôlée sur le système sur plusieurs aspects

- En 2005, Sun introduit les **conteneurs Solaris** décrits comme un « chroot sous stéroïdes » : comme les *jails* de FreeBSD

## Les *namespaces* (espaces de noms)

- Les ***namespaces***, un concept informatique pour parler simplement de...
- groupes séparés auxquels on donne un nom, d'ensembles de choses sur lesquelles on colle une étiquette
- on parle aussi de **contextes**
- jail était une façon de *compléter chroot*, pour FreeBSD.
- Pour Linux, ce concept est repris via la mise en place de **namespaces Linux**
- Les *namespaces* sont inventés en 2002
- popularisés lors de l'inclusion des 6 types de *namespaces* dans le **noyau Linux** (3.8) en **2013**
- Les conteneurs ne sont finalement que **plein de fonctionnalités Linux saucissonnées ensemble de façon cohérente**.
- Les *namespaces* correspondent à autant de types

de **compartiments** nécessaires dans l'architecture Linux pour isoler des processus.

Pour la culture, 6 types de *namespaces* :

- **Les namespaces PID** : “fournit l’isolation pour l’allocation des identifiants de processus (PIDs), la liste des processus et de leurs détails. Tandis que le nouvel espace de nom est isolé de ses adjacents, les processus dans son espace de nommage « parent » voient toujours tous les processus dans les espaces de nommage enfants — quoique avec des numéros de PID différent.”
- **Network namespace** : “isole le contrôleur de l’interface réseau (physique ou virtuel), les règles de pare-feu iptables, les tables de routage, etc.”
- **Mount namespace** : “permet de créer différents modèles de systèmes de fichiers, ou de créer certains points de montage en lecture-seule”
- **User namespace** : isolates the user IDs between namespaces (dernière pièce du puzzle)
- “UTS” namespace : permet de changer le nom d’hôte.
- IPC namespace : isole la communication inter-processus entre les espaces de nommage.

---

## Les **cgroups** : derniers détails pour une vraie isolation

- Après, il reste à s'occuper de limiter la capacité d'un conteneur à agir sur les ressources matérielles :
    - usage de la mémoire
    - du disque
    - du réseau
    - des appels système
    - du processeur (CPU)
  - En 2005, Google commence le développement des **cgroups** : une façon de *tagger* les demandes de processeur et les appels systèmes pour les grouper et les isoler.
- 

### Exemple : bloquer le système hôte depuis un simple conteneur

```
[ : () { : | : & } ; : ]
```

Ceci est une *fork bomb*. Dans un conteneur **non privilégié**, on bloque tout Docker, voire tout le système sous-jacent, en l'empêchant de créer de

nouveaux processus.

Pour éviter cela il faudrait limiter la création de processus via une option kernel.

Ex: docker run -it --ulimit nproc=3  
--name fork-bomb bash

**L'isolation des conteneurs n'est donc ni magique, ni automatique, ni absolue !**

Correctement paramétrée, elle est tout de même assez **robuste, mature et testée**.

---

## Les conteneurs : définition

On revient à notre définition d'un **conteneur** :

**Un conteneur est un groupe de *processus* associé à un ensemble de permissions sur le système.**

- 1 container = 1 groupe de *process* Linux
- des *namespaces* (séparation entre ces groups)
  - des *cgroups* (quota en ressources matérielles)
- 

## LXC (LinuX Containers)

- En 2008 démarre le projet LXC qui chercher à

rassembler :

- les **cgroups**
  - le **chroot**
  - les **namespaces**.
  - Originellement, Docker était basé sur **LXC**. Il a depuis développé son propre assemblage de ces 3 mécanismes.
- 

## Docker et LXC

- En 2013, Docker commence à proposer une meilleure finition et une interface simple qui facilite l'utilisation des conteneurs **LXC**.
  - Puis il propose aussi son cloud, le **Docker Hub** pour faciliter la gestion d'images toutes faites de conteneurs.
  - Au fur et à mesure, Docker abandonne le code de **LXC** (mais continue d'utiliser le **chroot**, les **cgroups** et **namespaces**).
  - Le code de base de Docker (notamment **runC**) est open source : l'**Open Container Initiative** vise à standardiser et rendre robuste l'utilisation de containers.
-

# Bénéfices par rapport aux machines virtuelles

Docker permet de faire des “quasi-machines” avec des performances proches du natif.

- Vitesse d'exécution.
  - Flexibilité sur les ressources (mémoire partagée).
  - Moins **complexe** que la virtualisation
  - Plus **standard** que les multiples hyperviseurs
  - notamment moins de bugs d'interaction entre l'hyperviseur et le noyau
- 

# Bénéfices par rapport aux machines virtuelles

VM et conteneurs proposent une flexibilité de manipulation des ressources de calcul mais les machines virtuelles sont trop lourdes pour être multipliées librement :

- elles ne sont pas efficaces pour isoler **chaque application**
- elles ne permettent pas la transformation profonde que permettent les conteneurs :

- le passage à une architecture **microservices**
  - et donc la **scalabilité** pour les besoins des services cloud
- 

## Avantages des machines virtuelles

- Les VM se rapprochent plus du concept de “boîte noire”: l’isolation se fait au niveau du matériel et non au niveau du noyau de l’OS.
- même si une faille dans l’hyperviseur reste possible car l’isolation n’est pas qu’uniquement matérielle
- Les VM sont-elles “plus lentes” ? Pas forcément.
- La RAM est-elle un facteur limite ? Non elle n'est pas cher
- Les CPU pareil : on est rarement bloqués par la puissance du CPU
- Le vrai problème c'est l'I/O : l'accès en entrée-sortie au disque et au réseau
- en réalité Docker peut être plus lent (par défaut) pour l'implémentation de la sécurité réseau (usage du NAT), ou l'implémentation du réseau de Docker Swarm
- pour l'accès au disque : la technologie d'*overlay*

(qui a une place centrale dans Docker) s'améliore, surtout si on utilise un filesystem optimisé pour cela (ZFS, btrfs...).

La comparaison VM / conteneurs est un thème extrêmement vaste et complexe.

---

## Pourquoi utiliser Docker ?

Docker est pensé dès le départ pour faire des **conteneurs applicatifs** :

- **isoler** les modules applicatifs.
  - gérer les **dépendances** en les embarquant dans le conteneur.
  - se baser sur l'**immutabilité** : la configuration d'un conteneur n'est pas faite pour être modifiée après sa création.
  - avoir un **cycle de vie court** -> logique DevOps du "bétail vs. animal de compagnie"
- 

## Pourquoi utiliser Docker ?

Docker modifie beaucoup la "**logistique**" applicative.

- **uniformisation** face aux divers langages de

programmation, configurations et briques logicielles

- **installation sans accroc et automatisation**  
beaucoup plus facile
  - permet de simplifier l'**intégration continue**, la **livraison continue** et le **déploiement continu**
  - **rapproche le monde du développement des opérations** (tout le monde utilise la même technologie)
  - Permet l'adoption plus large de la logique DevOps (notamment le concept *d'infrastructure as code*)
- 

## Infrastructure as Code

### Résumé

- on décrit en mode code un état du système.  
Avantages :
- pas de dérive de la configuration et du système (immutabilité)
- on peut connaître de façon fiable l'état des composants du système
- on peut travailler en collaboration plus facilement (grâce à Git notamment)

- on peut faire des tests
  - on facilite le déploiement de nouvelles instances
- 

## Docker : positionnement sur le marché

- Docker est la technologie ultra-dominante sur le marché de la conteneurisation
- La simplicité d'usage et le travail de standardisation (un conteneur Docker est un conteneur OCI : format ouvert standardisé par l'Open Container Initiative) lui garantissent légitimité et fiabilité
- La logique du conteneur fonctionne, et la bonne documentation et l'écosystème aident !
- **LXC** existe toujours et est très agréable à utiliser, notamment avec **LXD** (développé par Canonical, l'entreprise derrière Ubuntu).
- Il a cependant un positionnement différent : faire des conteneurs pour faire tourner des OS Linux complets.
- **Apache Mesos** : un logiciel de gestion de cluster qui permet de se passer de Docker, mais propose quand même un support pour les conteneurs OCI (Docker) depuis 2016.

- **Podman** : une alternative à Docker qui utilise la même syntaxe que Docker pour faire tourner des conteneurs OCI (Docker) qui propose un mode *rootless* et *daemonless* intéressant.
  - **systemd-nspawn** : technologie de conteneurs isolés proposée par systemd
- 

## 1 - Manipulation des conteneurs

### Terminologie et concepts fondamentaux

Deux concepts centraux :

- Une **image** : un modèle pour créer un conteneur
- Un **conteneur** : l'instance qui tourne sur la machine.

Autres concepts primordiaux :

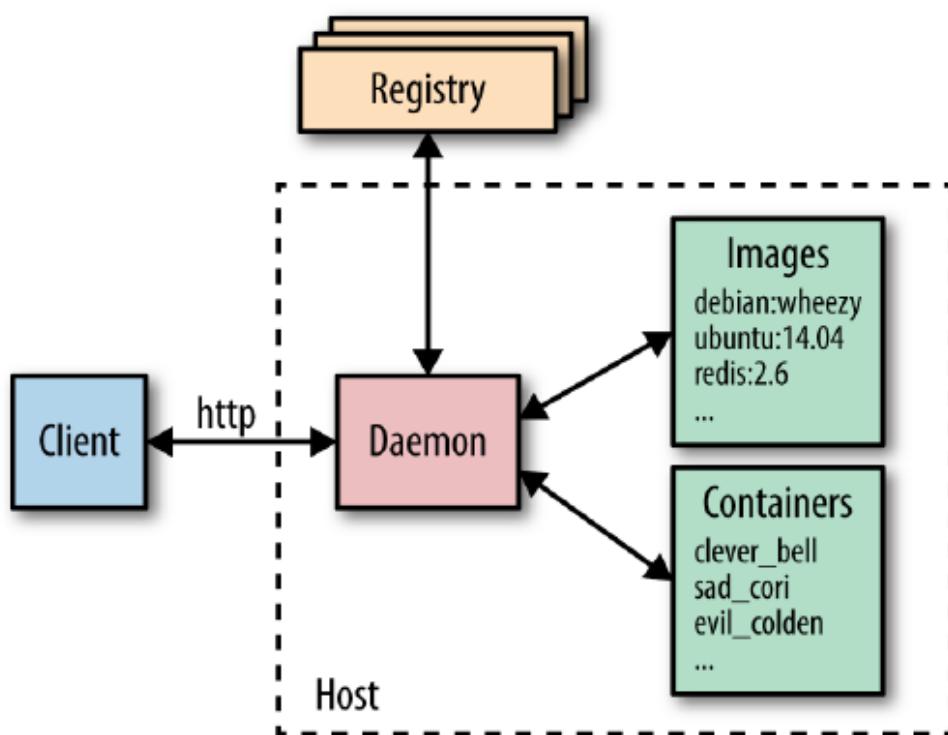
- Un **volume** : un espace virtuel pour gérer le stockage d'un conteneur et le partage entre conteneurs.
- un **registry** : un serveur ou stocker des artefacts docker c'est à dire des images versionnées.
- un **orchestrateur** : un outil qui gère

automatiquement le cycle de vie des conteneurs (création/suppression).

---

## Visualiser l'architecture Docker

### Daemon - Client - images - registry



## L'écosystème Docker

- **Docker Compose** : Un outil pour décrire des applications multiconteneurs.
- **Docker Machine** : Un outil pour gérer le déploiement Docker sur plusieurs machines depuis un hôte.
- **Docker Hub** : Le service d'hébergement d'images

proposé par Docker Inc. (le registry officiel)

---

## L'environnement de développement

- Docker Engine pour lancer des commandes docker
  - Docker Compose pour lancer des application multiconteneurs
  - Portainer, un GUI Docker
  - VirtualBox pour avoir une VM Linux quand on est sur Windows
- 

## Installer Docker sur Windows ou MacOS

Docker est basé sur le noyau Linux :

- En **production** il fonctionne nécessairement sur un **Linux** (virtualisé ou *bare metal*)
  - Pour **développer et déployer**, il marche parfaitement sur **MacOS** et **Windows** mais avec une méthode de **virtualisation** :
    - virtualisation optimisée via un hyperviseur
    - ou virtualisation avec logiciel de virtualisation “classique” comme VMWare ou VirtualBox.
-

# Installer Docker sur Windows

Quatre possibilités :

- Solution **Docker Desktop WSL2** :
  - Fonctionne avec Windows Subsystem for Linux :  
c'est une VM Linux très bien intégrée à Windows
  - Le meilleur des deux mondes ?
  - Workflow similaire à celui d'un serveur Linux
  - Solution VirtualBox : on utilise **Docker Engine** dans une VM Linux
  - Utilise une VM Linux avec VirtualBox
  - Workflow identique à celui d'un serveur Linux
  - Proche de la réalité de l'administration système actuelle
- 

# Installer Docker sous MacOS

- Solution standard : on utilise **Docker Desktop for MacOS** (fonctionne avec la bibliothèque HyperKit qui fait de l'hypervision)
  - Solution Virtualbox / *legacy* : On utilise une VM Linux
-

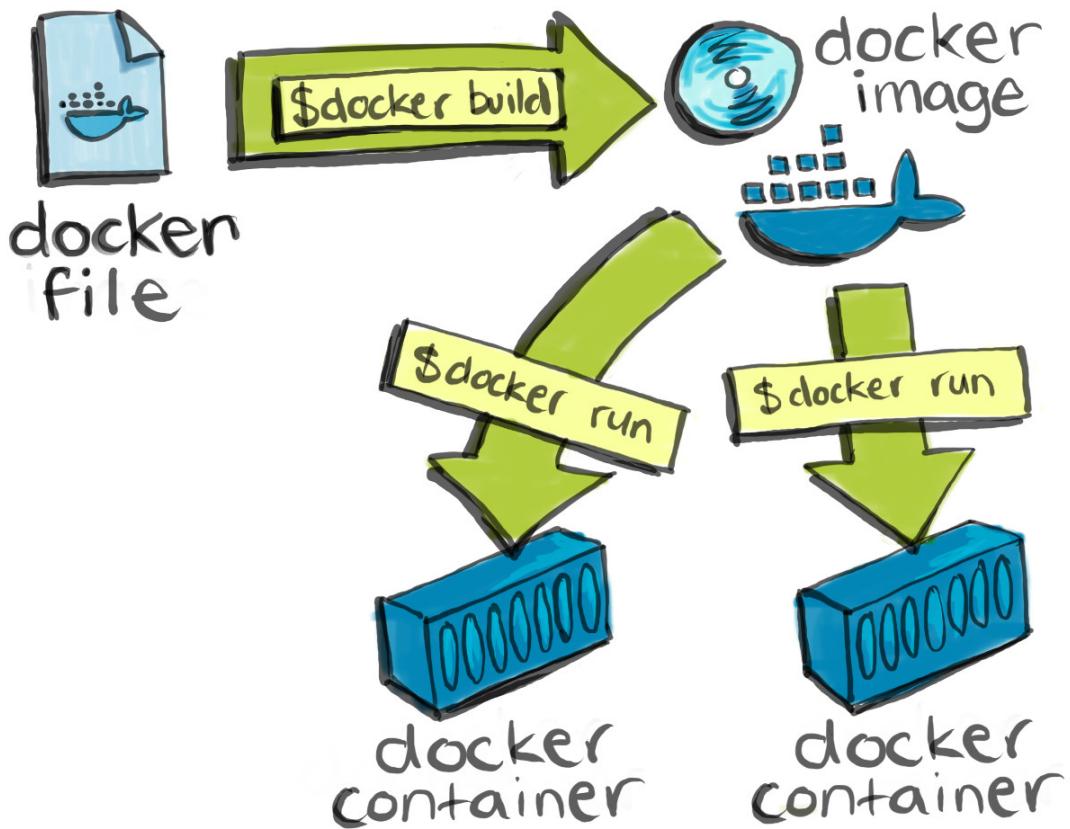
# Installer Docker sur Linux

Pas de virtualisation nécessaire car Docker (le Docker Engine) utilise le noyau du système natif.

- On peut l'installer avec le gestionnaire de paquets de l'OS mais cette version peut être trop ancienne.
  - Sur **Ubuntu** ou **CentOS** la méthode conseillée est d'utiliser les paquets fournis dans le dépôt officiel Docker (vous pouvez avoir des surprises avec la version *snap* d'Ubuntu).
  - Il faut pour cela ajouter le dépôt et les signatures du répertoire de packages Docker.
  - Documentation Ubuntu : <https://docs.docker.com/install/linux/docker-ce/ubuntu/>
- 

## Les images et conteneurs

### Les images



**Docker** possède à la fois un module pour lancer les applications (runtime) et un **outil de build** d'application.

- Une image est le **résultat** d'un build :
- on peut la voir un peu comme une boîte “modèle” : on peut l'utiliser plusieurs fois comme base de création de containers identiques, similaires ou différents.

Pour lister les images on utilise :

```
docker images
docker image ls
```

## Les conteneurs

- Un conteneur est une instance en cours de fonctionnement (“vivante”) d’une image.
- un conteneur en cours de fonctionnement est un processus (et ses processus enfants) qui tourne dans le Linux hôte (mais qui est isolé de celui-ci)

## Commandes Docker

Docker fonctionne avec des sous-commandes et propose de grandes quantités d’options pour chaque commande.

Utilisez `--help` au maximum après chaque commande, sous-commande ou sous-sous-commandes

---

## Pour vérifier l’état de Docker

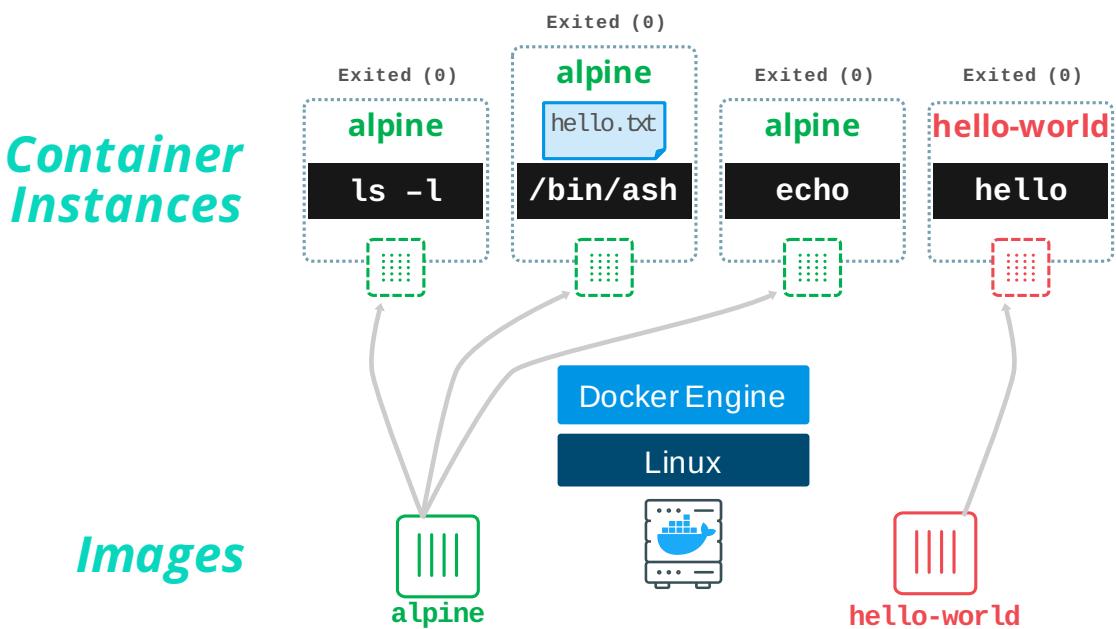
- Les commandes de base pour connaître l’état de Docker sont :

```
docker info # affiche plein  
d'information sur l'engine avec lequel  
vous êtes en contact  
docker ps    # affiche les conteneurs  
en train de tourner  
docker ps -a # affiche également les
```

conteneurs arrêtés

## Créer et lancer un conteneur

# Docker Container Isolation



- Un conteneur est une instance en cours de fonctionnement (“vivante”) d’une image.

```
docker run [-d] [-p port_h:port_c] [-v  
dossier_h:dossier_c] <image>  
<commande>
```

[créé et lance le conteneur

- **L’ordre des arguments est important !**
- **Un nom est automatiquement généré pour le conteneur à moins de fixer le nom avec --name**

- On peut facilement lancer autant d'instances que nécessaire tant qu'il n'y a **pas de collision de nom ou de port**.
- 

## Options docker run

- Les options facultatives indiquées ici sont très courantes.
  - **-d** permet\* de lancer le conteneur en mode **daemon** ou **détaché** et libérer le terminal
  - **-p** permet de mapper un *port réseau* entre l'intérieur et l'extérieur du conteneur, typiquement lorsqu'on veut accéder à l'application depuis l'hôte.
  - **-v** permet de monter un *volume* partagé entre l'hôte et le conteneur.
  - **--rm** (comme *remove*) permet de supprimer le conteneur dès qu'il s'arrête.
  - **-it** permet de lancer une commande en mode *interactif* (un terminal comme bash).
  - **-a** (ou **--attach**) permet de se connecter à l'entrée-sortie du processus dans le container.
- 

## Commandes Docker

- Le démarrage d'un conteneur est lié à une **commande**.
- Si le conteneur n'a pas de commande, il s'arrête dès qu'il a fini de démarrer

```
docker run debian # s'arrête tout de suite
```

- Pour utiliser une commande on peut simplement l'ajouter à la fin de la commande run.

```
docker run debian echo 'attendre 10s'  
&& sleep 10 # s'arrête après 10s
```

## Stopper et redémarrer un conteneur

docker run créé un nouveau conteneur à chaque fois.

```
docker stop <nom_ou_id_conteneur> # ne détruit pas le conteneur  
docker start <nom_ou_id_conteneur> # le conteneur a déjà été créé  
docker start --attach  
<nom_ou_id_conteneur> # lance le conteneur et s'attache à la sortie standard
```

# Isolation des conteneurs

- Les conteneurs sont plus que des processus, ce sont des boîtes isolées grâce aux **namespaces** et **cgroups**
  - Depuis l'intérieur d'un conteneur, on a l'impression d'être dans un Linux autonome.
  - Plus précisément, un conteneur est lié à un système de fichiers (avec des dossiers `/bin`, `/etc`, `/var`, des exécutables, des fichiers...), et possède des métadonnées (stockées en json quelque part par Docker)
  - Les utilisateurs Unix à l'intérieur du conteneur ont des UID et GID qui existent classiquement sur l'hôte mais ils peuvent correspondre à un utilisateur Unix sans droits sur l'hôte si on utilise les *user namespaces*.
- 

## Introspection de conteneur

- La commande `docker exec` permet d'exécuter une commande à l'intérieur du conteneur **s'il est lancé**.
- Une utilisation typique est d'introspecter un

conteneur en lançant bash (ou sh).

```
docker exec -it <conteneur> /bin/bash
```

---

## Docker Hub : télécharger des images

Une des forces de Docker vient de la distribution d'images :

- pas besoin de dépendances, on récupère une boîte autonome
- pas besoin de multiples versions en fonction des OS

Dans ce contexte un élément qui a fait le succès de Docker est le Docker Hub : [hub.docker.com](https://hub.docker.com)

Il s'agit d'un répertoire public et souvent gratuit d'images (officielles ou non) pour des milliers d'applications pré-configurées.

---

## Docker Hub:

- On peut y chercher et trouver presque n'importe quel logiciel au format d'image Docker.
- Il suffit pour cela de chercher l'identifiant et la version de l'image désirée.
- Puis utiliser `docker run [<compte>/]`

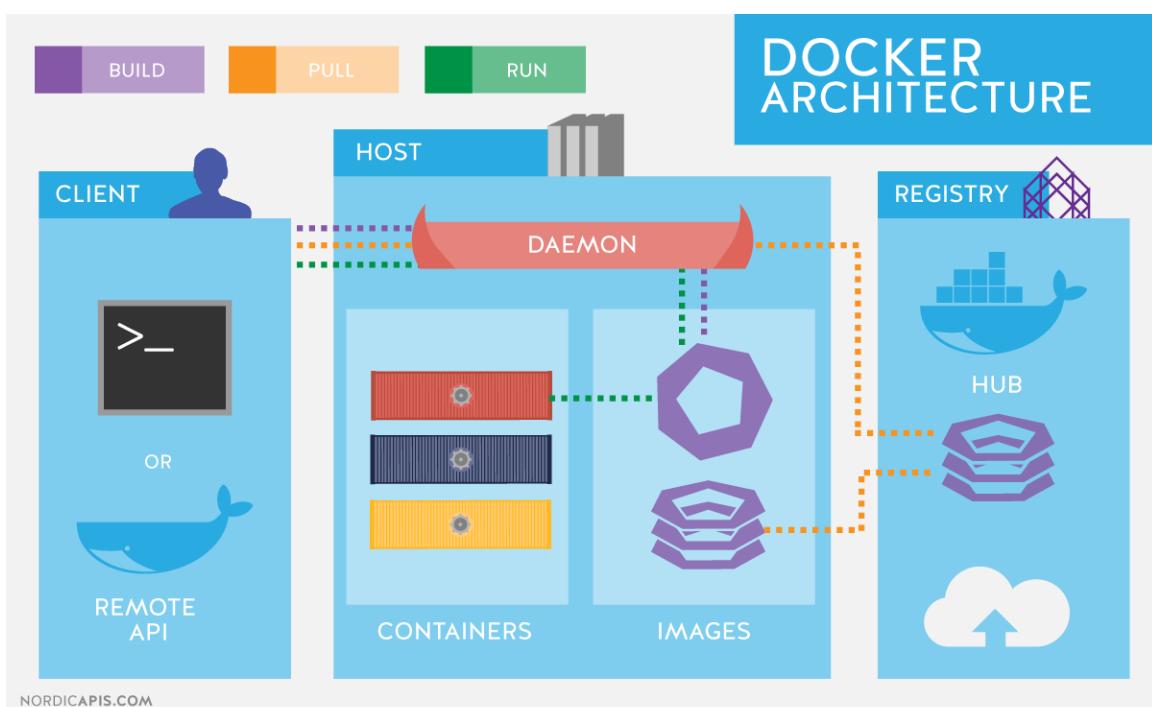
<id\_image>:<version>

- La partie compte est le compte de la personne qui a poussé ses images sur le Docker Hub. Les images Docker officielles (ubuntu par exemple) ne sont pas liées à un compte : on peut écrire simplement ubuntu:focal.
- On peut aussi juste télécharger l'image : docker pull <image>

On peut également y créer un compte gratuit pour pousser et distribuer ses propres images, ou installer son propre serveur de distribution d'images privé ou public, appelé **registry**.

---

## En résumé



# TP 1 - Installer Docker et jouer avec

## Premier TD : on installe Docker et on joue avec

### Installer Docker sur la VM Ubuntu dans Guacamole

- Accédez à votre VM via l'interface Guacamole
- Pour accéder au copier-coller de Guacamole, il faut appuyer sur **Ctrl+Alt+Shift** et utiliser la zone de texte qui s'affiche (réappuyer sur **Ctrl+Alt+Shift** pour revenir à la VM).
- Pour installer Docker, suivez la [documentation officielle pour installer Docker sur Ubuntu](#), depuis “Install using the repository” jusqu’aux deux commandes `sudo apt-get update` et `sudo apt-get install docker-ce docker-ce-cli containerd.io`.
- Docker nous propose aussi une installation en une ligne (*one-liner*), moins sécurisée : `curl -sSL https://get.docker.com | sudo sh`
- Lancez `sudo docker run hello-world`. Bien lire le message renvoyé (le traduire sur [DeepL](#) si

nécessaire). Que s'est-il passé ?

- Il manque les droits pour exécuter Docker sans passer par sudo à chaque fois.
- Le daemon tourne toujours en root
- Un utilisateur ne peut accéder au client que s'il est membre du groupe docker
- Ajoutez-le au groupe avec la commande sudo usermod -aG docker <user> (en remplaçant <user> par ce qu'il faut)
- Pour actualiser la liste de groupes auquel appartient l'utilisateur, redémarrez la VM avec sudo reboot puis reconnectez-vous avec Guacamole pour que la modification sur les groupes prenne effet.

Pour les prochaines fois, Docker nous propose aussi une installation en une ligne (*one-liner*) :

```
curl -sSL https://get.docker.com |  
sudo sh
```

## Autocomplétion

- Pour vous faciliter la vie, ajoutez le plugin *autocomplete* pour Docker et Docker Compose à

bash en copiant les commandes suivantes :

```
sudo apt update  
sudo apt install bash-completion curl  
sudo curl -L  
https://raw.githubusercontent.com/docker/compose/1.24.1/contrib/completion/bash/docker-compose -o  
/etc/bash_completion.d/docker-compose
```

**Important:** Vous pouvez désormais appuyer sur la touche pour utiliser l'autocomplétion quand vous écrivez des commandes Docker

---

## Pour vérifier l'installation

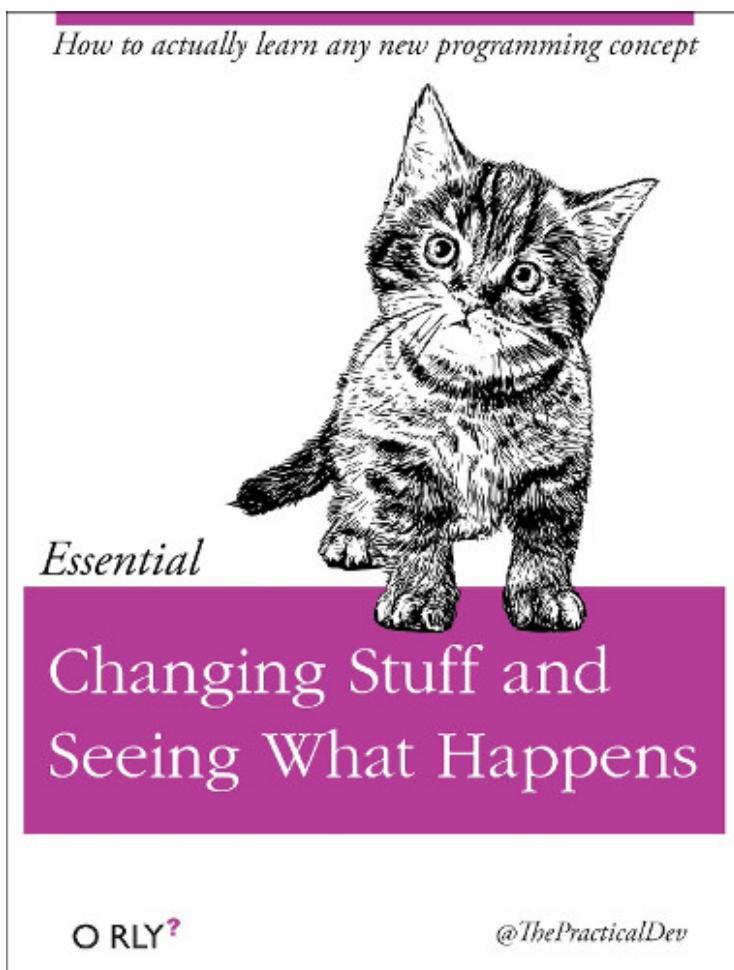
- Les commandes de base pour connaître l'état de Docker sont :

```
docker info # affiche plein d'information sur l'engine avec lequel vous êtes en contact  
docker ps    # affiche les conteneurs en train de tourner  
docker ps -a # affiche également les conteneurs arrêtés
```

## Manipuler un conteneur

- **Commandes utiles** : <https://devhints.io/docker>
- **Documentation docker run** :  
<https://docs.docker.com/engine/reference/run/>

Mentalité :



Il faut aussi prendre l'habitude de bien lire ce que la console indique après avoir passé vos commandes.

Avec l'aide du support et de `--help`, et en notant sur une feuille ou dans un fichier texte les commandes utilisées :

- Lancez simplement un conteneur Debian en mode `attach`. Que se passe-t-il ?

- Lancez un conteneur Debian (`docker run` puis les arguments nécessaires, cf. l'aide `--help`)  
avec l'option “mode détaché” et la commande passée au conteneur `echo "Je suis le conteneur basé sur Debian"`. Rien n'apparaît. En effet en mode détaché la sortie standard n'est pas connectée au terminal.
- Lancez `docker logs` avec le nom ou l'id du conteneur. Vous devriez voir le résultat de la commande `echo` précédente.
- Affichez la liste des conteneurs en cours d'exécution
- Affichez la liste des conteneurs en cours d'exécution et arrêtés.
- Lancez un conteneur `debian` **en mode détaché** avec la commande `sleep 3600`
- Réaffichez la liste des conteneurs qui tournent
- Tentez de stopper le conteneur, que se passe-t-il ?

```
docker stop <conteneur>
```

**NB:** On peut désigner un conteneur soit par le nom qu'on lui a donné, soit par le nom généré automatiquement, soit par son empreinte (toutes

ces informations sont indiquées dans un docker ps ou docker ps -a). L'autocomplétion fonctionne avec les deux noms.

- Trouvez comment vous débarrasser d'un conteneur récalcitrant (si nécessaire, relancez un conteneur avec la commande sleep 3600 en mode détaché).
- Tentez de lancer deux conteneurs avec le nom `debian_container`

Le nom d'un conteneur doit être unique (à ne pas confondre avec le nom de l'image qui est le modèle utilisé à partir duquel est créé le conteneur).

- Créez un conteneur avec le nom `debian2`

```
docker run debian -d --name debian2
sleep 500
```

- Lancez un conteneur `debian` en mode interactif (options `-i -t`) avec la commande `/bin/bash` et le nom `debian_interactif`.
- Explorer l'intérieur du conteneur : il ressemble à un OS Linux Debian normal.

---

## Chercher sur Docker Hub

- Visitez [hub.docker.com](https://hub.docker.com)
- Cherchez l'image de Nginx (un serveur web), et téléchargez la dernière version (pull).
- Lancez un conteneur Nginx. Notez que lorsque l'image est déjà téléchargée le lancement d'un conteneur est quasi instantané.

```
docker run --name "test_nginx" nginx
```

Ce conteneur n'est pas très utile, car on a oublié de configurer un port exposé sur localhost.

- Trouvez un moyen d'accéder quand même au Nginx à partir de l'hôte Docker (indice : quelle adresse IP le conteneur possède-t-il ?).
- Arrêtez le(s) conteneur(s) nginx créé(e)s.
- Relancez un nouveau conteneur nginx avec cette fois-ci le port correctement configuré dès le début pour pouvoir visiter votre Nginx en local.

```
docker run -p 8080:80 --name
"test2_nginx" nginx # la syntaxe est :
port_hote:port_container
```

- En visitant l'adresse et le port associé au conteneur Nginx, on doit voir apparaître des logs Nginx dans son terminal car on a lancé le conteneur en mode

*attach.*

- Supprimez ce conteneur. NB : On doit arrêter un conteneur avant de le supprimer, sauf si on utilise l'option “-f”.
- 

On peut lancer des logiciels plus ambitieux, comme par exemple Funkwhale, une sorte d'iTunes en web qui fait aussi réseau social :

```
docker run --name funky_conteneur -p  
80:80 funkwhale/all-in-one:1.0.1
```

Vous pouvez visiter ensuite ce conteneur Funkwhale sur le port 80 (après quelques secondes à suivre le lancement de l'application dans les logs) ! Mais il n'y aura hélas pas de musique dedans :(

*Attention à ne jamais lancer deux containers connectés au même port sur l'hôte, sinon cela échouera !*

- Supprimons ce conteneur :

```
docker rm -f funky_conteneur
```

***Facultatif : Wordpress, MySQL et les variables d'environnement***

- Lancez un conteneur Wordpress joignable sur le port 8080 à partir de l'image officielle de Wordpress du Docker Hub
- Visitez ce Wordpress dans le navigateur

Nous pouvons accéder au Wordpress, mais il n'a pas encore de base MySQL configurée. Ce serait un peu dommage de configurer cette base de données à la main. Nous allons configurer cela à partir de variables d'environnement et d'un deuxième conteneur créé à partir de l'image mysql.

Depuis Ubuntu:

- Il va falloir mettre ces deux conteneurs dans le même réseau (nous verrons plus tard ce que cela implique), créons ce réseau :

```
docker network create wordpress
```

- Cherchez le conteneur mysql version 5.7 sur le Docker Hub.
- Utilisons des variables d'environnement pour préciser le mot de passe root, le nom de la base de données et le nom d'utilisateur de la base de données (trouver la documentation sur le Docker Hub).

- Il va aussi falloir définir un nom pour ce conteneur
- inspectez le conteneur MySQL avec `docker inspect`
- Faites de même avec la documentation sur le Docker Hub pour préconfigurer l'app Wordpress.
- En plus des variables d'environnement, il va falloir le mettre dans le même réseau, et exposer un port
- regardez les logs du conteneur Wordpress avec `docker logs`
- visitez votre app Wordpress et terminez la configuration de l'application : si les deux conteneurs sont bien configurés, on ne devrait pas avoir à configurer la connexion à la base de données
- avec `docker exec`, visitez votre conteneur Wordpress. Pouvez-vous localiser le fichier `wp-config.php` ? Une fois localisé, utilisez `docker cp` pour le copier sur l'hôte.

## Faire du ménage

Il est temps de faire un petit `docker stats` pour découvrir l'utilisation du CPU et de la RAM de vos

conteneurs !

- Lancez la commande `docker ps -aq -f status=exited`. Que fait-elle ?
- Combinez cette commande avec `docker rm` pour supprimer tous les conteneurs arrêtés (indice : en Bash, une commande entre les parenthèses de “`$()`” est exécutée avant et utilisée comme chaîne de caractère dans la commande principale)
- S'il y a encore des conteneurs qui tournent (`docker ps`), supprimez un des conteneurs restants en utilisant l'autocomplétion et l'option adéquate
- Listez les images
- Supprimez une image
- Que fait la commande `docker image prune -a` ?

## Décortiquer un conteneur

- En utilisant la commande `docker export votre_conteneur -o conteneur.tar`, puis `tar -C conteneur_decompresser -xvf conteneur.tar` pour décompresser un conteneur

Docker, explorez (avec l'explorateur de fichiers par exemple) jusqu'à trouver l'exécutable principal contenu dans le conteneur.

## Portainer

Portainer est un portail web pour gérer une installation Docker via une interface graphique. Il va nous faciliter la vie.

- Lancer une instance de Portainer :

```
docker volume create portainer_data
docker run --detach --name portainer \
    -p 9000:9000 \
    -v portainer_data:/data \
    -v /var/run/docker.sock:/var/run
/docker.sock \
    portainer/portainer-ce
```

- Remarque sur la commande précédente : pour que Portainer puisse fonctionner et contrôler Docker lui-même depuis l'intérieur du conteneur il est nécessaire de lui donner accès au socket de l'API Docker de l'hôte grâce au paramètre `--volume` ci-dessus.
- Visitez ensuite la page <http://localhost:9000> ou l'adresse IP publique de votre serveur Docker sur

le port 9000 pour accéder à l'interface.

- il faut choisir l'option “local” lors de la configuration
- Créez votre user admin et choisir un mot de passe avec le formulaire.
- Explorez l'interface de Portainer.
- Créez un conteneur.

## 2 - Images et conteneurs

### Créer une image en utilisant un Dockerfile

- Jusqu'ici nous avons utilisé des images toutes prêtes.
- Une des fonctionnalités principales de Docker est de pouvoir facilement construire des images à partir d'un simple fichier texte : **le Dockerfile**.

### Le processus de build Docker

- Un image Docker ressemble un peu à une VM car on peut penser à un Linux “freezé” dans un état.
- En réalité c'est assez différent : il s'agit uniquement d'un système de fichier (par couches ou *layers*) et d'un manifeste JSON (des méta-données).

- Les images sont créés en empilant de nouvelles couches sur une image existante grâce à un système de fichiers qui fait du *union mount*.
- Chaque nouveau build génère une nouvelle image dans le répertoire des images (/var/lib/docker/images) (attention ça peut vite prendre énormément de place)
- On construit les images à partir d'un fichier Dockerfile en décrivant procéduralement (étape par étape) la construction.

## Dockerfiles

### Dockerfile:

```
FROM alpine
RUN apk update && apk add nodejs
COPY . /app
WORKDIR /app
CMD ["node", "index.js"]
```

**FROM** alpine



**RUN** apk

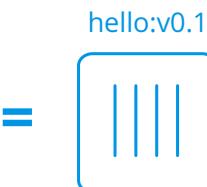
apk update  
apk add nodejs

**COPY** . /app

host/index.js -->  
container/app

**WORKDIR** /app  
**CMD** "node" "index.js"

\$ cd /app  
\$ node index.js



## Exemple de Dockerfile :

```
FROM debian:latest

RUN apt update && apt install htop

CMD ['sleep 1000']
```

- La commande pour construire l'image est :

```
docker build [-t tag] [-f dockerfile]
<build_context>
```

- généralement pour construire une image on se place directement dans le dossier avec le Dockerfile et les éléments de contexte nécessaire (programme, config, etc), le contexte est donc le caractère ., il est obligatoire de préciser un contexte.
- exemple : `docker build -t mondebian .`
- Le **Dockerfile** est un fichier procédural qui permet de décrire l'installation d'un logiciel (la configuration d'un container) en enchaînant des instructions Dockerfile (en MAJUSCULE).
- Exemple:

```
# our base image
FROM alpine:3.5
```

```
# Install python and pip
RUN apk add --update py2-pip

# upgrade pip
RUN pip install --upgrade pip

# install Python modules needed by the
Python app
COPY requirements.txt /usr/src/app/
RUN pip install --no-cache-dir -r
/usr/src/app/requirements.txt

# copy files required for the app to
run
COPY app.py /usr/src/app/
COPY templates/index.html /usr/src
/app/templates/

# tell the port number the container
should expose
EXPOSE 5000

# run the application
CMD ["python", "/usr/src/app/app.py"]
```

## **Instruction FROM**

- L'image de base à partir de laquelle est construite l'image actuelle.

## **Instruction RUN**

- Permet de lancer une commande shell (installation, configuration).

## **Instruction ADD**

- Permet d'ajouter des fichiers depuis le contexte de build à l'intérieur du conteneur.
  - Généralement utilisé pour ajouter le code du logiciel en cours de développement et sa configuration au conteneur.
- 

## **Instruction CMD**

- Généralement à la fin du Dockerfile : elle permet de préciser la commande par défaut lancée à la création d'une instance du conteneur avec `docker run`. On l'utilise avec une liste de paramètres

```
CMD ["echo 'Conteneur démarré'"]
```

## Instruction ENTRYPPOINT

- Précise le programme de base avec lequel sera lancé la commande

```
ENTRYPOINT [/usr/bin/python3]
```

## CMD et ENTRYPPOINT

- Ne surtout pas confondre avec RUN qui exécute une commande Bash uniquement pendant la construction de l'image.

L'instruction CMD a trois formes :

- CMD ["executable", "param1", "param2"]  
(*exec form*, forme à préférer)
- CMD ["param1", "param2"] (combinée à une instruction ENTRYPPOINT)
- CMD command param1 param2 (*shell form*)

Si l'on souhaite que notre container lance le même exécutable à chaque fois, alors on peut opter pour l'usage d'ENTRYPPOINT en combinaison avec CMD.

## Instruction ENV

- Une façon recommandée de configurer vos applications Docker est d'utiliser les variables d'environnement UNIX, ce qui permet une configuration “au *runtime*”.
- 

## Instruction HEALTHCHECK

HEALTHCHECK permet de vérifier si l'app contenue dans un conteneur est en bonne santé.

```
HEALTHCHECK CMD curl --fail  
http://localhost:5000/health || exit 1
```

---

## Les variables

On peut utiliser des variables d'environnement dans les Dockerfiles. La syntaxe est \${...}.

Exemple :

```
FROM busybox  
ENV FOO=/bar  
WORKDIR ${FOO}      # WORKDIR /bar  
ADD . $FOO          # ADD . /bar  
COPY \${FOO} /quux # COPY ${FOO} /quux
```

Se référer au [mode d'emploi](#) pour la logique plus précise de fonctionnement des variables.

## Documentation

- Il existe de nombreuses autres instructions possibles très clairement décrites dans la documentation officielle : <https://docs.docker.com/engine/reference/builder/>
- 

## Lancer la construction

- La commande pour lancer la construction d'une image est :

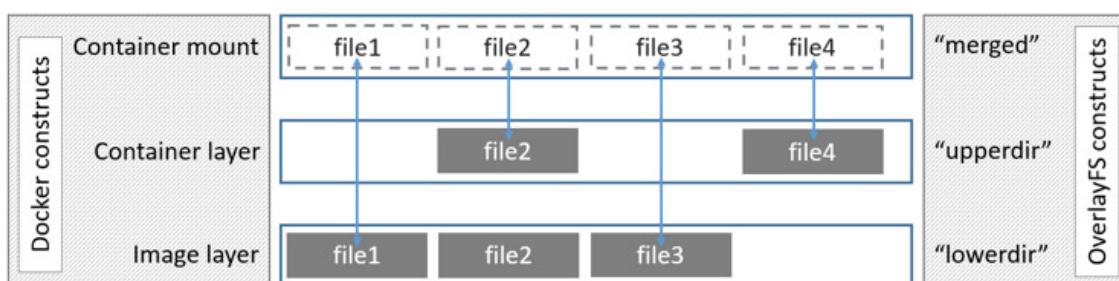
```
docker build [-t <tag:version>] [-f  
<chemin_du_dockerfile>]  
<contexte_de_construction>
```

- Lors de la construction, Docker télécharge l'image de base. On constate plusieurs téléchargements en parallèle.
- Il lance ensuite la séquence des instructions du Dockerfile.
- Observez l'historique de construction de l'image avec `docker image history <image>`

- Il lance ensuite la série d'instructions du Dockerfile et indique un *hash* pour chaque étape.
  - C'est le *hash* correspondant à un *layer* de l'image
- 

## Les layers et la mise en cache

- Docker construit les images comme une série de “couches” de fichiers successives.
- On parle d'**Union Filesystem** car chaque couche (de fichiers) écrase la précédente.



- Chaque couche correspond à une instruction du Dockerfile.
- `docker image history <conteneur>` permet d'afficher les layers, leur date de construction et taille respectives.
- Ce principe est au cœur de l'**immutabilité** des images Docker.
- Au lancement d'un container, le Docker Engine rajoute une nouvelle couche de filesystem “normal” read/write par dessus la pile des couches de

l'image.

- `docker diff <container>` permet d'observer les changements apportés au conteneur depuis le lancement.
- 

## Optimiser la création d'images

- Les images Docker ont souvent une taille de plusieurs centaines de **mégaoctets** voire parfois **gigaoctets**. `docker image ls` permet de voir la taille des images.
- Or, on construit souvent plusieurs dizaines de versions d'une application par jour (souvent automatiquement sur les serveurs d'intégration continue).
- L'espace disque devient alors un sérieux problème.
- Le principe de Docker est justement d'avoir des images légères car on va créer beaucoup de conteneurs (un par instance d'application/service).
- De plus on télécharge souvent les images depuis un registry, ce qui consomme de la bande passante.

La principale **bonne pratique** dans la construction d'images est de **limiter leur taille au maximum**.

---

## Limiter la taille d'une image

- Choisir une image Linux de base **minimale**:
  - Une image ubuntu complète pèse déjà presque une soixantaine de mégaoctets.
  - mais une image trop rudimentaire (busybox) est difficile à débugger et peu bloquer pour certaines tâches à cause de binaires ou de bibliothèques logicielles qui manquent (compilation par exemple).
  - Souvent on utilise des images de base construites à partir de alpine qui est un bon compromis (6 mégaoctets seulement et un gestionnaire de paquets apk).
  - Par exemple python3 est fourni en version python:alpine (99 Mo), python:3-slim (179 Mo) et python:latest (918 Mo).
- 

## Les multi-stage builds

Quand on tente de réduire la taille d'une image, on a recours à un tas de techniques. Avant, on utilisait deux Dockerfile différents : un pour la version prod, léger, et un pour la version dev, avec des

outils en plus. Ce n'était pas idéal. Par ailleurs, il existe une limite du nombre de couches maximum par image (42 layers). Souvent on enchaînait les commandes en une seule pour économiser des couches (souvent, les commandes RUN et ADD), en y perdant en lisibilité.

Maintenant on peut utiliser les multistage builds.

Avec les multi-stage builds, on peut utiliser plusieurs instructions FROM dans un Dockerfile. Chaque instruction FROM utilise une base différente. On sélectionne ensuite les fichiers intéressants (des fichiers compilés par exemple) en les copiant d'un stage à un autre.

Exemple de Dockerfile utilisant un multi-stage build :

```
FROM golang:1.7.3 AS builder
WORKDIR /go/src/github.com/alexellis
/href-counter/
RUN go get -d -v golang.org/x/net/html
COPY app.go .
RUN CGO_ENABLED=0 GOOS=linux go build
-a -installsuffix cgo -o app .

FROM alpine:latest
```

```
RUN apk --no-cache add ca-certificates  
WORKDIR /root/  
COPY --from=builder /go/src/github.com/  
/alexellis(href-counter/app .  
CMD [ "./app" ]
```

---

## Créer des conteneurs personnalisés

- Il n'est pas nécessaire de partir d'une image Linux vierge pour construire un conteneur.
- On peut utiliser la directive FROM avec n'importe quelle image.
- De nombreuses applications peuvent être configurées en étendant une image officielle
- *Exemple : une image Wordpress déjà adaptée à des besoins spécifiques.*
- L'intérêt ensuite est que l'image est disponible préconfigurée pour construire ou mettre à jour une infrastructure, ou lancer plusieurs instances (plusieurs containers) à partir de cette image.
- C'est grâce à cette fonctionnalité que Docker peut être considéré comme un outil d'*infrastructure as code*.
- On peut également prendre une sorte de snapshot

du conteneur (de son système de fichiers, pas des processus en train de tourner) sous forme d'image avec `docker commit <image>` et `docker push`.

---

## **Publier des images vers un registry privé**

- Généralement les images spécifiques produites par une entreprise n'ont pas vocation à finir dans un dépôt public.
  - On peut installer des **registries privés**.
  - On utilise alors `docker login <adresse_repo>` pour se logger au registry et le nom du registry dans les tags de l'image.
  - Exemples de registries :
  - **Gitlab** fournit un registry très intéressant car intégré dans leur workflow DevOps.
- 

## **TP 2 - Images et conteneurs**

### **Découverte d'une application web flask**

- Récupérez d'abord une application Flask exemple

en la clonant :

```
git clone https://github.com/uptime-formation/microblog/
```

- Ouvrez VSCode avec le dossier `microblog` en tapant code `microblog` ou bien en lançant VSCode avec code puis en cliquant sur Open Folder.
- Dans VSCode, vous pouvez faire Terminal > New Terminal pour obtenir un terminal en bas de l'écran.
- Observons ensemble le code dans VSCode.

## Passons à Docker

Déployer une application Flask manuellement à chaque fois est relativement pénible. Pour que les dépendances de deux projets Python ne se perturbent pas, il faut normalement utiliser un environnement virtuel `virtualenv` pour séparer ces deux apps. Avec Docker, les projets sont déjà isolés dans des conteneurs. Nous allons donc construire une image de conteneur pour empaqueter l'application et la manipuler plus facilement. Assurez-vous que Docker est installé.

Pour connaître la liste des instructions des Dockerfiles et leur usage, se référer au [manuel de référence sur les Dockerfiles](#).

- Dans le dossier du projet ajoutez un fichier nommé Dockerfile et sauvegardez-le
- Normalement, VSCode vous propose d'ajouter l'extension Docker. Il va nous faciliter la vie, installez-le. Une nouvelle icône apparaît dans la barre latérale de gauche, vous pouvez y voir les images téléchargées et les conteneurs existants. L'extension ajoute aussi des informations utiles aux instructions Dockerfile quand vous survolez un mot-clé avec la souris.
- Ajoutez en haut du fichier : FROM  
ubuntu:latest Cette commande indique que notre image de base est la dernière version de la distribution Ubuntu.
- Nous pouvons déjà construire un conteneur à partir de ce modèle Ubuntu vide : docker build -t  
microblog .
- Une fois la construction terminée lancez le conteneur.
- Le conteneur s'arrête immédiatement. En effet il ne

contient aucune commande bloquante et nous n'avons précisé aucune commande au lancement. Pour pouvoir observer le conteneur convenablement il faudrait faire tourner quelque chose à l'intérieur. Ajoutez à la fin du fichier la ligne : `CMD [ "/bin/sleep" , "3600" ]` Cette ligne indique au conteneur d'attendre pendant 3600 secondes comme au TP précédent.

- Reconstruisez l'image et relancez un conteneur
- Affichez la liste des conteneurs en train de fonctionner
- Nous allons maintenant rentrer dans le conteneur en ligne de commande pour observer. Utilisez la commande : `docker exec -it <id_du_conteneur> /bin/bash`
- Vous êtes maintenant dans le conteneur avec une invite de commande. Utilisez quelques commandes Linux pour le visiter rapidement (`ls`, `cd...`).
- Il s'agit d'un Linux standard, mais il n'est pas conçu pour être utilisé comme un système complet, juste pour une application isolée. Il faut maintenant ajouter notre application Flask à l'intérieur. Dans le Dockerfile supprimez la ligne CMD, puis ajoutez :

```
RUN apt-get update -y  
RUN apt-get install -y python3-pip
```

- Reconstruisez votre image. Si tout se passe bien, poursuivez.
- Pour installer les dépendances python et configurer la variable d'environnement Flask ajoutez:

```
COPY ./requirements.txt  
/requirements.txt  
RUN pip3 install -r requirements.txt  
ENV FLASK_APP microblog.py
```

- Reconstruisez votre image. Si tout se passe bien, poursuivez.
- Ensuite, copions le code de l'application à l'intérieur du conteneur. Pour cela ajoutez les lignes :

```
COPY ./ /microblog  
WORKDIR /microblog
```

Cette première ligne indique de copier tout le contenu du dossier courant sur l'hôte dans un dossier /microblog à l'intérieur du conteneur. Nous n'avons pas copié les requirements en même temps pour pouvoir tirer partie des fonctionnalités de cache de Docker, et ne pas avoir à retélécharger les dépendances de l'application à chaque fois que

I'on modifie le contenu de l'app.

Puis, dans la 2e ligne, le dossier courant dans le conteneur est déplacé à / .

- Reconstruisez votre image. **Observons que le build recommence à partir de l'instruction modifiée. Les layers précédents avaient été mis en cache par le Docker Engine.**
- Si tout se passe bien, poursuivez.
- Enfin, ajoutons la section de démarrage à la fin du Dockerfile, c'est un script appelé boot . sh :
  - Reconstruisez l'image et lancez un conteneur basé sur l'image en ouvrant le port 5000 avec la commande : `docker run -p 5000:5000 microblog`
  - Naviguez dans le navigateur à l'adresse `localhost:5000` pour admirer le prototype microblog.
  - Lancez un deuxième container cette fois avec :  
`docker run -d -p 5001:5000 microblog`
  - Une deuxième instance de l'app est maintenant en fonctionnement et accessible à l'adresse `localhost:5001`

# Docker Hub

- Avec docker login, docker tag et docker push, poussez l'image microblog sur le Docker Hub. Créez un compte sur le Docker Hub le cas échéant.

## Améliorer le Dockerfile

### Une image plus simple

- A l'aide de l'image python:3.9-alpine et en remplaçant les instructions nécessaires (pas besoin d'installer python3-pip car ce programme est désormais inclus dans l'image de base), repackagez l'app microblog en une image taggée microblog:slim ou microblog:light. Comparez la taille entre les deux images ainsi construites.

### Faire varier la configuration en fonction de l'environnement

Le serveur de développement Flask est bien pratique pour debugger en situation de développement, mais n'est pas adapté à la

production. Nous pourrions créer deux images pour les deux situations mais ce serait aller contre l'impératif DevOps de rapprochement du dev et de la prod.

Pour démarrer l'application, nous avons fait appel à un script de boot `boot.sh` avec à l'intérieur :

```
#!/bin/bash

# ...

set -e

if [ "$CONTEXT" = 'DEV' ]; then
    echo "Running Development Server"
    FLASK_ENV=development exec flask
run -h 0.0.0.0
else
    echo "Running Production Server"
    exec gunicorn -b :5000 --access-
logfile - --error-logfile -
app_name:app
fi
```

- Déclarez maintenant dans le Dockerfile la variable d'environnement `CONTEXT` avec comme valeur par défaut `PROD`.

- Construisez l'image avec `build`.
- Puis, grâce aux bons arguments allant avec `docker run`, lancez une instance de l'app en configuration PROD et une instance en environnement DEV (joignables sur deux ports différents).
- Avec `docker ps` ou en lisant les logs, vérifiez qu'il existe bien une différence dans le programme lancé.

## Exposer le port

- Ajoutons l'instruction `EXPOSE 5000` pour indiquer à Docker que cette app est censée être accédée via son port 5000.
- NB : Publier le port grâce à l'option `-p port_de_l-hote:port_du_container` reste nécessaire, l'instruction `EXPOSE` n'est là qu'à titre de documentation de l'image.

## Dockerfile amélioré

## L'instruction `HEALTHCHECK`

`HEALTHCHECK` permet de vérifier si l'app contenue

dans un conteneur est en bonne santé.

- Dans un nouveau dossier ou répertoire, créez un fichier Dockerfile dont le contenu est le suivant :

```
FROM python:alpine

RUN apk add curl
RUN pip install flask

ADD /app.py /app/app.py
WORKDIR /app
EXPOSE 5000

HEALTHCHECK CMD curl --fail
http://localhost:5000/health || exit 1

CMD python app.py
```

- Créez aussi un fichier app.py avec ce contenu :

```
from flask import Flask

healthy = True

app = Flask(__name__)

@app.route('/health')
```

```

def health():
    global healthy

    if healthy:
        return 'OK', 200
    else:
        return 'NOT OK', 500

@app.route('/kill')
def kill():
    global healthy
    healthy = False
    return 'You have killed your app.', 200

if __name__ == "__main__":
    app.run(host="0.0.0.0")

```

- Observez bien le code Python et la ligne HEALTHCHECK du Dockerfile puis lancez l'app. A l'aide de docker ps, relevez où Docker indique la santé de votre app.
- Visitez l'URL /kill de votre app dans un navigateur. Refaites docker ps. Que s'est-il passé

?

- (*Facultatif*) Rajoutez une instruction HEALTHCHECK au Dockerfile de notre app microblog.
- 

## ***Facultatif : construire une image “à la main”***

Avec docker commit, trouvons comment ajouter une couche à une image existante. La commande docker diff peut aussi être utile.

## ***Facultatif : Décorner une image***

Une image est composée de plusieurs layers empilés entre eux par le Docker Engine et de métadonnées.

- Affichez la liste des images présentes dans votre Docker Engine.
- Inspectez la dernière image que vous venez de créer (docker image --help pour trouver la commande)
- Observez l'historique de construction de l'image avec docker image history <image>
- Visitons **en root** (sudo su) le dossier /var/lib

/docker/ sur l'hôte. En particulier,  
image/overlay2/layerdb/sha256/ :

- On y trouve une sorte de base de données de tous les layers d'images avec leurs ancêtres.
  - Il s'agit d'une arborescence.
  - Vous pouvez aussi utiliser la commande docker save votre\_image -o image.tar, et utiliser tar -C image\_decompressée/ -xvf image.tar pour décompresser une image Docker puis explorer les différents layers de l'image.
  - Pour explorer la hiérarchie des images vous pouvez installer <https://github.com/wagoodman/dive>
- 

## ***Facultatif : un Registry privé***

- En récupérant [la commande indiquée dans la doc officielle](#), créez votre propre registry.
- Puis trouvez comment y pousser une image dessus.
- Enfin, supprimez votre image en local et récupérez-la depuis votre registry.

## **Facultatif : Faire parler la vache**

Créons un nouveau Dockerfile qui permet de faire dire des choses à une vache grâce à la commande cowsay. Le but est de faire fonctionner notre programme dans un conteneur à partir de commandes de type :

- `docker run --rm cowsay Coucou !`
- `docker run --rm cowsay -f stegosaurus Yo !`
- `docker run --rm cowsay -f elephant-in-snake Un éléphant dans un boa.`
- Doit-on utiliser la commande ENTRYPPOINT ou la commande CMD ? Se référer au [manuel de référence sur les Dockerfiles](#) si besoin.
- Pour information, cowsay s'installe dans `/usr/games/cowsay`.
- La liste des options (incontournables) de cowsay se trouve ici : <https://debian-facile.org/doc:jeux:cowsay>
- L'instruction ENTRYPPOINT et la gestion des entrées-sorties des programmes dans les

Dockerfiles peut être un peu capricieuse et il faut parfois avoir de bonnes notions de Bash et de Linux pour comprendre (et bien lire la documentation Docker).

- On utilise parfois des conteneurs juste pour qu'ils s'exécutent une fois (pour récupérer le résultat dans la console, ou générer des fichiers). On utilise alors l'option `--rm` pour les supprimer dès qu'ils s'arrêtent.

## ***Facultatif : Un multi-stage build***

Transformez le Dockerfile de l'app dnmonster située à l'adresse suivante pour réaliser un multi-stage build afin d'obtenir l'image finale la plus légère possible : <https://github.com/amouat/dnmonster/>

La documentation pour les multi-stage builds est à cette adresse : <https://docs.docker.com/develop/develop-images/multistage-build/>

## **3 - Volumes et réseaux**

### **Cycle de vie d'un conteneur**

- Un conteneur a un cycle de vie très court: il doit

pouvoir être créé et supprimé rapidement même en contexte de production.

Conséquences :

- On a besoin de mécanismes d'autoconfiguration, en particulier réseau car les IP des différents conteneur changent tout le temps.
- On ne peut pas garder les données persistantes dans le conteneur.

Solutions :

- Des réseaux dynamiques par défaut automatiques (DHCP mais surtout DNS automatiques)
  - Des volumes (partagés ou non, distribués ou non) montés dans les conteneurs
- 

## Réseau

### Gestion des ports réseaux (*port mapping*)

- L'instruction EXPOSE dans le Dockerfile informe Docker que le conteneur écoute sur les ports réseau au lancement. L'instruction EXPOSE **ne publie pas les ports**. C'est une sorte de **documentation entre la personne qui construit les images et la personne qui lance le**

## **conteneur à propos des ports que l'on souhaite publier.**

- Par défaut les conteneurs n'ouvrent donc pas de port même s'ils sont déclarés avec EXPOSE dans le Dockerfile.
  - Pour publier un port au lancement d'un conteneur, c'est l'option `-p <port_host>:<port_guest>` de `docker run`.
  - Instruction `port` : d'un compose file.
- 

## **Bridge et overlay**

- Un réseau bridge est une façon de créer un pont entre deux carte réseaux pour construire un réseau à partir de deux.
- Par défaut les réseaux docker fonctionne en bridge (le réseau de chaque conteneur est bridgé à un réseau virtuel docker)
- par défaut les adresses sont en 172.0.0.0/8, typiquement chaque hôte définit le bloc d'IP 172.17.0.0/16 configuré avec DHCP.
- Un réseau overlay est un réseau virtuel privé déployé par dessus un réseau existant

(typiquement public). Pour par exemple faire un cloud multi-datacenters.

---

## Le réseau Docker est très automatique

- Serveur DNS et DHCP intégré dans le “user-defined network” (c'est une solution IPAM)
- Donne un nom de domaine automatique à chaque conteneur.
- Mais ne pas avoir peur d'aller voir comment on perçoit le réseau de l'intérieur. Nécessaire pour bien contrôler le réseau.
- `ingress` : un loadbalancer automatiquement connecté aux nœuds d'un Swarm. Voir la [doc sur les réseaux overlay](#).

## Lier des conteneurs

- Aujourd'hui il faut utiliser un réseau dédié créé par l'utilisateur (“user-defined bridge network”)
- avec l'option `--network` de `docker run`
- avec l'instruction `networks` : dans un `docker compose`
- On peut aussi créer un lien entre des conteneurs

- avec l'option `--link` de `docker run`
- avec l'instruction `link` : dans un `docker compose`
- MAIS cette fonctionnalité est **obsolète** et déconseillée

## Plugins réseaux

Il existe :

- les réseaux par défaut de Docker
  - plusieurs autres solutions spécifiques de réseau disponibles pour des questions de performance et de sécurité
  - Ex. : **Weave Net** pour un cluster Docker Swarm
  - fournit une autoconfiguration très simple
  - de la sécurité
  - un DNS qui permet de simuler la découverte de service
  - Du multicast UDP
- 

## Volumes

### Les volumes Docker via la sous-commande `volume`

- docker volume ls
- docker volume inspect
- docker volume prune
- docker volume create
- docker volume rm

## Bind mounting

Lorsqu'un répertoire hôte spécifique est utilisé dans un volume (la syntaxe `-v HOST_DIR:CONTAINER_DIR`), elle est souvent appelée **bind mounting** ("montage lié"). C'est quelque peu trompeur, car tous les volumes sont techniquement "bind mounted". La particularité, c'est que le point de montage sur l'hôte est explicite plutôt que caché dans un répertoire appartenant à Docker.

Exemple :

```
# Sur l'hôte
docker run -it -v /home/user
/app/config.conf:/config/main.conf:ro
-v /home/user/app/data:/data ubuntu
/bin/bash
```

```
# Dans le conteneur  
cd /data/  
touch testfile  
exit  
  
# Sur l'hôte  
ls /home/user/app/data:
```

## Volumes nommés

- L'autre technique est de créer d'abord un volume nommé avec :  
docker volume create  
mon\_volume  
docker run -d -v  
mon\_volume:/data redis
- 

## L'instruction VOLUME dans un Dockerfile

L'instruction VOLUME dans un Dockerfile permet de désigner les volumes qui devront être créés lors du lancement du conteneur. On précise ensuite avec l'option -v de docker run à quoi connecter ces volumes. Si on ne le précise pas, Docker crée quand même un volume Docker au nom généré aléatoirement, un volume "caché".

## Partager des données avec un volume

- Pour partager des données on peut monter le même volume dans plusieurs conteneurs.
- Pour lancer un conteneur avec les volumes d'un autre conteneur déjà montés on peut utiliser `--volumes-from <container>`
- On peut aussi créer le volume à l'avance et l'attacher après coup à un conteneur.
- Par défaut le driver de volume est local c'est-à-dire qu'un dossier est créé sur le disque de l'hôte.

```
docker volume create --driver local \
    --opt type=btrfs \
    --opt device=/dev/sda2 \
    monVolume
```

## Plugins de volumes

On peut utiliser d'autres systèmes de stockage en installant de nouveau plugins de driver de volume.

Par exemple, le plugin vieux/sshfs permet de piloter un volume distant via SSH.

Exemples:

- SSHFS (utilisation d'un dossier distant via SSH)
- NFS (protocole NFS)

- BeeGFS (système de fichier distribué générique)
- Amazon EBS (vendor specific)
- etc.

```
docker volume create -d vieux/sshfs -o  
sshcmd=<sshcmd> -o allow_other  
sshvolume  
docker run -p 8080:8080 -v  
sshvolume:/path/to/folder --name test  
someimage
```

---

Ou via docker-compose :

```
volumes:  
  sshfsdata:  
    driver: vieux/sshfs:latest  
    driver_opts:  
      sshcmd:  
      "username@server:/location/on/the  
      /server"  
      allow_other: ""
```

---

## Permissions

- Un volume est créé avec les permissions du dossier préexistant.

```
FROM debian

RUN groupadd -r graphite && useradd -r
-g graphite graphite
RUN mkdir -p /data/graphite && chown
-R graphite:graphite /data/graphite
VOLUME /data/graphite
USER graphite
CMD ["echo", "Data container for
graphite"]
```

## Backups de volumes

- Pour effectuer un backup la méthode recommandée est d'utiliser un conteneur supplémentaire dédié
- qui accède au volume avec `--volume-from`
- qui est identique aux autres et donc normalement avec les mêmes UID/GID/permissions.

## TP 3 - Réseaux

### Portainer

Si vous aviez déjà créé le conteneur Portainer, vous pouvez le relancer en faisant `docker start`

portainer, sinon créez-le comme suit :

```
docker volume create portainer_data
docker run --detach --name portainer \
-p 9000:9000 \
-v portainer_data:/data \
-v /var/run/docker.sock:/var/run
/docker.sock \
portainer/portainer-ce
```

## Partie 1 : Docker networking

Pour expérimenter avec le réseau, nous allons lancer une petite application nodejs d'exemple (moby-counter) qui fonctionne avec une file (*queue*) redis (comme une base de données mais pour stocker des paires clé/valeur simples).

Récupérons les images depuis Docker Hub:

- docker image pull redis:alpine
- docker image pull russmckendrick/moby-counter
- Lancez la commande ip -br a pour lister vos interfaces réseau

Pour connecter les deux applications créons un réseau manuellement:

- docker network create moby-network

Docker implémente ces réseaux virtuels en créant des interfaces. Lancez la commande `ip -br` à de nouveau et comparez. Qu'est-ce qui a changé ?

Maintenant, lançons les deux applications en utilisant notre réseau :

- docker run -d --name redis --network <r  seau> redis:alpine
- docker run -d --name moby-counter --network <r  seau> -p 80:80 russmckendrick/moby-counter
- Visitez la page de notre application. Qu'en pensez vous ? Moby est le nom de la mascotte Docker . Faites un motif en cliquant. .

Comment notre application se connecte-t-elle au conteneur redis ? Elle utilise ces instructions JS dans son fichier `server.js`:

```
var port = opts.redis_port ||  
process.env.USE_REDIS_PORT || 6379;  
var host = opts.redis_host ||  
process.env.USE_REDIS_HOST || "redis";
```

En r  sum   par d  faut, notre application se

connecte sur l'hôte `redis` avec le port 6379

Explorons un peu notre réseau Docker.

- Exécutez (`docker exec`) la commande `ping -c 3 redis` à l'intérieur de notre conteneur applicatif (`moby-counter` donc). Quelle est l'adresse IP affichée ?

```
docker exec moby-counter ping -c3  
redis
```

- De même, affichez le contenu des fichiers `/etc/hosts` du conteneur (c'est la commande `cat` couplée avec `docker exec`). Nous constatons que Docker a automatiquement configuré l'IP externe **du conteneur dans lequel on est** avec l'identifiant du conteneur.
- Qu'est-ce que Docker fournit qui permet que ce ping fonctionne ?
- Pour s'en assurer, interrogeons le serveur DNS de notre réseau `moby-network` en lançant la commande `nslookup redis` grâce à `docker exec` :  
`docker exec moby-counter nslookup redis`
- Créez un deuxième réseau `moby-network2`

- Créez une deuxième instance de l'application dans ce réseau : `docker run -d --name moby-counter2 --network moby-network2 -p 9090:80 russmckendrick/moby-counter`
- Lorsque vous pingez `redis` depuis cette nouvelle instance `moby-counter2`, qu'obtenez-vous ? Pourquoi ?

Vous ne pouvez pas avoir deux conteneurs avec les mêmes noms, comme nous l'avons déjà découvert. Par contre, notre deuxième réseau fonctionne complètement isolé de notre premier réseau, ce qui signifie que nous pouvons toujours utiliser le nom de domaine `redis`. Pour ce faire, nous devons spécifier l'option `--network-alias` :

- Créons un deuxième `redis` avec le même domaine :  
`docker run -d --name redis2 --network moby-network2 --network-alias redis redis:alpine`
- Lorsque vous pingez `redis` depuis cette nouvelle instance de l'application, quelle IP obtenez-vous ?
- Lancez `nslookup redis` dans le conteneur `moby-counter2` pour tester la résolution de DNS.

- Vous pouvez retrouver la configuration du réseau et les conteneurs qui lui sont reliés avec docker network inspect moby-network2. Notez la section IPAM (IP Address Management).
  - Arrêtons nos conteneurs : docker stop moby-counter2 redis2.
  - Pour faire rapidement le ménage des conteneurs arrêtés lancez docker container prune.
  - De même docker network prune permet de faire le ménage des réseaux qui ne sont plus utilisés par aucun conteneur.
- 

## TP 3bis - Volumes

### Portainer

Si vous aviez déjà créé le conteneur Portainer, vous pouvez le relancer en faisant docker start portainer, sinon créez-le comme suit :

```
docker volume create portainer_data
docker run --detach --name portainer \
-p 9000:9000 \
-v portainer_data:/data \
-v /var/run/docker.sock:/var/run
```

```
/docker.sock \  
portainer/portainer-ce
```

- Remarque sur la commande précédente : pour que Portainer puisse fonctionner et contrôler Docker lui-même depuis l'intérieur du conteneur il est nécessaire de lui donner accès au socket de l'API Docker de l'hôte grâce au paramètre `--volume` ci-dessus.
- Visitez ensuite la page <http://localhost:9000> pour accéder à l'interface.
- Créez votre user admin avec le formulaire.
- Explorez l'interface de Portainer.

## Partie 2 : Volumes Docker

### Introduction aux volumes

- Pour comprendre ce qu'est un volume, lançons un conteneur en mode interactif et associons-y le dossier `/tmp/dossier-hôte` de l'hôte au dossier `/dossier-conteneur` sur le conteneur :

```
docker run -it -v /tmp/dossier-  
hôte:/dossier-conteneur ubuntu  
/bin/bash
```

- Dans le conteneur, navigons dans ce dossier et créons-y un fichier :

```
cd /dossier-conteneur/  
touch test-depuis-conteneur
```

- Sortons ensuite de ce conteneur avec la commande `exit`
- Après être sorti·e du conteneur, listons le contenu du dossier **sur l'hôte** avec la commande suivante ou avec le navigateur de fichiers d'Ubuntu :

Le fichier `test-depuis-conteneur` a été crée par le conteneur au dossier que l'on avait connecté grâce à `-v /tmp/dossier-hote:/dossier-conteneur`

- Tentez de créer un fichier **depuis l'hôte** dans ce dossier. Que se passe-t-il ? Que faut-il faire ? Pourquoi ?

## L'app **moby-counter**, **Redis** et les **volumes**

Pour ne pas interférer avec la deuxième partie du TP :

- Stoppez tous les conteneurs redis et moby-counter

avec docker stop ou avec Portainer.

- Supprimez les conteneurs arrêtés avec docker container prune
- Lancez docker volume prune pour faire le ménage de volume éventuellement créés dans les TP précédents
- Lancez aussi docker network prune pour nettoyer les réseaux inutilisés

## Volumes nommés

Lorsqu'un répertoire hôte spécifique est utilisé dans un volume (la syntaxe -v HOST\_DIR:CONTAINER\_DIR), elle est souvent appelée **bind mounting**. C'est quelque peu trompeur, car tous les volumes sont techniquement "bind mounted". La différence, c'est que le point de montage est explicite plutôt que caché dans un répertoire géré par Docker.

Nous allons recréer un conteneur avec cette fois-ci un volume nommé.

En effet, la bonne façon de créer des volumes consiste à les créer manuellement dans un premier temps (volumes nommés), puis d'y associer un

```
conteneur : docker volume create  
redis_data.
```

- Lancez docker volume inspect redis\_data.
- Créez le conteneur moby-counter à l'intérieur :

```
docker network create moby-network  
docker run -d --network moby-network  
--name moby-counter -p 8000:80  
russmckendrick/moby-counter
```

- Puis, à l'aide de la documentation disponible sur le Docker Hub, trouvons le point de montage où connecter un conteneur Redis pour que ses données persistent à la suppression du conteneur.
- créons le conteneur Redis connecté à notre volume nommé (il faut remplacer \_\_VOLUME\_\_ : \_\_POINT\_DE\_MONTAGE\_\_ par les bonnes informations) :

```
docker run -d --name redis --network  
moby-network --volume  
__VOLUME__ : __POINT_DE_MONTAGE__ redis
```

## Récupérer un volume d'un conteneur supprimé

- supprimez le conteneur redis : docker stop redis puis docker rm redis
- recréons le conteneur redis, mais **par erreur nous allons oublier de le connecter à un volume à la création :**

```
docker run -d --name redis --network
moby-network redis
docker run -d --name moby-counter
--network moby-network -p 8000:80
russmckendrick/moby-counter
```

- Visitez votre application dans le navigateur. **Faites un motif reconnaissable en cliquant.**
- supprimez le nouveau conteneur redis : docker stop redis puis docker rm redis
- Visitez votre application dans le navigateur. Elle est maintenant déconnectée de son backend.
- Avons-nous vraiment perdu les données de notre conteneur précédent ? Non ! Le Dockerfile pour l'image officielle Redis ressemble à ça :

```
FROM alpine:3.5

RUN addgroup -S redis && adduser -S -G
redis redis
```

```
RUN apk add --no-cache 'su-exec>=0.2'
ENV REDIS_VERSION 3.0.7
ENV REDIS_DOWNLOAD_URL
http://download.redis.io/releases
/redis-3.0.7.tar.gz
ENV REDIS_DOWNLOAD_SHA
e56b4b7e033ae8dbf311f9191cf6fdf3ae974d
RUN set -x \
    && apk add --no-cache --virtual \
.build-deps \
    gcc \
    linux-headers \
    make \
    musl-dev \
    tar \
    && wget -O redis.tar.gz
"$REDIS_DOWNLOAD_URL" \
    && echo "$REDIS_DOWNLOAD_SHA"
\*redis.tar.gz" | sha1sum -c - \
    && mkdir -p /usr/src/redis \
    && tar -xzf redis.tar.gz -C /usr/src \
/redis --strip-components=1 \
    && rm redis.tar.gz \
    && make -C /usr/src/redis \
    && make -C /usr/src/redis install \
```

```
&& rm -r /usr/src/redis \
&& apk del .build-deps

RUN mkdir /data && chown redis:redis
/data

VOLUME /data

WORKDIR /data

COPY docker-entrypoint.sh /usr/local
/bin/

RUN ln -s usr/local/bin/docker-
entrypoint.sh /entrypoint.sh #
backwards compat

ENTRYPOINT ["docker-entrypoint.sh"]

EXPOSE 6379

CMD [ "redis-server" ]
```

Notez que, vers la fin du fichier, il y a une instruction VOLUME ; cela signifie que lorsque notre conteneur a été lancé, un volume “caché” a effectivement été créé par Docker.

Beaucoup de conteneurs Docker sont des applications *stateful*, c'est-à-dire qui stockent des données. Automatiquement ces conteneurs créent des volumes anonymes en arrière plan qu'il faut ensuite supprimer manuellement (avec rm ou

prune).

- Inspectez la liste des volumes (par exemple avec Portainer) pour retrouver l'identifiant du volume caché. Normalement il devrait y avoir un volume `portainer_data` (si vous utilisez Portainer) et un volume anonyme avec un hash.
- Créez un nouveau conteneur redis en le rattachant au volume redis “caché” que vous avez retrouvé (en copiant l'id du volume anonyme) :  
`docker container run -d --name redis -v <volume_id>/_data:/data --network moby-network redis:alpine`
- Visitez la page de l'application. Normalement un motif de logos *moby* d'une précédente session devrait s'afficher (après un délai pouvant aller jusqu'à plusieurs minutes)
- Affichez le contenu du volume avec la commande :  
`docker exec redis ls -lha /data`

## **Supprimer les volumes et réseaux**

- Pour nettoyer tout ce travail, arrêtez d'abord les différents conteneurs `redis` et `moby-counter`.
- Lancez la fonction `prune` pour les conteneurs

d'abord, puis pour les réseaux, et enfin pour les volumes.

Comme les réseaux et volumes n'étaient plus attachés à des conteneurs en fonctionnement, ils ont été supprimés.

***Généralement, il faut faire beaucoup plus attention au prune de volumes (données à perdre) qu'au prune de conteneurs (rien à perdre car immutable et en général dans le registry).***

## Facultatif : utiliser VOLUME avec microblog

- Clonons le repo microblog ailleurs :

```
git clone https://github.com/uptime-formation/microblog/ --branch tp2-dockerfile microblog-volume
```

- Ouvrons ça avec VSCode : code microblog-volume
- Lire le Dockerfile de l'application microblog.

Un volume Docker apparaît comme un dossier à l'intérieur du conteneur. Nous allons faire apparaître le volume Docker comme un dossier à

l'emplacement /data sur le conteneur.

- Pour que l'app Python soit au courant de l'emplacement de la base de données, ajoutez à votre Dockerfile une variable d'environnement DATABASE\_URL ainsi (cette variable est lue par le programme Python) :

```
ENV DATABASE_URL=sqlite:///data  
/app.db
```

Cela indique que l'on va demander à Python d'utiliser SQLite pour stocker la base de données comme un unique fichier au format .db (SQLite) dans un dossier accessible par le conteneur. On a en fait indiqué à l'app Python que chemin de la base de données est : /data/app.db

- Ajouter au Dockerfile une instruction VOLUME pour stocker la base de données SQLite de l'application.
- Créez un volume nommé appelé microblog\_db, et lancez un conteneur l'utilisant, créez un compte et écrivez un message.
- Vérifier que le volume nommé est bien utilisé en branchant un deuxième conteneur microblog utilisant le même volume nommé.

---

## ( facultatif ) Deux conteneurs Redis sur un seul volume

- Créez un réseau `moby-network2` et ajoutez un deuxième conteneur `redis2` qui va partager les même données que le premier :
- situé à l'intérieur du nouveau réseau (`moby-network2`) comme à la partie précédent.
- utilisant l'option `--network-alias redis` pour pouvoir être joignable par `moby-counter2` (que nous n'avons pas encore créé).
- partageant le volume de données du premier (cf. cours)
- monté en `read-only` (:`ro` après le paramètre de la question précédente)

Le `read-only` est nécessaire pour que les deux Redis n'écrivent pas de façon contradictoire dans la base de valeurs.

- Ajoutez une deuxième instance de l'application dans le deuxième réseau connectée à ce nouveau Redis.
- Visitez la deuxième application : vous devriez voir également le motif de `moby` apparaître.

## **Facultatif : Packagez votre propre app**

Vous possédez tous les ingrédients pour packager l'app de votre choix désormais ! Récupérez une image de base, basez-vous sur un Dockerfile existant s'il vous inspire, et lancez-vous !

## **4 - Créez une application multiconteneur**

### **Docker Compose**

- Nous avons pu constater que lancer plusieurs conteneurs liés avec leur mapping réseau et les volumes liés implique des commandes assez lourdes. Cela devient ingérable si l'on a beaucoup d'applications microservice avec des réseaux et des volumes spécifiques.
- Pour faciliter tout cela et dans l'optique d'**Infrastructure as Code**, Docker introduit un outil nommé **docker-compose** qui permet de décrire de applications multiconteneurs grâce à des fichiers **YAML**.
- Pour bien comprendre qu'il ne s'agit que de convertir des options de commande Docker en YAML, un site vous permet de convertir une

commande docker run en fichier Docker

Compose : <https://www.composerize.com/>

- Le “langage” de Docker Compose : [la documentation du langage \(DSL\) des compose-files](#) est essentielle.
- 

## A quoi ça ressemble, YAML ?

- marché:

    lieu: Marché de la Défense

    jour: jeudi

    horaire:

        unité: "heure"

        min: 12

        max: 20

    fruits:

        - nom: pomme

            couleur: "verte"

            pesticide: avec

        - nom: poires

            couleur: jaune

            pesticide: sans

    légumes:

        - courgettes

- salade
  - potiron
- 

## Syntaxe

- Alignement ! (**2 espaces !!**)
  - ALIGNEMENT !! (comme en python)
  - **ALIGNEMENT !!!** (le défaut du YAML, pas de correcteur syntaxique automatique, c'est bête mais vous y perdrez forcément quelques heures !)
  - des listes (tirets)
  - des paires **clé: valeur**
  - Un peu comme du JSON, avec cette grosse différence que le JSON se fiche de l'alignement et met des accolades et des points-virgules
  - **les extensions Docker et YAML dans VSCode vous aident à repérer des erreurs**
- 

## Un exemple de fichier Docker Compose

```
services:  
  postgres:  
    image: postgres:10  
    environment:
```

```
    POSTGRES_USER: rails_user
    POSTGRES_PASSWORD:
rails_password
    POSTGRES_DB: rails_db
networks:
    - back_end

redis:
    image: redis:3.2-alpine
networks:
    - back_end

rails:
    build: .
depends_on:
    - postgres
    - redis
environment:
    DATABASE_URL:
"postgres://rails_user:rails_password@
/rails_db"
    REDIS_HOST: "redis:6379"
networks:
    - front_end
    - back_end
```

```
volumes:
  - .:/app

nginx:
  image: nginx:latest
  networks:
    - front_end
  ports:
    - 3000:80
  volumes:
    - ./nginx.conf:/etc/nginx/conf.d
/default.conf:ro

networks:
  front_end:
  back_end:
```

Un deuxième exemple :

```
services:
  wordpress:
    depends_on:
      - mysqlpourwordpress
    environment:
      -
      "WORDPRESS_DB_HOST=mysqlpourwordpress":
      -
```

```
WORDPRESS_DB_PASSWORD=monwordpress
  - WORDPRESS_DB_USER=wordpress
networks:
  - wordpress
ports:
  - "80:80"
image: wordpress
volumes:
  - wordpress_config:/var
/www/html/
mysqlpourwordpress:
  image: "mysql:5.7"
  environment:
    -
  MYSQL_ROOT_PASSWORD=motdepasseroot
    - MYSQL_DATABASE=wordpress
    - MYSQL_USER=wordpress
    - MYSQL_PASSWORD=monwordpress
networks:
  - wordpress
volumes:
  - wordpress_data:/var/lib/mysql/
networks:
```

```
wordpress:
```

```
volumes:
```

```
  wordpress_config:
```

```
  wordpress_data:
```

## Le workflow de Docker Compose

Les commandes suivantes sont couramment utilisées lorsque vous travaillez avec Compose. La plupart se passent d'explications et ont des équivalents Docker directs, mais il vaut la peine d'en être conscient·e :

- `up` démarre tous les conteneurs définis dans le fichier `compose` et agrège la sortie des logs.  
Normalement, vous voudrez utiliser l'argument `-d` pour exécuter Compose en arrière-plan.
- `build` reconstruit toutes les images créées à partir de Dockerfiles. La commande `up` ne construira pas une image à moins qu'elle n'existe pas, donc utilisez cette commande à chaque fois que vous avez besoin de mettre à jour une image (quand vous avez édité un Dockerfile). On peut aussi faire

```
docker-compose up --build
```

- `ps` fournit des informations sur le statut des conteneurs gérés par Compose.
- `run` fait tourner un conteneur pour exécuter une commande unique. Cela aura aussi pour effet de faire tourner tout conteneur décrit dans `depends_on`, à moins que l'argument `--no-deps` ne soit donné.
- `logs` affiche les logs. De façon générale la sortie des logs est colorée et agrégée pour les conteneurs gérés par Compose.
- `stop` arrête les conteneurs sans les enlever.
- `rm` enlève les conteneurs à l'arrêt. N'oubliez pas d'utiliser l'argument `-v` pour supprimer tous les volumes gérés par Docker.
- `down` détruit tous les conteneurs définis dans le fichier Compose, ainsi que les réseaux

## Le “langage” de Docker Compose

- N'hésitez pas à passer du temps à explorer les options et commandes de `docker-compose`.
- [La documentation du langage \(DSL\) des compose-](#)

files est essentielle.

- il est aussi possible d'utiliser des variables d'environnement dans Docker Compose : se référer au [mode d'emploi](#) pour les subtilités de fonctionnement
- 

## **Visualisation des applications microservice complexes**

- Certaines applications microservice peuvent avoir potentiellement des dizaines de petits conteneurs spécialisés. Le service devient alors difficile à lire dans le compose file.
- Il est possible de visualiser l'architecture d'un fichier Docker Compose en utilisant [docker-compose-viz](#)
- Cet outil peut être utilisé dans un cadre d'intégration continue pour produire automatiquement la documentation pour une image en fonction du code.

## **TP 4 - Créer une application multiconteneur**

### **Articuler deux images avec Docker**

# compose

- Installez docker-compose avec `sudo apt install docker-compose`.
- Pour vous faciliter la vie et si ce n'est pas déjà le cas, ajoutez le plugin *autocomplete* pour Docker et Docker Compose à bash en copiant les commandes suivantes :

```
sudo apt update
sudo apt install bash-completion curl
sudo curl -L
https://raw.githubusercontent.com
/docker/compose/1.24.1/contrib
/completion/bash/docker-compose -o
/etc/bash_completion.d/docker-compose
```

## **identidock : une application Flask qui se connecte à redis**

- Démarrez un nouveau projet dans VSCode (créez un dossier appelé `identidock` et chargez-le avec la fonction *Add folder to workspace*)
- Dans un sous-dossier `app`, ajoutez une petite application python en créant ce fichier

identidock.py :

```
from flask import Flask, Response,
request, abort
import requests
import hashlib
import redis
import os
import logging

LOGLEVEL = os.environ.get('LOGLEVEL',
'INFO').upper()
logging.basicConfig(level=LOGLEVEL)

app = Flask(__name__)
cache =
redis.StrictRedis(host='redis',
port=6379, db=0)
salt = "UNIQUE_SALT"
default_name = 'toi'

@app.route('/', methods=['GET',
'POST'])
def mainpage():

    name = default_name
```

```
if request.method == 'POST':
    name = request.form['name']

    salted_name = salt + name
    name_hash =
        hashlib.sha256(salted_name.encode()).h
        header = '<html><head>
<title>Identidock</title></head>
<body>'

        body = '''<form method="POST">
            Salut <input
type="text" name="name" value="{0}"> !
            <input type="submit"
value="submit">
        </form>
        <p>Tu ressembles à ça
        :
        
        '''.format(name,
name_hash)
        footer = '</body></html>'

    return header + body + footer
```

```
@app.route('/monster/<name>')
def get_identicon(name):
    found_in_cache = False

    try:
        image = cache.get(name)
        redis_unreachable = False
        if image is not None:
            found_in_cache = True
            logging.info("Image trouvée dans le cache")
    except:
        redis_unreachable = True
        logging.warning("Cache redis injoignable")

    if not found_in_cache:
        logging.info("Image non trouvée dans le cache")
        try:
            r =
            requests.get('http://dnmonster:8080
/monster/' + name + '?size=80')
            image = r.content
            logging.info("Image
```

```

générée grâce au service dnmonster")

        if not redis_unreachable:
            cache.set(name, image)
            logging.info("Image
enregistrée dans le cache redis")
        except:
            logging.critical("Le
service dnmonster est injoignable !")
            abort(503)

    return Response(image,
mimetype='image/png')

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0',
port=5000)

```

- uWSGI est un serveur python de production très adapté pour servir notre serveur intégré Flask, nous allons l'utiliser.
- Dockerisons maintenant cette nouvelle application avec le Dockerfile suivant :

```
FROM python:3.7
```

```
RUN groupadd -r uwsgi && useradd -r -g uwsgi uwsgi
RUN pip install Flask uWSGI requests redis
WORKDIR /app
COPY app/identidock.py /app

EXPOSE 5000 9191
USER uwsgi
CMD ["uwsgi", "--http",
"0.0.0.0:5000", "--wsgi-file",
"/app/identidock.py", \
"--callable", "app", "--stats",
"0.0.0.0:9191"]
```

- Observons le code du Dockerfile ensemble s'il n'est pas clair pour vous. Juste avant de lancer l'application, nous avons changé d'utilisateur avec l'instruction USER, pourquoi ?.
- Construire l'application, pour l'instant avec docker build, la lancer et vérifier avec docker exec, whoami et id l'utilisateur avec lequel tourne le conteneur.

## Le fichier Docker Compose

- A la racine de notre projet `identidock` (à côté du `Dockerfile`), créez un fichier de déclaration de notre application appelé `docker-compose.yml` avec à l'intérieur :

```
services:  
  identidock:  
    build: .  
    ports:  
      - "5000:5000"
```

- Plusieurs remarques :
- la première ligne après `services` déclare le conteneur de notre application
- les lignes suivantes permettent de décrire comment lancer notre conteneur
- `build: .` indique que l'image d'origine de notre conteneur est le résultat de la construction d'une image à partir du répertoire courant (équivaut à `docker build -t identidock .`)
- la ligne suivante décrit le mapping de ports entre l'extérieur du conteneur et l'intérieur.
- Lancez le service (pour le moment mono-conteneur) avec `docker-compose up` (cette

commande sous-entend docker-compose build)

- Visitez la page web de l'app.
- Ajoutons maintenant un deuxième conteneur. Nous allons tirer parti d'une image déjà créée qui permet de récupérer une "identicon". Ajoutez à la suite du fichier Compose (**attention aux indentations !**) :

```
dnmonster:  
  image: amouat/dnmonster:1.0
```

Le docker-compose.yml doit pour l'instant ressembler à ça :

```
services:  
  identidock:  
    build: .  
    ports:  
      - "5000:5000"  
  
  dnmonster:  
    image: amouat/dnmonster:1.0
```

Enfin, nous déclarons aussi un réseau appelé identinet pour y mettre les deux conteneurs de notre application.

- Il faut déclarer ce réseau à la fin du fichier (notez

que l'on doit spécifier le driver réseau) :

```
networks:  
  identinet:  
    driver: bridge
```

- Il faut aussi mettre nos deux services `identidock` et `dnmonster` sur le même réseau en ajoutant **deux fois** ce bout de code où c'est nécessaire (**attention aux indentations !**) :
- Ajoutons également un conteneur `redis` (**attention aux indentations !**). Cette base de données sert à mettre en cache les images et à ne pas les recalculer à chaque fois.

```
redis:  
  image: redis  
  networks:  
    - identinet
```

`docker-compose.yml` final :

```
services:  
  identidock:  
    build: .  
    ports:  
      - "5000:5000"  
      - "9191:9191" # port pour les
```

```
stats
  networks:
    - identinet

dnmonster:
  image: amouat/dnmonster:1.0
  networks:
    - identinet

redis:
  image: redis
  networks:
    - identinet

networks:
  identinet:
    driver: bridge
```

- Lancez l'application et vérifiez que le cache fonctionne en cherchant les messages dans les logs de l'application.
- N'hésitez pas à passer du temps à explorer les options et commandes de docker-compose, ainsi que [la documentation officielle du langage des Compose files](#).

## **Le Hot Code Reloading (rechargement du code à chaud)**

En s'inspirant des exercices sur les volumes (TP3) et du fichier boot.sh de l'app microblog (TP2), modifions le docker-compose.yml pour y inclure des instructions pour lancer le serveur python en mode debug : la modification du code source devrait immédiatement être répercutée dans les logs d'identidock, et recharger la page devrait nous montrer la nouvelle version du code de l'application.

## **D'autres services**

### **Exercices de *google-fu***

#### **ex: un pad HedgeDoc**

On se propose ici d'essayer de déployer plusieurs services pré-configurés comme Wordpress, Nextcloud, Sentry ou votre logiciel préféré.

- Récupérez (et adaptez si besoin) à partir d'Internet un fichier docker-compose.yml permettant de lancer un pad HedgeDoc ou autre avec sa base de données. Je vous conseille de toujours chercher

**dans la documentation officielle** ou le repository officiel (souvent sur Github) en premier.

- Vérifiez que le service est bien accessible sur le port donné.
- Si besoin, lisez les logs en quête bug et adaptez les variables d'environnement.

## Une stack Elastic

### Centraliser les logs

L'utilité d'Elasticsearch est que, grâce à une configuration très simple de son module Filebeat, nous allons pouvoir centraliser les logs de tous nos conteneurs Docker. Pour ce faire, il suffit d'abord de télécharger une configuration de Filebeat prévue à cet effet :

```
curl -L -O  
https://raw.githubusercontent.com  
/elastic/beats/7.10/deploy/docker  
/filebeat.docker.yml
```

Renommons cette configuration et rectifions qui possède ce fichier pour satisfaire une contrainte de sécurité de Filebeat :

```
mv filebeat.docker.yml filebeat.yml
```

```
sudo chown root filebeat.yml  
sudo chmod go-w filebeat.yml
```

Enfin, créons un fichier docker-compose.yml pour lancer une stack Elasticsearch :

```
services:  
  elasticsearch:  
    image:  
      docker.elastic.co/elasticsearch  
      /elasticsearch:7.5.0  
    environment:  
      - discovery.type=single-node  
      - xpack.security.enabled=false  
    networks:  
      - logging-network  
  
  filebeat:  
    image: docker.elastic.co/beats  
    /filebeat:7.5.0  
    user: root  
    depends_on:  
      - elasticsearch  
    volumes:  
      - ./filebeat.yml:/usr/share  
      /filebeat/filebeat.yml:ro
```

```
    - /var/lib/docker/containers:  
/var/lib/docker/containers:ro  
        - /var/run/docker.sock:/var/run  
/docker.sock  
    networks:  
        - logging-network  
environment:  
    - -strict.perms=false  
  
kibana:  
    image: docker.elastic.co/kibana  
/kibana:7.5.0  
    depends_on:  
        - elasticsearch  
    ports:  
        - 5601:5601  
    networks:  
        - logging-network  
  
networks:  
    logging-network:  
        driver: bridge
```

Il suffit ensuite de :

- se rendre sur Kibana (port 5601)

- de configurer l'index en tapant \* dans le champ indiqué, de valider
- et de sélectionner le champ @timestamp, puis de valider.

L'index nécessaire à Kibana est créé, vous pouvez vous rendre dans la partie Discover à gauche (l'icône boussole  ) pour lire vos logs.

Il est temps de faire un petit docker stats pour découvrir l'utilisation du CPU et de la RAM de vos conteneurs !

### ***Facultatif : Ajouter un nœud Elasticsearch***

Puis, à l'aide de la documentation Elasticsearch et/ou en adaptant de bouts de code Docker Compose trouvés sur internet, ajoutez et configurez un nœud Elastic. Toujours à l'aide de la documentation Elasticsearch, vérifiez que ce nouveau nœud communique bien avec le premier.

### ***Facultatif : ajouter une stack ELK à microblog***

Dans la dernière version de l'app microblog, Elasticsearch est utilisé pour fournir une fonctionnalité de recherche puissante dans les

posts de l'app. Avec l'aide du [tutoriel de Miguel Grinberg](#), écrivez le docker-compose.yml qui permet de lancer une stack entière pour microblog. Elle devra contenir un conteneur microblog, un conteneur mysql, un conteneur elasticsearch et un conteneur kibana.

### ***Facultatif : Utiliser Traefik***

Vous pouvez désormais faire [l'exercice 1 du TP7](#) pour configurer un serveur web qui permet d'accéder à vos services via des domaines.

## **5 - Orchestration et clustering**

### **Orchestration**

- Un des intérêts principaux de Docker et des conteneurs en général est de :
- favoriser la modularité et les architectures microservice.
- permettre la scalabilité (mise à l'échelle) des applications en multipliant les conteneurs.
- A partir d'une certaine échelle, il n'est plus question de gérer les serveurs et leurs conteneurs à la main.

Les nœuds d'un cluster sont les machines (serveurs physiques, machines virtuelles, etc.) qui font tourner vos applications (composées de conteneurs).

L'orchestration consiste à automatiser la création et la répartition des conteneurs à travers un cluster de serveurs. Cela peut permettre de :

- déployer de nouvelles versions d'une application progressivement.
  - faire grandir la quantité d'instances de chaque application facilement.
  - voire dans le cas de l'auto-scaling de faire grossir l'application automatiquement en fonction de la demande.
- 

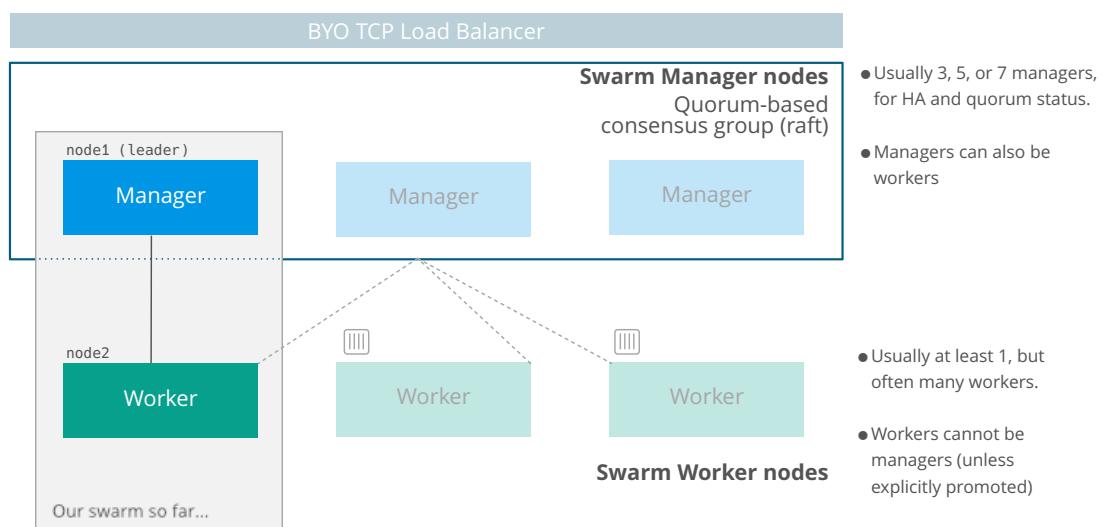
## Docker Swarm

- Swarm est l'**outil de clustering et d'orchestration natif** de Docker (développé par Docker Inc.).
- Il s'intègre très bien avec les autres commandes docker (on a même pas l'impression de faire du clustering).
- Il permet de gérer de très grosses productions Docker.

- Swarm utilise l'API standard du Docker Engine (sur le port 2376) et sa propre API de management Swarm (sur le port 2377).
  - Il a perdu un peu en popularité face à Kubernetes mais c'est très relatif (voir comparaison plus loin).
- 

## Architecture de Docker Swarm

### Docker Enterprise Edition: Swarm Architecture



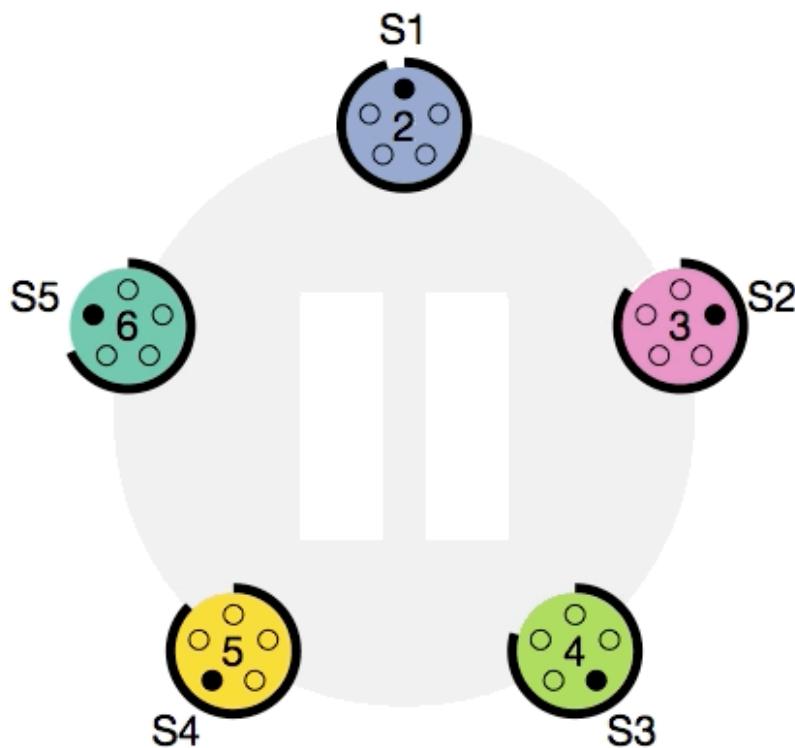
]

- Un ensemble de nœuds de contrôle pour gérer les conteneurs
- Un ensemble de nœuds worker pour faire tourner les conteneurs

- Les nœuds managers sont en fait aussi des workers et font tourner des conteneurs, c'est leur rôles qui varient.

## Consensus entre managers Swarm

- L'algorithme Raft : <http://thesecretlivesofdata.com>



[/raft/](#)

- Pas d'*intelligent balancing* dans Swarm
- l'algorithme de choix est “spread”, c'est-à-dire qu'il répartit au maximum en remplissant tous les nœuds qui répondent aux contraintes données.

## Docker Services et Stacks

- les **services** : la distribution **d'un seul conteneur**

## en plusieurs exemplaires

- les **stacks** : la distribution (en plusieurs exemplaires) **d'un ensemble de conteneurs (app multiconteneurs)** décrits dans un fichier Docker Compose

```
services:  
  web:  
    image: username/repo  
    deploy:  
      replicas: 5  
      resources:  
        limits:  
          cpus: "0.1"  
          memory: 50M  
      restart_policy:  
        condition: on-failure  
    ports:  
      - "4000:80"  
  networks:  
    - webnet  
networks:  
  webnet:
```

- Référence pour les options Swarm de Docker Compose : <https://docs.docker.com/compose>

## [/compose-file/#deploy](#)

- Le mot-clé `deploy` est lié à l'usage de Swarm
  - options intéressantes :
  - `update_config` : pour pouvoir rollback si l'update fail
  - `placement` : pouvoir choisir le nœud sur lequel sera déployé le service
  - `replicas` : nombre d'exemplaires du conteneur
  - `resources` : contraintes d'utilisation de CPU ou de RAM sur le nœud
- 

## **Sous-commandes Swarm**

- `swarm init` : Activer Swarm et devenir manager d'un cluster d'un seul nœud
- `swarm join` : Rejoindre un cluster Swarm en tant que nœud manager ou worker
- `service create` : Créer un service (= un conteneur en plusieurs exemplaires)
- `service inspect` : Infos sur un service
- `service ls` : Liste des services
- `service rm` : Supprimer un service

- `service scale` : Modifier le nombre de conteneurs qui fournissent un service
- `service ps` : Liste et état des conteneurs qui fournissent un service
- `service update` : Modifier la définition d'un service
- `docker stack deploy` : Déploie une stack (= fichier Docker compose) ou update une stack existante
- `docker stack ls` : Liste les stacks
- `docker stack ps` : Liste l'état du déploiement d'une stack
- `docker stack rm` : Supprimer une ou des stacks
- `docker stack services` : Liste les services qui composent une stack
- `docker node inspect` : Informations détaillées sur un nœud
- `docker node ls` : Liste les nœuds
- `docker node ps` : Liste les tâches en cours sur un nœud
- `docker node promote` : Transforme un nœud

worker en manager

- docker node demote : Transforme un nœud manager en worker
- 

## Répartition de charge (load balancing)

- Un load balancer : une sorte d'**“aiguillage” de trafic réseau**, typiquement HTTP(S) ou TCP.
- Un aiguillage **intelligent** qui se renseigne sur plusieurs critères avant de choisir la direction.
- Cas d'usage :
- Éviter la surcharge : les requêtes sont réparties sur différents backends pour éviter de les saturer.
- Haute disponibilité : on veut que notre service soit toujours disponible, même en cas de panne (partielle) ou de maintenance.
- Donc on va dupliquer chaque partie de notre service et mettre les différentes instances derrière un load balancer.
- Le load balancer va vérifier pour chaque backend s'il est disponible (**healthcheck**) avant de rediriger le trafic.
- Répartition géographique : en fonction de la

provenance des requêtes on va rediriger vers un datacenter adapté (+ ou - proche)

---

## Le loadbalancing de Swarm est automatique

- Loadbalancer intégré : Ingress
  - Permet de router automatiquement le trafic d'un service vers les nœuds qui l'hébergent et sont disponibles.
  - Pour héberger une production il suffit de rajouter un loadbalancer externe qui pointe vers un certain nombre de nœuds du cluster et le trafic sera routé automatiquement à partir de l'un des nœuds.
- 

## Solutions de loadbalancing externe

- **HAProxy** : Le plus répandu en loadbalancing
  - **Træfik** : Simple à configurer et fait pour l'écosystème Docker
  - **NGINX** : Serveur web générique mais a depuis quelques années des fonctions puissantes de loadbalancing et de TCP forwarding.
-

# Gérer les données sensibles dans Swarm avec les secrets Docker

- echo "This is a secret" | docker secret create my\_secret\_data
  - docker service create --name monservice --secret my\_secret\_data redis:alpine => monte le contenu secret dans /var/run/my\_secret\_data
- 

## Docker Machine

- C'est l'outil de gestion d'hôtes Docker
- Il est capable de créer des serveurs Docker “à la volée”
- Concrètement, docker-machine permet de **créer automatiquement des machines** avec le **Docker Engine** et **ssh** configuré et de gérer les **certificats TLS** pour se connecter à l'API Docker des différents serveurs.
- Il permet également de changer le contexte de la ligne de commande Docker pour basculer sur l'un ou l'autre serveur avec les variables d'environnement adéquates.

- Il permet également de se connecter à une machine en ssh en une simple commande.

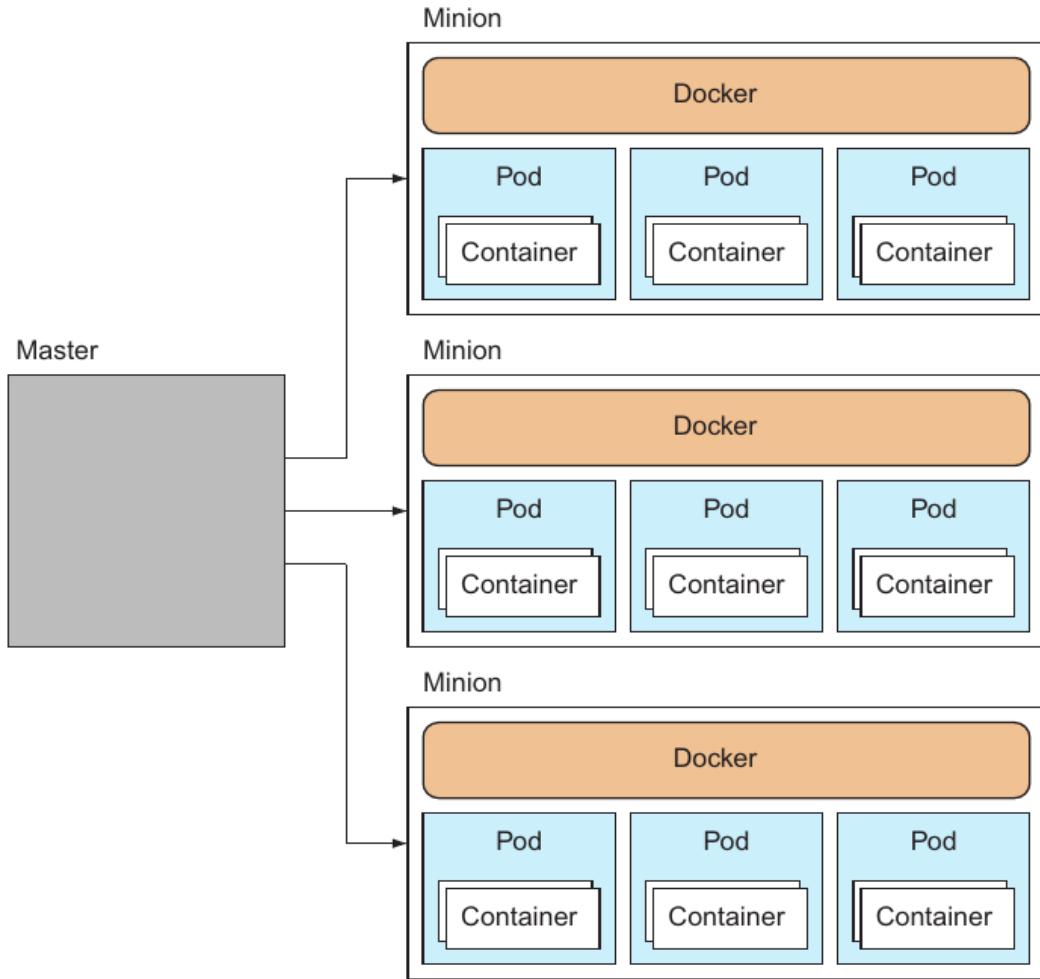
Exemple :

```
docker-machine create --driver
digitalocean \
    --digitalocean-ssh-key-
fingerprint
41:d9:ad:ba:e0:32:73:58:4f:09:28:15:f2
\
    --digitalocean-access-token
"a94008870c9745febbb2bb84b01d16b6bf837
\
    hote-digitalocean
```

Pour basculer eval \$(docker env hote-digitalocean);

- docker run -d nginx:latest créé ensuite un conteneur **sur le droplet digitalocean** précédemment créé.
- docker ps -a affiche le conteneur en train de tourner à distance.
- wget \$(docker-machine ip hote-digitalocean) va récupérer la page nginx.

# Présentation de Kubernetes



- Les **pods** Kubernetes servent à grouper des conteneurs en unités d'application (microservices ou non) fortement couplées (un peu comme les *stacks* Swarm)
- Les **services** sont des groupes de pods exposés à l'extérieur
- 
- Les **deployments** sont une abstraction pour scaler ou mettre à jours des groupes de **pods** (un peu

comme les *tasks* dans Swarm).

---

## Présentation de Kubernetes

- Une autre solution très à la mode depuis 4 ans. Un buzz word du DevOps en France :)
  - Une solution **robuste, structurante et open source** d'orchestration Docker.
  - Au cœur du consortium **Cloud Native Computing Foundation** très influent dans le monde de l'informatique.
  - Hébergeable de façon identique dans le cloud, on-premise ou en mixte.
  - Kubernetes a un flat network (un overlay de plus bas niveau que Swarm) : <https://neuvendor.com/network-security/kubernetes-networking/>
- 

## Comparaison Swarm et Kubernetes

- Swarm plus intégré avec la CLI et le workflow Docker.
- Swarm est plus fluide, moins structurant mais moins automatique que Kubernetes.
- Swarm groupe les containers entre eux par **stack**.

- Kubernetes au contraire crée des **pods** avec une meilleure isolation.
  - Kubernetes a une meilleure fault tolerance que Swarm
  - attention au contre-sens : un service Swarm est un seul conteneur répliqué, un service Kubernetes est un groupe de conteneurs (pod) répliqué, plus proche des Docker Stacks.
- 

## Comparaison Swarm et Kubernetes

- Kubernetes a plus d'outils intégrés. Il s'agit plus d'un écosystème qui couvre un large panel de cas d'usage.
  - Swarm est beaucoup plus simple à mettre en œuvre qu'une stack Kubernetes.
  - Swarm serait donc mieux pour les clusters moyen et Kubernetes pour les très gros
- 

## TP 5 - Orchestration et clustering

### Introduction à Swarm

Initialisez Swarm avec `docker swarm init`.

# Créer un service

A l'aide de la propriété `deploy` : de `docker compose`, créer un service en 5 exemplaires (`replicas`) à partir de l'image `traefik/whoami` accessible sur le port 9999 et connecté au port 80 des 5 replicas.

Accédez à votre service et actualisez plusieurs fois la page. Les informations affichées changent.

Pourquoi ?

- Lancez une commande `service scale` pour changer le nombre de `replicas` de votre service et observez le changement avec `docker service ps hello`

## La stack `example-voting-app`

- Cloner l'application `example-voting-app` ici :  
<https://github.com/dockersamples/example-voting-app>
- Lire le schéma d'architecture de l'app `example-voting-app` sur Github. A noter que le service `worker` existe en deux versions utilisant un langage de programmation différent (Java ou

.NET), et que tous les services possèdent des images pour conteneurs Windows et pour conteneurs Linux. Ces versions peuvent être déployées de manière interchangeable et ne modifient pas le fonctionnement de l'application multi-conteneur. C'est une démonstration de l'utilité du paradigme de la conteneurisation et de l'architecture dite “*micro-service*”.

- Lire attentivement les fichiers `docker-compose.yml`, `docker-compose-simple.yml`, `docker-stack-simple.yml` et `docker-stack.yml`. Ce sont tous des fichiers Docker Compose classiques avec différentes options liées à un déploiement via Swarm. Quelles options semblent spécifiques à Docker Swarm ? Ces options permettent de configurer des fonctionnalités d'**orchestration**.
- Dessiner rapidement le schéma d'architecture associé au fichier `docker-compose-simple.yml`, puis celui associé à `docker-stack.yml` en indiquant bien à quel réseau quel service appartient.
- Avec `docker swarm init`, transformer son

installation Docker en une installation Docker compatible avec Swarm. Lisez attentivement le message qui vous est renvoyé.

- Déployer la stack du fichier `docker-stack.yml` :  
`docker stack deploy --compose-file docker-stack.yml vote`
- `docker stack ls` indique 6 services pour la stack `vote`. Observer également l'output de `docker stack ps vote` et de `docker stack services vote`. Qu'est-ce qu'un service dans la terminologie de Swarm ?
- Accéder aux différents front-ends de la stack grâce aux informations contenues dans les commandes précédentes. Sur le front-end lié au `vote`, actualiser plusieurs fois la page. Que signifie la ligne `Processed by container ID [...]` ? Pourquoi varie-t-elle ?
- Scaler la stack en ajoutant des *replicas* du front-end lié au `vote` avec l'aide de `docker service --help`. Accédez à ce front-end et vérifier que cela a bien fonctionné en actualisant plusieurs fois.

## Clustering entre ami·es

## Avec un service

- Se grouper par 2 ou 3 pour créer un cluster à partir de vos VM respectives (il faut utiliser une commande Swarm pour récupérer les instructions nécessaires).
- Si grouper plusieurs des VM n'est pas possible, vous pouvez créer un cluster multi-nodes très simplement avec l'interface du site [Play With Docker](#), il faut s'y connecter avec vos identifiants Docker Hub.
- Vous pouvez faire `docker swarm --help` pour obtenir des infos manquantes, ou faire `docker swarm leave --force` pour réinitialiser votre configuration Docker Swarm si besoin.
- N'hésitez pas à regarder dans les logs avec `systemctl status docker` comment se passe l'élection du nœud *leader*, à partir du moment où vous avez plus d'un manager.
- Lancez le service suivant : `docker service create --name whoami --replicas 5 --publish published=80,target=80 traefik/whoami`

- Accédez au service depuis un node, et depuis l'autre. Actualisez plusieurs fois la page. Les informations affichées changent. Lesquelles, et pourquoi ?

## Avec la stack example-voting-app

- Si besoin, cloner de nouveau le dépôt de l'application example-voting-app avec `git clone https://github.com/docker-samples/example-voting-app` puis déployez la stack de votre choix.
- Ajouter dans le Compose file des instructions pour scaler différemment deux services (3 *replicas* pour le service *front* par exemple). N'oubliez pas de redéployer votre Compose file.
- puis spécifier quelques options d'orchestration exclusives à Docker Swarm : que fait mode : `global` ? N'oubliez pas de redéployer votre Compose file.
- Avec Portainer ou avec [docker-swarm-visualizer](#), explorer le cluster ainsi créé (le fichier `docker-stack.yml` de l'app example-voting-app contient déjà un exemplaire de `docker-swarm-`

visualizer).

- Trouver la commande pour déchoir et promouvoir l'un de vos nœuds de manager à worker et vice-versa.
- Puis sortir un nœud du cluster (drain) : docker node update --availability drain <node-name>

## ***Facultatif : débugger la config Docker de example-voting-app***

Vous avez remarqué ? Nous avons déployé une super stack d'application de vote avec succès mais, si vous testez le vote, vous verrez que ça ne marche pas, il n'est pas comptabilisé. Outre le fait que c'est un plaidoyer vivant contre le vote électronique, vous pourriez tenter de débugger ça maintenant (c'est plutôt facile).

Solution / explications :

## **Introduction à Kubernetes**

Le fichier kube-deployment.yml de l'app [example-voting-app](#) décrit la même app pour un déploiement dans Kubernetes plutôt que dans

Docker Compose ou Docker Swarm. Tentez de retrouver quelques équivalences entre Docker Compose / Swarm et Kubernetes en lisant attentivement ce fichier qui décrit un déploiement Kubernetes.

### ***Facultatif : Utiliser Traefik avec Swarm***

Vous pouvez désormais faire [l'exercice 2 du TP 7](#) pour configurer un serveur web qui permet d'accéder à vos services Swarm via des domaines spécifiques.

## **Conclusion**

---

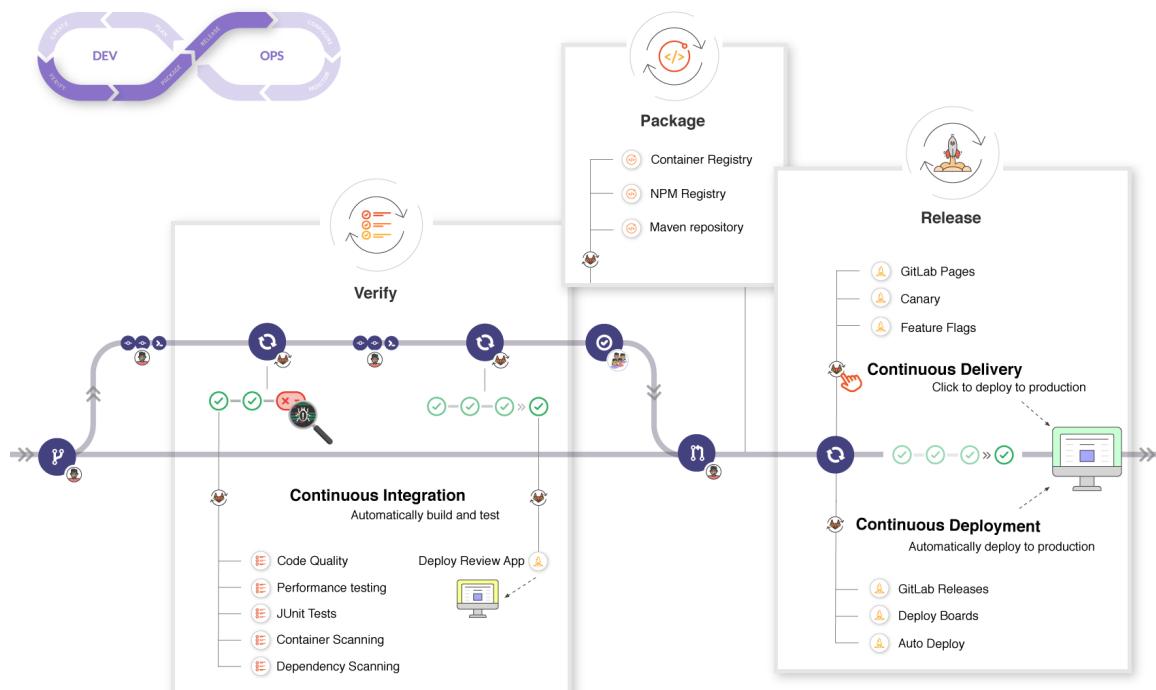
### **Conclusions sur l'écosystème Docker**

#### **Configurer de la CI/CD**

- La nature facile à déployer des conteneurs et l'intégration du principe d'Infrastructure-as-Code les rend indispensable dans de la CI/CD (intégration continue et déploiement continu).
- Les principaux outils de CI sont Gitlab, Jenkins, Github Actions, Travis CI...
- Gitlab propose par défaut des runners

préconfigurés qui utilisent des conteneurs Docker et tournent en général dans un cluster Kubernetes.

- Gitlab propose aussi un registry d'images Docker, privé ou public, par projet.
  - Les tests à l'intérieur des conteneurs peuvent aussi être faits de façon plus poussée, avec par exemple Ansible comme source de healthcheck ou comme suite pour les tests.
  - Dans une autre catégorie, Gitpod base son workflow sur des images Docker permettant de configurer un environnement de développement



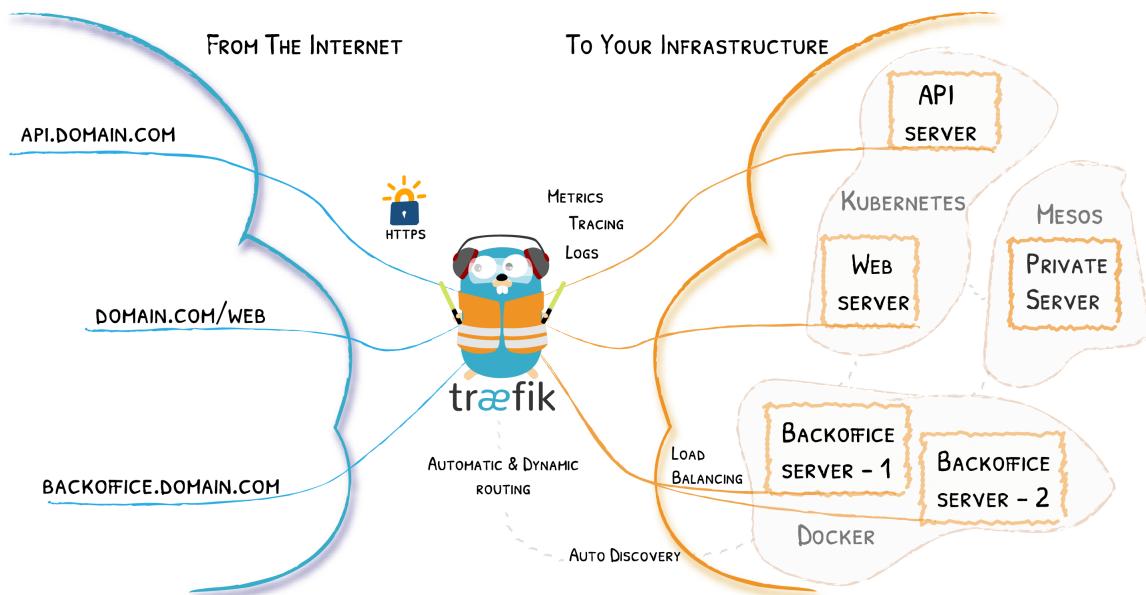
	#49651920 by	⚠ use-fallback... ⚠ 3c50e19e Use encrypted runner tokens		⌚ 00:14:45 ⌚ 24 minutes ago	
	#49649365 by <b>latest</b>	⚠ feature/add... ⚠ fef3c0a8 Fix RuboCop offense			
	#49647437 by	⚠ master ⚠ 40ec172f Merge branch 'web-ide-defa...		⌚ 00:54:04 ⌚ 18 minutes ago	

# Gérer les logs des conteneurs

Avec Elasticsearch, Filebeat et Kibana... grâce aux labels sur les conteneurs Docker

## Gérer le reverse proxy

Avec Traefik, aussi grâce aux labels sur les conteneurs Docker



Ou avec Nginx, avec deux projets :

- <https://github.com/nginx-proxy/nginx-proxy>
- <https://github.com/nginx-proxy/acme-companion>

## Monitorer des conteneurs

- Avec Prometheus pour Docker et Docker Swarm
- Ou bien Netdata, un peu plus joli et configuré pour

monitorer des conteneurs *out-of-the-box*

## Tests sur des conteneurs

Ansible comme source de healthcheck

---

## Bonnes pratiques et outils

### Sécurité / durcissement

- **un conteneur privilégié est *root* sur la machine !**
- des *cgroups* correct : `ulimit -a`
- par défaut les *user namespaces* ne sont pas utilisés !
- exemple de faille : <https://medium.com/@mccode/processes-in-containers-should-not-run-as-root-2feae3f0df3b>
- exemple de durcissement conseillé :  
<https://docs.docker.com/engine/security/users-remap/>
- le benchmark Docker CIS : <https://github.com/docker/docker-bench-security/>
- La sécurité de Docker c'est aussi celle de la chaîne de dépendance, des images, des packages

installés dans celles-ci : on fait confiance à trop de petites briques dont on ne vérifie pas la provenance ou la mise à jour

- Clair : l'analyse statique d'images Docker
- docker-socket-proxy : protéger la *socket* Docker quand on a besoin de la partager à des conteneurs comme Traefik ou Portainer

## **Limites de Docker**

### **Stateful**

- les conteneurs stateless c'est bien beau mais avec une base de données, ça ne se gère pas magiquement du tout
- quelques ressources sur le stateful avec Docker :  
<https://container.training/swarm-selfpaced.yml.html#450>

### **Configurer le réseau de façon plus complexe avec des plugins réseau**

- Réseaux “overlay”: IP in IP, VXLAN...
- ...mais on a rapidement besoin de plugins exclusifs à Kubernetes : Calico, Flannel, Canal (Calico +

Flannel), [Cilium](#) (qui utilise eBPF)

## Volumes distribués

- problème des volumes partagés / répliqués
- domaine à part entière
- **Solution 1** : solutions applicatives robustes
  - pour MySQL/MariaDB : [Galera](#)
  - pour Postgres : on peut citer [Citus](#) ou [pgpool](#), voir la [comparaison de différentes solutions](#)
- Elasticsearch est distribué *out-of-the-box*
- **Solution 2** : volume drivers avec Docker
  - [Flocker](#), [Convoy](#), visent à intégrer une technologie de réPLICATION
  - c'est un moyen, pas une solution : reste un outil pour configurer ce que l'on souhaite

## DataOps

- Doit se baser sur une solution applicative complète comme Kafka
- avec Compose pour les différents services et Swarm pour le scaling

- Solutions actuelles peut-être plus orientées Kubernetes
- Ressources :
- Pas-à-pas très détaillé d'une pipeline "ETL"  
<https://medium.com/sfu-cspmp/building-data-pipeline-kafka-docker-4d2a6fc92ca>
- dépôt lié : <https://github.com/salcaino/sfucmpt733>
- Exemple avec Kafka : <https://github.com/rogaha/data-processing-pipeline/blob/master/docker-compose.yml>

## Aller plus loin

- Le livre *Mastering Docker*, de Russ McKendrick et Scott Gallagher
- les ressources présentes dans la [bibliographie](#)
- la liste de [Awesome Docker](#)



[\*Dockercraft\*](#) :

## Retours

- Comment ça s'est passé ?
- Difficulté : trop facile ? trop dur ? quoi en particulier ?
- Vitesse : trop rapide ? trop lent ? lors de quoi en particulier ?
- Attentes sur le contenu ? Les manipulations ?
- Questions restées ouvertes ? Nouvelles questions ?
- Envie d'utiliser Docker ? ou de le jeter à la poubelle ?

## **TP 6 (bonus) - Intégration continue avec Gitlab**

### **Créer une pipeline de build d'image Docker avec les outils CI/CD Gitlab**

1. Si vous n'en avez pas déjà un, créez un compte sur Gitlab.com : [https://gitlab.com/users/sign\\_in#register-pane](https://gitlab.com/users/sign_in#register-pane)

2. Créez un nouveau projet et avec Git, le Web IDE Gitlab, ou bien en forkant une app existante depuis l'interface Gitlab, poussez-y l'app de votre choix (par exemple [microblog](#), [dnmonster](#) ou l'app `healthcheck` vue au TP2).
3. Ajoutez un Dockerfile à votre repository ou vérifiez qu'il en existe bien un.
4. Créez un fichier `.gitlab-ci.yml` depuis l'interface web de Gitlab et choisissez “Docker” comme template. Observons-le ensemble attentivement.
5. Faites un commit de ce fichier.
6. Vérifiez votre CI : il faut vérifier sur le portail de Gitlab comment s'est exécutée la pipeline.
7. Vérifiez dans la section Container Registry que votre image a bien été push.

## Ressources

- La [section Quick Start de la documentation Gitlab-CI](#)
- Vous pouvez trouver [des exemples de CI dans la documentation Gitlab](#).

- La section dédiée à docker build de la documentation Gitlab
- La section de la documentation dédiée au Container Registry

## Avec BitBucket

BitBucket propose aussi son outil de pipeline, à la différence qu'il n'a pas de registry intégré, le template par défaut propose donc de pousser son image sur le registry Docker Hub.

- Il suffit de créer un repo BitBucket puis d'y ajouter le template de CI Docker proposé (le template est caché derrière un bouton *See more*).
- Ensuite, il faut ajouter des *Repository variables* avec ses identifiants Docker Hub. Dans le template, ce sont les variables DOCKERHUB\_USERNAME, DOCKERHUB\_PASSWORD et DOCKERHUB\_NAMESPACE (identique à l'username ici).

## Ressources

- <https://support.atlassian.com/bitbucket-cloud/docs/run-docker-commands-in-bitbucket-pipelines/>

# Conclusion

## Déployer notre container ou notre projet Docker Compose

Nous avons fait la partie CI (intégration continue).

Une étape supplémentaire est nécessaire pour ajouter le déploiement continu de l'app (CD) : si aucune étape précédente n'a échoué, la nouvelle version de l'app devra être déployée sur votre serveur, via une connexion SSH et rsync par exemple. Il faudra ajouter des variables secrètes au projet (clé SSH privée par exemple), cela se fait dans les options de Gitlab ou de BitBucket.

## TP 7 (bonus) - Docker et les reverse proxies

### Exercice 1 - Utiliser Traefik pour le routage

Traefik est un reverse proxy très bien intégré à Docker. Il permet de configurer un routage entre un point d'entrée (ports 80 et 443 de l'hôte) et des containers Docker, grâce aux informations du daemon Docker et aux labels sur chaque

containers. Nous allons nous baser sur le guide d'introduction [Traefik - Getting started](#).

- Avec l'aide de la documentation Traefik, ajoutez une section pour le reverse proxy Traefik pour dans un fichier Docker Compose de votre choix.
- Explorez le dashboard Traefik accessible sur le port indiqué dans le fichier Docker Compose.

Pour que Traefik fonctionne, 2 étapes :

- faire en sorte que Traefik reçoive la requête quand on s'adresse à l'URL voulue (DNS + routage)
- faire en sorte que Traefik sache vers quel conteneur rediriger le trafic reçu (et qu'il puisse le faire)
- Ajouter des labels à l'app web que vous souhaitez desservir grâce à Traefik à partir de l'exemple de la doc Traefik, grâce aux labels ajoutés dans le docker-compose.yml (attention à l'indentation).
- Avec l'aide de la [documentation Traefik sur Let's Encrypt et Docker Compose](#), configurez Traefik pour qu'il crée un certificat Let's Encrypt pour votre container.

## **Exercice 2 - Swarm avec Traefik**

- Avec l'aide de la [documentation Traefik sur Docker Swarm](#), configurez Traefik avec Swarm.

## QCM Docker

**Entourez la bonne réponse**

### Question 1

Quelle est la principale différence entre une machine virtuelle (VM) et un conteneur ?

1. Un conteneur est une boîte qui contient un logiciel Windows alors qu'une VM fonctionne généralement sous Linux.
2. Un conteneur permet de faire des applications distribuées dans le cloud contrairement aux machines virtuelles.
3. Un conteneur partage le noyau du système hôte alors qu'une machine virtuelle virtualise son propre noyau indépendant.

### Question 2

En quoi Docker permet de faire de l'*Infrastructure as Code* ?

1. Comme Ansible, Docker se connecte en SSH à un

Linux pour décrire des configurations.

2. Docker permet avec les Dockerfiles et les fichiers Compose de décrire l'installation d'un logiciel et sa configuration.

## Question 3

Quels sont les principaux atouts de Docker ?

1. Il permet de rendre compatible tous les logiciels avec le cloud (AWS, etc.) et facilite l'IoT.
2. Il utilise le langage Go qui est de plus en plus populaire et accélère les logiciels qui l'utilise.
3. Il permet d'uniformiser les déploiements logiciels et facilite la construction d'application distribuées.

## Question 4

Pour créer un conteneur Docker à partir du code d'un logiciel il faut d'abord :

1. Écrire un Dockerfile qui explique comment empaqueter le code puis construire l'image Docker avec docker build.
2. Créer un cluster avec docker-machine puis compiler le logiciel avec Docker Stack.

## **Question 5**

Un volume Docker est :

1. Un espace de stockage connecté à un ou plusieurs conteneurs docker.
2. Une image fonctionnelle à partir de laquelle on crée des conteneurs identiques.
3. Un snapshot de l'application que l'on déploie dans un cluster comme Swarm.

## **Question 6**

Indiquez la ou les affirmations vraies :

Comment configurer de préférence un conteneur à sa création (lancement avec docker run) ?

1. Reconstruire l'image à chaque fois à partir du Dockerfile avant.
2. Utiliser des variables d'environnement pour définir les paramètres à la volée.
3. Faire docker exec puis aller modifier les fichiers de configuration à l'intérieur
4. Associer le conteneur à un volume qui rassemble des fichiers de configuration

## **Question 7**

Un *Compose file* ou fichier Compose permet :

1. D'installer Docker facilement sur des VPS et de contrôler un cluster.
2. D'alléger les images et de détecter les failles de sécurité dans le packaging d'une application.
3. De décrire une application multiconteneurs, sa configuration réseau et son stockage.

## **Question 8**

Indiquez la ou les affirmations vraies :

La philosophie de Docker est basée sur :

1. L'immutabilité, c'est-à-dire le fait de jeter et recréer un conteneur pour le changer plutôt que d'aller modifier l'intérieur.
2. Le cloud, c'est-à-dire la vente de plateforme et de logiciel “as a service”.
3. L'infrastructure-as-code, c'est-à-dire la description d'un état souhaité de l'infrastructure hébergeant application

## **Question 9**

Indiquez la ou les affirmations vraies :

1. Docker est très pratique pour distribuer un logiciel mais tous les conteneurs doivent obligatoirement être exposés à Internet.
2. Docker utilise un cloud pour distribuer facilement des logiciels dans de nombreuses versions.
3. Docker est une catastrophe en terme de sécurité car les conteneurs sont peu isolés.

## Question 10

Docker Swarm est :

1. Un cloud où pousser et récupérer les images Docker de la terre entière.
2. Une solution de clustering et d'orchestration intégrée directement avec Docker.
3. Un logiciel qui complète ce qu'offre Kubernetes en y ajoutant des fonctionnalités manquantes

## Bibliographie

### Docker

- McKendrick, Gallagher 2017 Mastering Docker - Second Edition

## Pour aller plus loin

- Miell,Sayers2019 - Docker in Practice

## Cheatsheet

- <https://devhints.io/docker>

## Ressources

- Doc officielle : <https://docs.docker.com/>
- Référence officielle : <https://docs.docker.com/reference/>
- Awesome Docker, liste de ressources sur Docker :  
<https://github.com/veggiemonk/awesome-docker>
- Portainer, GUI pour Docker :  
<https://www.portainer.io/installation/>
- Lazy Docker, terminal pour Docker :  
<https://github.com/jesseduffield/lazydocker>
- Convoy, driver pour volumes Docker :  
<https://github.com/rancher/convoy>
- Marcel, le Docker français : <https://github.com/brouberol/marcel>
- Accéder à ses containers dans Minecraft :

<https://github.com/docker/dockercraft>

- Doc officielle sur la sécurité dans Docker :  
<https://docs.docker.com/engine/security/>
- Documentation sur le système de filesystem overlay de Docker : <https://docs.docker.com/storage/storagedriver/overlayfs-driver/>
- Tutoriels officiels sur la sécurité dans Docker :  
<https://github.com/docker/labs/tree/master/security>
- Vidéo sur les bonnes pratiques dans Docker :  
<https://noti.st/aurelievache/PrttUh>
- Vidéo “Containers, VMs... Comment ces technologies fonctionnent et comment les différencier ?” (Quentin Adam)  
[https://www.youtube.com/watch?v=wG4\\_JQXvZlc](https://www.youtube.com/watch?v=wG4_JQXvZlc)
- Diapositives sur Docker, Swarm, Kubernetes :  
<https://container.training/>
- en particulier les problèmes du stateful :  
<https://container.training/swarm-selfpaced.yml.html#450>

## DevOps

- Krief - Learning DevOps - The complete guide

(Azure Devops, Jenkins, Kubernetes, Terraform, Ansible, sécurité) - 2019

- The DevOps Handbook

## Kubernetes

## Introduction DevOps

### A propos de moi

### A propos de vous

- Attentes ?
- Début du cursus :
- Est-ce que ça vous plait ?
- Quels modules avez vous déjà fait ?

## Le mouvement DevOps

Le DevOps est avant tout le nom d'un mouvement de transformation professionnelle et technique de l'informatique.

Ce mouvement se structure autour des solutions **humaines** (organisation de l'entreprise et des équipes) et **techniques** (nouvelles technologies de rupture) apportées pour répondre aux défis que

sont:

- L'agrandissement rapide face à la demande des services logiciels et infrastructures les supportant.
- La célérité de déploiement demandée par le développement agile (cycles journaliers de développement).
- Difficultées à organiser des équipes hétérogènes de grande taille et qui s'agrandissent très vite selon le modèle des startups.

Il y a de nombreuses versions de ce que qui caractérise le DevOps mais pour résumer:

Du côté humain:

- Application des processus de management agile aux opérations et la gestion des infrastructures (pour les synchroniser avec le développement).
- Remplacement des procédés d'opérations humaines complexes et spécifiques par des opérations automatiques et mieux standardisées.
- Réconciliation de deux cultures divergentes (Dev et Ops) rapprochant en pratique les deux métiers du développeur et de l'administrateur système.

Du côté technique:

- L'intégration et le déploiement continus des logiciels/produits.
- L'infrastructure as code: gestion sous forme de code de l'état des infrastructures d'une façon le plus possible déclarative.
- Les conteneurs (Docker surtout mais aussi Rkt et LXC/LXD): plus léger que la virtualisation = permet d'isoler chaque service dans son “OS” virtuel sans dupliquer le noyau.
- Le cloud (Infra as a service, Plateforme as a Service, Software as a service) permet de fluidifier l'informatique en alignant chaque niveau d'abstraction d'une pile logicielle avec sa structuration économique sous forme de service.

## Aller plus loin

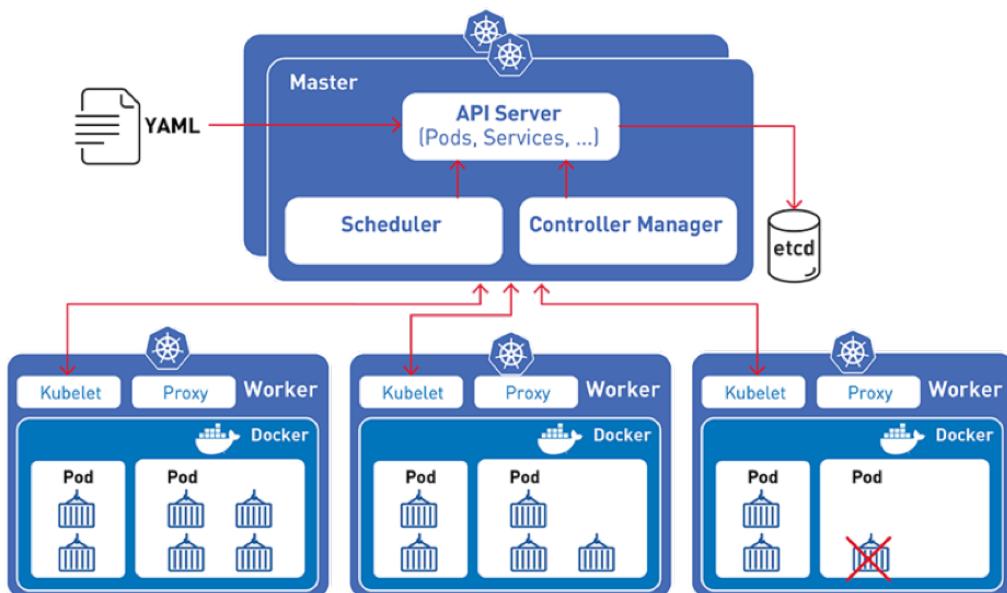
- La DevOps roadmap: <https://github.com/kamranahmedse/developer-roadmap#devops-roadmap>

## 01 - Cours - Présentation de Kubernetes

- Kubernetes est une solution d'orchestration de conteneurs extrêmement populaire.

- Le projet est très ambitieux : une façon de considérer son ampleur est de voir Kubernetes comme un système d'exploitation (et un standard ouvert) pour les applications distribuées et le cloud.
- Le projet est développé en Open Source au sein de la Cloud Native Computing Foundation.

## Concrètement : Architecture de Kubernetes



- Kubernetes rassemble en un cluster et fait coopérer un groupe de serveurs appelés **noeuds**(nodes).
- Kubernetes a une architecture **Master/workers** (cf. cours 2) composée d'un **control plane** et de nœuds de calculs (**workers**).
- Cette architecture permet essentiellement de

rassembler les machines en un **cluster unique** sur lequel on peut faire tourner des “**charges de calcul**” (**workloads**) très diverses.

- Sur un tel cluster le déploiement d'un workload prend la forme de **ressources (objets k8s)** qu'on **décrit sous forme de code** et qu'on crée ensuite effectivement via l'API Kubernetes.
- Pour uniformiser les déploiements logiciel Kubernetes est basé sur le standard des **conteneurs** (défini aujourd'hui sous le nom **Container Runtime Interface**, Docker est l'implémentation la plus connue).
- Plutôt que de déployer directement des conteneurs, Kubernetes crée des **aggrégats de un ou plusieurs conteneurs** appelés des **Pods**. Les pods sont donc l'unité de base de Kubernetes.

## **Philosophie derrière Kubernetes et le mouvement “Cloud Native”**

### **Historique et popularité**





Kubernetes est un logiciel développé originellement par Google et basé sur une dizaine d'années d'expérience de déploiement d'applications énormes (distribuées) sur des clusters de machines.

Dans la mythologie Cloud Native on raconte que son ancêtre est l'orquestrateur borg utilisé par Google dans les années 2000.

La première version est sortie en 2015 et k8s est devenu depuis l'un des projets open source les plus populaires du monde.

L'écosystème logiciel de Kubernetes s'est développée autour la **Cloud Native Computing Foundation** qui comprend notamment : Google, CoreOS, Mesosphere, Red Hat, Twitter, Huawei, Intel, Cisco, IBM, Docker, Univa et VMware. Cette fondation vise au pilotage et au financement collaboratif du développement de Kubernetes (un peu comme la Linux Foundation).

**Trois transformations profondes de l'informatique**

Kubernetes se trouve au cœur de trois transformations profondes techniques, humaines et économiques de l'informatique:

- Le cloud
- La conteneurisation logicielle
- Le mouvement DevOps

Il est un des projets qui symbolise et supporte techniquement ces transformations. D'où son omniprésence dans les discussions informatiques actuellement.

## Le Cloud

- Au delà du flou dans l'emploi de ce terme, le cloud est un mouvement de réorganisation technique et économique de l'informatique.
- On retourne à la consommation de “temps de calcul” et de services après une “aire du Personal Computer”.
- Pour organiser cela on définit trois niveaux à la fois techniques et économiques de l'informatique:
- **Software as a Service:** location de services à travers internet pour les usagers finaux

- **Platform as a Service:** location d'un environnement d'exécution logiciel flexible à destination des développeurs
- **Infrastructure as a Service:** location de ressources "matérielles" à la demande pour installer des logiciels sans avoir à maintenir un data center.

## Conteneurisation

La conteneurisation est permise par l'isolation au niveau du noyau du système d'exploitation du serveur : les processus sont isolés dans des namespaces au niveau du noyau. Cette innovation permet de simuler l'isolation sans ajouter une couche de virtualisation comme pour les machines virtuelles.

Ainsi les conteneurs permettent d'avoir des performances proche d'une application traditionnelle tournant directement sur le système d'exploitation hôte et ainsi d'optimiser les ressources.

Les images de conteneurs sont aussi beaucoup plus légers qu'une image de VM ce qui permet de Les technologies de conteneurisation permettent

donc de faire des boîtes isolées avec les logiciels pour apporter l'uniformisation du déploiement:

- Un façon standard de packager un logiciel (basée sur le)
- Cela permet d'assembler de grosses applications comme des legos
- Cela réduit la complexité grâce:
  - à l'intégration de toutes les dépendance déjà dans la boîte
  - au principe d'immutabilité qui implique de jeter les boîtes ( automatiser pour lutter contre la culture prudence). Rend l'infra prédictible.

Les conteneurs sont souvent comparés à l'innovation du porte conteneur pour le transport de marchandise.

## **Le mouvement DevOps**

- Dépasser l'opposition culturelle et de métier entre les développeurs et les administrateurs système.
- Intégrer tout le monde dans une seule équipe et ...
- Calquer les rythmes de travail sur l'organisation agile du développement logiciel

- Rapprocher techniquement la gestion de l'infrastructure du développement avec l'infrastructure as code.
- Concrètement on écrit des fichiers de code pour gérer les éléments d'infra
- l'état de l'infrastructure est plus claire et documentée par le code
- la complexité est plus gérable car tout est déclaré et modifiable au fur et à mesure de façon centralisée
- l'usage de git et des branches/tags pour la gestion de l'évolution d'infrastructure

## **Objectifs du DevOps**

- Rapidité (velocity) de déploiement logiciel (organisation agile du développement et livraison jusqu'à plusieurs fois par jour)
- Implique l'automatisation du déploiement et ce qu'on appelle la CI/CD c'est à dire une infrastructure de déploiement continu à partir de code.
- Passage à l'échelle (horizontal scaling) des logiciels et des équipes de développement

(nécessaire pour les entreprises du cloud qui doivent servir pleins d'utilisateurs)

- Meilleure organisation des équipes
- meilleure compréhension globale du logiciel et de son installation de production car le savoir est mieux partagé
- organisation des équipes par thématique métier plutôt que par spécialité technique (l'équipe scale mieux)

## **Apports techniques de Kubernetes pour le DevOps**

- Abstraction et standardisation des infrastructures:
- Langage descriptif et incrémental: on décrit ce qu'on veut plutôt que la logique complexe pour l'atteindre
- Logique opérationnelle intégrée dans l'orchestrateur: la responsabilité des l'état du cluster est laissé au contrôleur k8s ce qui simplifie le travail

On peut alors espérer **fluidifier** la gestion des défis techniques d'un grosse application et atteindre plus ou moins la livraison logicielle continue (CD de

## Architecture logicielle optimale pour Kubernetes

Kubernetes est très versatile et permet d'installer des logiciels traditionnels “monolithiques” (gros backends situés sur une seule machine).

Cependant aux vues des transformations humaines et techniques précédentes, l'organisation de Kubernetes prend vraiment sens pour le développement d'applications microservices:

- des applications avec de nombreux de “petits” services.
- chaque service a des problématiques très limitées (gestion des factures = un logiciel qui fait que ça)
- les services communiquent par le réseaux selon différents modes/API (REST, gRPC, job queues, GraphQL)

Les microservices permettent justement le DevOps car:

- ils peuvent être déployés séparément
- une petite équipe gère chaque service ou groupe

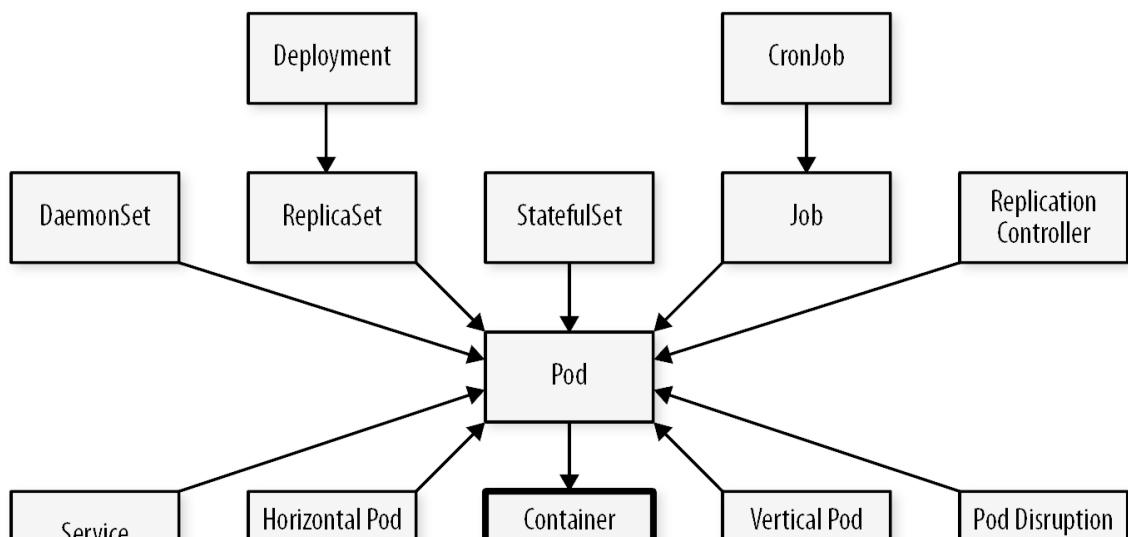
thématique de services

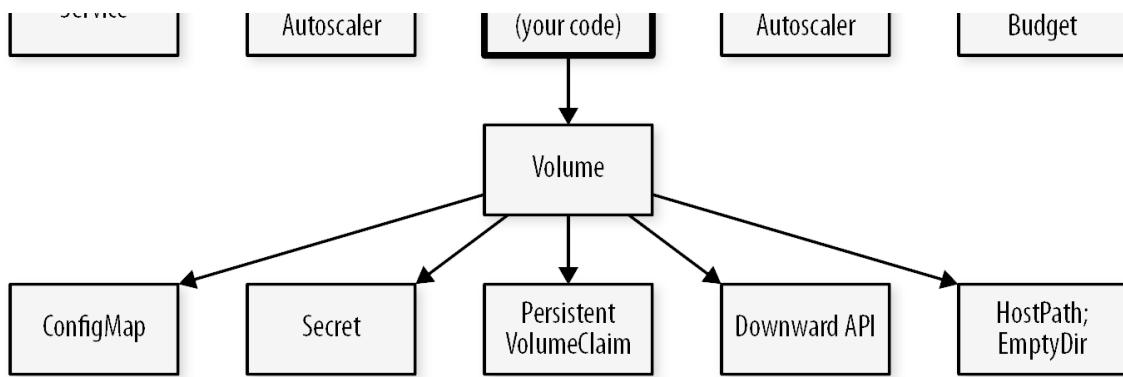
Nous y reviendrons pour expliquer l'usage des ressources Kubernetes.

## Objets fondamentaux de Kubernetes

- Les **pods** Kubernetes servent à grouper des conteneurs fortement couplés en unités d'application
- Les **deployments** sont une abstraction pour **créer ou mettre à jour** (ex : scaler) des groupes de **pods**.
- Enfin, les **services** sont des points d'accès réseau qui permettent aux différents workloads (deployments) de communiquer entre eux et avec l'extérieur.

Au delà de ces trois éléments, l'écosystème d'objets de Kubernetes est vaste et complexe





## Kubernetes entre Cloud et auto-hébergement

Un des intérêts principaux de Kubernetes est de fournir un modèle de Plateform as a Service (PaaS) suffisamment versatile qui permet l'interopérabilité entre des fournisseurs de clouds différents et des solutions auto-hébergées (on premise).

Cependant cette interopérabilité n'est pas automatique (pour les cas complexes) car Kubernetes permet beaucoup de variations.

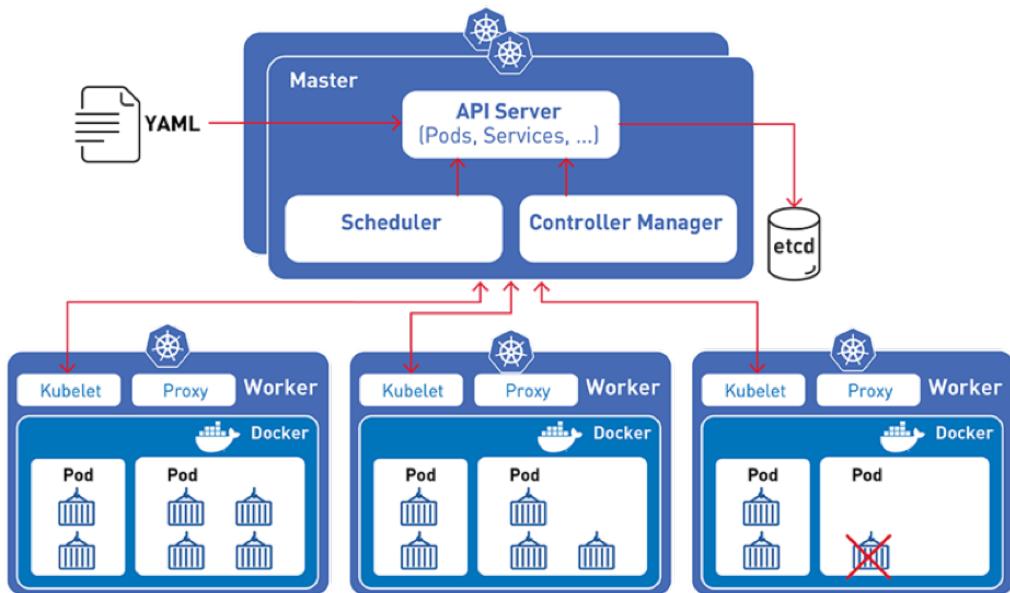
Concrètement il existe des variations entre les installations possibles de Kubernetes

## 02 - Cours - Mettre en place un cluster Kubernetes

### Architecture de Kubernetes

Kubernetes a une architecture master/worker ce qui

signifie que certains noeuds du cluster contrôlent l'exécution par les autres noeuds de logiciels ou jobs.



## Les noeuds Kubernetes

Les nœuds d'un cluster sont les machines (serveurs physiques, machines virtuelles, etc. généralement Linux mais plus toujours) qui exécutent vos applications et vos workflows. Tous les noeuds d'un cluster (master et workers) font tourner trois services de base:

- Comme tout est conteneur dans Kubernetes, chaque noeud doit avoir une runtime de conteneur compatible OCI comme Docker (ou containerd ou cri-o, Docker n'est pas la plus recommandée).

- Le kubelet composant (binaire en go, le seul composant non conteneur) qui contrôle la création et l'état des pods/conteneur sur son noeud.
- Le kube-proxy, un proxy réseau reflétant les services Kubernetes sur chaque nœud.

Pour utiliser Kubernetes, on définit un état souhaité en créant des ressources (pods/conteneurs, volumes, permissions etc). Cet état souhaité et son application est géré par le control plane composé des noeuds master.

## **Les noeuds master kubernetes forment le Control Plane du Cluster**

Le control plane est responsable du maintien de l'état souhaité des différents éléments de votre cluster. Lorsque vous interagissez avec Kubernetes, par exemple en utilisant l'interface en ligne de commande kubectl, vous communiquez avec les noeuds master de votre cluster.

Le control plane conserve un enregistrement de tous les objets Kubernetes du système. À tout moment, des boucles de contrôle s'efforcent de faire converger l'état réel de tous les objets du

système pour correspondre à l'état souhaité que vous avez fourni. Pour gérer l'état réel de ces objets sous forme de conteneurs (toujours) avec leur configuration le control plane envoie des instructions aux différents kubelets des noeuds.

Donc concrètement les noeuds du control plane Kubernetes font tourner, en plus de kubelet et kube-proxy, un ensemble de services de contrôle:

- **kube-apiserver**: expose l'API (rest) kubernetes, point d'entrée universel pour parler au cluster.
- **kube-controller-manager**: contrôle en permanence l'état des resources et essaie de le corriger s'il n'est plus conforme.
- **kube-scheduler**: Surveille et cartographie les resources matérielles et le placement des pods sur les différents noeuds pour décider ou doivent être créés ou supprimés les conteneurs/pods.
- **cloud-controller-manager**: Composant *facultatif* qui gère l'intégration avec le fournisseur de cloud comme par exemple la création automatique de loadbalancers pour exposer les applications kubernetes à l'extérieur du cluster.

L'ensemble de la configuration kubernetes est stockée de façon résiliante (consistance + haute disponibilité) dans un gestionnaire configuration distribué qui est presque toujours etcd (mais d'autres base de données peuvent être utilisées).

etcd peut être installé de façon redondante sur les noeuds du control plane ou configuré comme un système externe sur un autre ensemble de serveurs.

Lien vers la documentation pour plus de détails sur les composants : <https://kubernetes.io/docs/concepts/overview/components/>

## Interagir avec le cluster : le client CLI `kubectl`

En pratique on interagit avec le cluster à l'aide d'un client CLI depuis sa machine de travail. Ce client se charge de traduire en appel d'API les commandes de manipulation. Cette cli ressemble sous pas mal d'aspect à celle de Docker (cf. TP1 et TP2). Elle permet de :

- Lister les ressources
- Créer et supprimer les ressources

- Gérer les droits d'accès
- etc.

kubectl s'installe avec un gestionnaire de paquet classique mais est souvent fourni directement avec les distributions de développement de kubernetes que nous verrons par la suite.

## Configuration de connexion `kubeconfig`

Pour se connecter, kubectl a besoin de l'adresse de l'API Kubernetes, d'un nom d'utilisateur et d'un certificat.

- Ces informations sont fournies sous forme d'un fichier YAML appelé `kubeconfig`
- Comme nous le verrons en TP ces informations sont généralement fournies directement par le fournisseur d'un cluster k8s (provider ou k8s de dev)

Le fichier `kubeconfig` par défaut se trouve sur Linux à l'emplacement `~/.kube/config`.

On peut aussi préciser la configuration au *runtime* comme ceci: `kubectl --kubeconfig=fichier_kubeconfig.yaml`

## <commandes\_k8s>

Le même fichier kubeconfig peut stocker plusieurs configurations dans un fichier YAML :

Exemple :

```
apiVersion: v1

clusters:
- cluster:
    certificate-authority: /home/jacky/.minikube/ca.crt
    server: https://172.17.0.2:8443
    name: minikube
- cluster:
    certificate-authority-data:
        LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk
    server: https://5ba26bee-
00f1-4088-
ae11-22b6dd058c6e.k8s.ondigitalocean.c
    name: do-lon1-k8s-tp-cluster

contexts:
- context:
    cluster: minikube
    user: minikube
```

```
    name: minikube
- context:
    cluster: do-lon1-k8s-tp-cluster
    user: do-lon1-k8s-tp-cluster-admin
    name: do-lon1-k8s-tp-cluster
current-context: do-lon1-k8s-tp-
cluster

kind: Config
preferences: {}

users:
- name: do-lon1-k8s-tp-cluster-admin
  user:
    token:
8b2d33e45b980c8642105ec827f41ad343e818
- name: minikube
  user:
    client-certificate: /home/jacky
    /.minikube/profiles/minikube
    /client.crt
    client-key: /home/jacky/.minikube
    /profiles/minikube/client.key
```

Ce fichier déclare 2 clusters (un local, un distant), 2 contextes et 2 users.

# Créer un cluster Kubernetes

## Installation de développement

Pour installer un cluster de développement :

- solution officielle : Minikube, tourne dans Docker par défaut (ou dans des VMs)
- solution très pratique et “vanilla”: kind
- avec Docker Desktop depuis peu (dans une VM aussi)
- un cluster léger avec k3s, de Rancher (simple et utilisable en production/edge)

## Créer un cluster en tant que service (*managed cluster*) chez un fournisseur de cloud.

Tous les principaux fournisseurs de cloud proposent depuis plus ou moins longtemps des solutions de cluster gérées par eux :

- Google Cloud Plateform avec Google Kubernetes Engine (GKE) : très populaire car très flexible et l'implémentation de référence de Kubernetes.
- AWS avec EKS : Kubernetes assez standard mais à la sauce Amazon pour la gestion de l'accès, des

loadbalancers ou du scaling.

- Azure avec AKS : Kubernetes assez standard mais à la sauce Amazon pour la gestion de l'accès, des loadbalancers ou du scaling.
- DigitalOcean ou Scaleway : un peu moins de fonctions mais plus simple à appréhender

Pour sa qualité on recommande parfois Google GKE qui est plus ancien. Mais comme les gros fournisseurs proposent des services éprouvés, il s'agit surtout de faciliter l'intégration avec l'existant:

- Si vous utilisez déjà des ressources AWS ou Azure il est plus commode de louer chez l'un d'eux votre cluster

## **Installer un cluster de production on premise : l'outil “officiel” kubeadm**

kubeadm est un utilitaire (on parle parfois d'opérateur) aider à générer les certificats et les configurations spéciques pour le control plane et connecter les noeuds au control plane. Il permet également d'assister les tâches de maintenance comme la mise à jour progressive (rolling) de chaque noeud du cluster.

- Installer le dæmon Kubelet sur tous les noeuds
- Installer l'outil de gestion de cluster kubeadm sur un noeud master
- Générer les bons certificats avec kubeadm
- Installer un réseau CNI k8s comme flannel (d'autres sont possible et le choix vous revient)
- Déployer la base de données etcd avec kubeadm
- Connecter les nœuds worker au master.

L'installation est décrite dans la [documentation officielle](#)

Opérer et maintenir un cluster de production Kubernetes “à la main” est très complexe et une tâche à ne pas prendre à la légère. De nombreux éléments doivent être installés et géré par une équipe opérationnelle.

- Mise à jour et passage de version de kubernetes qui doit être fait très régulièrement car une version n'est supportée que 2 ans.
- Choix d'une configuration réseau et de sécurité adaptée.
- Installation probable de système de stockage distribué comme Ceph à maintenir également dans

- le temps.
- Etc.

## **Kubespray : intégration “officielle” de kubeadm et Ansible pour gérer un cluster**

<https://kubespray.io/#/>

En réalité utiliser kubeadm directement en ligne de commande n'est pas la meilleure approche car cela ne respecte pas l'infrastructure as code et rend plus périlleux la maintenance/maj du cluster par la suite.

Le projet kubespray est un installer de cluster kubernetes utilisant Ansible et kubeadm. C'est probablement l'une des méthodes les plus populaires pour véritablement gérer un cluster de production on premise.

Mais la encore il s'agit de ne pas sous-estimer la complexité de la maintenance (comme avec kubeadm).

## **Installer un cluster complètement à la main pour s'exercer**

On peut également installer Kubernetes de façon

encore plus manuelle pour mieux comprendre ses rouages et composants. Ce type d'installation est décrite par exemple ici : [Kubernetes the hard way](#).

## **Bibliographie pour approfondir le choix d'une distribution Kubernetes :**

- Chapitre 3 du livre Cloud Native DevOps with Kubernetes chez Oreilly

## **03 - TP1 - Installation et configuration de Kubernetes**

Au cours de nos TPs nous allons passer rapidement en revue deux manières de mettre en place Kubernetes :

- Un cluster de développement avec `minikube`
- Un cluster managed loué chez un provider (Scaleway, DigitalOcean, Azure ou Google Cloud)

Nous allons d'abord passer par la première option.

## **Découverte de Kubernetes**

### **Installer le client CLI `kubectl`**

`kubectl` est le point d'entrée universel pour contrôler

tous les types de cluster Kubernetes. C'est un client en ligne de commande qui communique en REST avec l'API d'un cluster.

Nous allons explorer kubectl au fur et à mesure des TP. Cependant à noter que :

- kubectl peut gérer plusieurs clusters/configurations et switcher entre ces configurations
- kubectl est nécessaire pour le client graphique Lens que nous utiliserons plus tard.

La méthode d'installation importe peu. Pour installer kubectl sur Ubuntu nous ferons simplement: `sudo snap install kubectl --classic`.

- Faites `kubectl version` pour afficher la version du client kubectl.

## Installer Minikube

**Minikube** est la version de développement de Kubernetes (en local) la plus répandue. Elle est maintenue par la cloud native foundation et très proche de Kubernetes upstream. Elle permet de simuler un ou plusieurs noeuds de cluster sous

forme de conteneurs docker ou de machines virtuelles.

- Pour installer minikube la méthode recommandée est indiquée ici: <https://minikube.sigs.k8s.io/docs/start/>

Nous utiliserons classiquement docker comme runtime pour minikube (les noeuds k8s seront des conteneurs simulant des serveurs). Ceci est, bien sur, une configuration de développement. Elle se comporte cependant de façon très proche d'un véritable cluster.

- Si Docker n'est pas installé, installer Docker avec la commande en une seule ligne : `curl -fsSL https://get.docker.com | sh`, puis ajoutez-vous au groupe Docker avec `sudo usermod -a -G docker <votrenom>`, et faites `sudo reboot` pour que cela prenne effet.
- Pour lancer le cluster faites simplement: `minikube start` (il est également possible de préciser le nombre de coeurs de calcul, la mémoire et d'autre paramètre pour adapter le cluster à nos besoins.)

Minikube configure automatiquement kubectl (dans

le fichier `~/.kube/config`) pour qu'on puisse se connecter au cluster de développement.

- Testez la connexion avec `kubectl get nodes`. Affichez à nouveau la version `kubectl version`. Cette fois-ci la version de kubernetes qui tourne sur le cluster actif est également affichée. Idéalement le client et le cluster devrait être dans la même version mineure par exemple `1.20.x`.

### Bash completion

Pour permettre à `kubectl` de compléter le nom des commandes et ressources avec `<Tab>` il est utile d'installer l'autocomplétion pour Bash :

```
sudo apt install bash-completion
```

```
source <(kubectl completion bash)
```

```
echo "source <(kubectl completion  
bash)" >> ${HOME}/.bashrc
```

**Vous pouvez désormais appuyer sur `<Tab>` pour compléter vos commandes `kubectl`, c'est très utile !**

## Explorons notre cluster k8s

Notre cluster k8s est plein d'objets divers, organisés entre eux de façon dynamique pour décrire des applications, tâches de calcul, services et droits d'accès. La première étape consiste à explorer un peu le cluster :

- Listez les nodes pour récupérer le nom de l'unique node (`kubectl get nodes`) puis affichez ses caractéristiques avec `kubectl describe node/minikube`.

La commande `get` est générique et peut être utilisée pour récupérer la liste de tous les types de ressources.

De même, la commande `describe` peut s'appliquer à tout objet k8s. On doit cependant préfixer le nom de l'objet par son type (ex : `node/minikube` ou `nodes minikube`) car k8s ne peut pas deviner ce que l'on cherche quand plusieurs ressources ont le même nom.

- Pour afficher tous les types de ressources à la fois que l'on utilise : `kubectl get all`

NAME	TYPE

CLUSTER-IP	EXTERNAL-IP	PORT(S)
AGE		
service/kubernetes	ClusterIP	
10.96.0.1	<none>	443/TCP
2m34s		

Il semble qu'il n'y a qu'une ressource dans notre cluster. Il s'agit du service d'API Kubernetes, pour qu'on puisse communiquer avec le cluster.

En réalité il y en a généralement d'autres cachés dans les autres namespaces. En effet les éléments internes de Kubernetes tournent eux-mêmes sous forme de services et de daemons Kubernetes. Les *namespaces* sont des groupes qui servent à isoler les ressources de façon logique et en termes de droits (avec le *Role-Based Access Control* (RBAC) de Kubernetes).

Pour vérifier cela on peut :

- Afficher les namespaces : `kubectl get namespaces`

Un cluster Kubernetes a généralement un namespace appelé `default` dans lequel les commandes sont lancées et les ressources créées si on ne précise rien. Il a également aussi un

namespace `kube-system` dans lequel résident les processus et ressources système de k8s. Pour préciser le namespace on peut rajouter l'argument `-n` à la plupart des commandes k8s.

- Pour lister les ressources liées au `kubectl get all -n kube-system`.
- Ou encore : `kubectl get all --all-namespaces` (peut être abrégé en `kubectl get all -A`) qui permet d'afficher le contenu de tous les namespaces en même temps.
- Pour avoir des informations sur un namespace :  
`kubectl describe namespace/kube-system`

## Déployer une application en CLI

Nous allons maintenant déployer une première application conteneurisée. Le déploiement est un peu plus complexe qu'avec Docker, en particulier car il est séparé en plusieurs objets et plus configurable.

- Pour créer un déploiement en ligne de commande (par opposition au mode déclaratif que nous verrons plus loin), on peut lancer par exemple:  
`kubectl create deployment`

démonstration

--image=monachus/rancher-demo.

Cette commande crée un objet de type deployment. Nous pourrons étudier ce deployment avec la commande kubectl describe deployment/démonstration.

- Notez la liste des événements sur ce déploiement en bas de la description.
- De la même façon que dans la partie précédente, listez les pods avec kubectl. Combien y en a-t-il ?
- Agrandissons ce déploiement avec kubectl scale deployment démonstration --replicas=5
- kubectl describe deployment/démonstration permet de constater que le service est bien passé à 5 replicas.
- Observez à nouveau la liste des évènements, le scaling y est enregistré...
- Listez les pods pour constater  
A ce stade impossible d'afficher l'application : le

déploiement n'est pas encore accessible de l'extérieur du cluster. Pour régler cela nous devons l'exposer grâce à un service :

- `kubectl expose deployment demonstration --type=NodePort --port=8080 --name=demonstration-service`
- Affichons la liste des services pour voir le résultat:  
`kubectl get services`

Un service permet de créer un point d'accès unique exposant notre déploiement. Ici nous utilisons le type Nodeport car nous voulons que le service soit accessible de l'extérieur par l'intermédiaire d'un forwarding de port.

Avec minikube ce forwarding de port doit être concrétisé avec la commande `minikube service demonstration-service`.

Normalement la page s'ouvre automatiquement et nous voyons notre application.

- Sauriez-vous expliquer ce que l'app fait ?
- Pour le comprendre ou le confirmer, diminuez le nombre de réplicats à l'aide de la commande utilisée précédemment pour passer à 5 réplicats. Qu

se passe-t-il ?

Une autre méthode pour accéder à un service (quel que soit son type) en mode développement est de forwarder le traffic par l'intermédiaire de kubectl (et des composants kube-proxy installés sur chaque noeuds du cluster).

- Pour cela on peut par exemple lancer: `kubectl port-forward svc/demonstration-service 8080:8080 --address 127.0.0.1`
- Vous pouvez désormais accéder à votre app via `kubectl` sur: `http://localhost:8080`. Quelle différence avec l'exposition précédente via minikube ?

=> Un seul conteneur s'affiche. En effet `kubectl port-forward` sert à créer une connexion de développement/debug qui pointe toujours vers le même pod en arrière plan.

Pour exposer cette application en production sur un véritable cluster, nous devrions plutôt avoir recours à un service de type LoadBalancer. Mais minikube ne propose pas par défaut de loadbalancer. Nous y reviendrons dans le cours sur les objets Kubernetes.

## Simplifier les lignes de commande k8s

- Pour gagner du temps on dans les commandes Kubernetes on peut définir un alias: alias kc='kubectl' (à mettre dans votre .bash\_profile en faisant echo "alias kc='kubectl'" >> ~/.bash\_profile, puis en faisant source ~/.bash\_profile).
- Vous pouvez ensuite remplacer kubectl par kc dans les commandes.
- Également pour gagner du temps en ligne de commande, la plupart des mots-clés de type Kubernetes peuvent être abrégés :
  - services devient svc
  - deployments devient deploy
  - etc.

La liste complète : <https://blog.heptio.com/kubectl-resource-short-names-heptioprotip-c8eff9fb7202>

- Essayez d'afficher les serviceaccounts (users) et les namespaces avec une commande courte.

## Une 2e installation : Mettre en place un cluster K8s managé chez le provider de

## cloud Scaleway

Je vais louer pour vous montrer un cluster kubernetes managé. Vous pouvez également louez le votre si vous préférez en créant un compte chez ce provider de cloud.

- Créez un compte (ou récupérez un accès) sur [Scaleway](#).
- Créez un cluster Kubernetes avec [l'interface Scaleway](#)

La création prend environ 5 minutes.

- Sur la page décrivant votre cluster, un gros bouton en bas de la page vous incite à télécharger ce même fichier kubeconfig (*Download Kubeconfig*).

Ce fichier contient la **configuration kubectl** adaptée pour la connexion à notre cluster.

## Une 3e installation: k3s sur votre VPS

K3s est une distribution de Kubernetes orientée vers la création de petits clusters de production notamment pour l'informatique embarquée et l'Edge computing. Elle a la caractéristique de

rassembler les différents composants d'un cluster kubernetes en un seul "binaire" pouvant s'exécuter en mode `master` (noeud du control plane) ou `agent` (noeud de calcul).

Avec K3s, il est possible d'installer un petit cluster d'un seul noeud en une commande ce que nous allons faire ici:

- Lancez dans un terminal la commande suivante:

```
curl -sfL https://get.k3s.io |  
INSTALL_K3S_EXEC="--disable=traefik"  
sh -
```

La configuration `kubectl` pour notre nouveau cluster k3s est dans le fichier `/etc/rancher/k3s/k3s.yaml` et accessible en lecture uniquement par `root`. Pour se connecter au cluster on peut donc faire (parmis d'autre méthodes pour gérer la `kubeconfig`):

- Copie de la conf `sudo cp /etc/rancher/k3s/k3s.yaml ~/.kube/k3s.yaml`
- Changer les permission `sudo chown $USER ~/.kube/k3s.yaml`
- activer cette configuration pour `kubectl` avec une

variable d'environnement: export

```
KUBECONFIG=~/.kube/k3s.yaml
```

- Tester la configuration avec kubectl get nodes qui devrait renvoyer quelque chose proche de:

NAME	AGE	VERSION	STATUS	ROLES
vnc-stagiaire-...	plane, master	10m	Ready	control-

## Merger la configuration kubectl

La/Les configurations de kubectl sont à déclarer dans la variable d'environnement KUBECONFIG.

Nous allons déclarer deux fichiers de config et les merger automatiquement.

- Téléchargeons le fichiers de configuration scaleway fourni par le formateur ou à récupérer sur votre espace Scaleway. Enregistrez le par exemple dans ~/.kube/scaleway.yaml.
- Copiez le fichier de config k3s /etc/rancher/k3s/k3s.yaml dans ~/.kube: sudo cp /etc/rancher/k3s/k3s.yaml ~/.kube/ && sudo chown stagiaire ~/.kube/k3s.yaml
- Changez la variable d'environnement pour déclarer

la config par défaut avec en plus nos deux nouvelles configs: export

```
KUBECONFIG=~/.kube/config:~/.kube  
/scaleway.yaml:~/.kube/k3s.yaml
```

- Pour afficher la configuration fusionnée des fichiers et l'exporter lancez: `kubectl config view --flatten >> ~/.kube/merged.yaml.`
- Pour sélectionner ensuite cette configuration mergée: `export KUBECONFIG='~/.kube/merged.yaml'.`
- Maintenant que nos trois configs sont fusionnées, observons l'organisation du fichier `~/.kube/config` en particulier les éléments des listes YAML de:
  - `clusters`
  - `contexts`
  - `users`
- Listez les contextes avec `kubectl config get-contexts` et affichez le contexte courant avec `kubectl config current-context`.
- Changez de contexte avec `kubectl config`

`use-context <nom_contexte>`.

- Testons quelle connexion nous utilisons avec `kubectl get nodes`.
- Observons les derniers évènements arrivés à notre cluster avec `kubectl get events --watch`.

## Au-delà de la ligne de commande...

### Accéder à la dashboard Kubernetes

Le moyen le plus classique pour avoir une vue d'ensemble des ressources d'un cluster est d'utiliser la Dashboard officielle. Cette Dashboard est généralement installée par défaut lorsqu'on loue un cluster chez un provider.

On peut aussi l'installer dans minikube ou k3s.

=> Démonstration

### Installer Lens

Lens est une interface graphique (un client “lourd”) pour Kubernetes. Elle se connecte en utilisant `kubectl` et la configuration `~/.kube/config` par défaut et nous permettra d'accéder à un dashboard puissant et agréable à utiliser.

Vous pouvez l'installer en lançant ces commandes :

```
## Install Lens
curl -L0 https://github.com/lensapp
/lens/releases/download/v4.1.4/Lens-
4.1.4.amd64.deb
sudo dpkg -i Lens-4.1.4.amd64.deb
```

- Lancez l'application Lens dans le menu “internet” de votre machine VNC
- Sélectionnez le cluster Scaleway en cliquant sur le bouton plus au lancement
- Explorons ensemble les ressources dans les différentes rubriques et namespaces

## Installer Argocd sur notre cluster k3s

Argocd est une solution de “Continuous Delivery” dédiée au **GitOps** avec Kubernetes. Elle fourni une interface assez géniale pour détecter et moniturer les ressources d'un cluster.

Nous verrons dans le TP5 comment l'installer et l'utiliser.

## 04 - Cours - Objets Kubernetes - Partie 1

# L'API et les Objets Kubernetes

Utiliser Kubernetes consiste à déclarer des objets grâce à l'API Kubernetes pour décrire l'état souhaité d'un cluster : quelles applications ou autres processus exécuter, quelles images elles utilisent, le nombre de replicas, les ressources réseau et disque que vous mettez à disposition, etc.

On définit des objets généralement via l'interface en ligne de commande et `kubectl` de deux façons :

- en lançant une commande `kubectl run <conteneur> . . . , kubectl expose . . .`
- en décrivant un objet dans un fichier YAML ou JSON et en le passant au client `kubectl apply -f monpod.yml`

Vous pouvez également écrire des programmes qui utilisent directement l'API Kubernetes pour interagir avec le cluster et définir ou modifier l'état souhaité.

**Kubernetes est complètement automatisable !**

## La commande `apply`

Kubernetes encourage le principe de l'infrastructure-as-code : il est recommandé d'utiliser une description YAML et versionnée des objets et configurations Kubernetes plutôt que la CLI.

Pour cela la commande de base est `kubectl apply -f object.yaml`.

La commande inverse `kubectl delete -f object.yaml` permet de détruire un objet précédemment appliqué dans le cluster à partir de sa description.

Lorsqu'on vient d'appliquer une description on peut l'afficher dans le terminal avec `kubectl apply -f myobj.yaml view-last-applied`. Globalement Kubernetes garde un historique de toutes les transformations des objets : on peut explorer, par exemple avec la commande `kubectl rollout history deployment`.

## Parenthèse : Le YAML

Kubernetes décrit ses ressources en YAML. A quoi ça ressemble, YAML ?

- marché :

```
lieu: Marché de la Place
jour: jeudi
horaire:
    unité: "heure"
    min: 12
    max: 20
fruits:
    - nom: pomme
        couleur: "verte"
        pesticide: avec

    - nom: poires
        couleur: jaune
        pesticide: sans

légumes:
    - courgettes
    - salade
    - potiron
```

---

## Syntaxe

- Alignement ! (**2 espaces !!**)
- ALIGNEMENT !! (comme en python)
- **ALIGNEMENT !!!** (le défaut du YAML, pas de correcteur syntaxique automatique, c'est bête mais

vous y perdrez forcément du temps !)

- des listes (tirets)
- des paires **clé: valeur**
- Un peu comme du JSON, avec cette grosse différence que le JSON se fiche de l'alignement et met des accolades et des points-virgules
- **les extensions Kubernetes et YAML dans VSCode vous aident à repérer des erreurs**

## **Syntaxe de base d'une description YAML Kubernetes**

Les descriptions YAML permettent de décrire de façon lisible et manipulable de nombreuses caractéristiques des ressources Kubernetes (un peu comme un *Compose file* par rapport à la CLI Docker).

### **Exemple**

Création d'un service simple :

```
kind: Service
apiVersion: v1
metadata:
  labels:
```

```
k8s-app: kubernetes-dashboard
  name: kubernetes-dashboard
  namespace: kubernetes-dashboard
spec:
  ports:
    - port: 443
      targetPort: 8443
  selector:
    k8s-app: kubernetes-dashboard
  type: NodePort
```

Remarques de syntaxe :

- Toutes les descriptions doivent commencer par spécifier la version d'API (minimale) selon laquelle les objets sont censés être créés
- Il faut également préciser le type d'objet avec kind
- Le nom dans metadata:\n name: value est également obligatoire.
- On rajoute généralement une description longue démarrant par spec :

## Description de plusieurs ressources

- On peut mettre plusieurs ressources à la suite dans un fichier k8s : cela permet de décrire une

installation complexe en un seul fichier

- par exemple le dashboard Kubernetes  
[https://github.com/kubernetes/dashboard  
/blob/master/aio/deploy/recommended.yaml](https://github.com/kubernetes/dashboard/blob/master/aio/deploy/recommended.yaml)
- L'ordre n'importe pas car les ressources sont décrites déclarativement c'est-à-dire que:
- Les dépendances entre les ressources sont déclarées
- Le control plane de Kubernetes se charge de planifier l'ordre correct de création en fonction des dépendances (pods avant le déploiement, rôle avec l'utilisateur lié au rôle)
- On préfère cependant les mettre dans un ordre logique pour que les humains puissent les lire.
- On peut sauter des lignes dans le YAML et rendre plus lisible les descriptions
- On sépare les différents objets par - - -

## Objets de base

### Les namespaces

Tous les objets Kubernetes sont rangés dans différents espaces de travail isolés appelés

namespaces.

Cette isolation permet 3 choses :

- ne voir que ce qui concerne une tâche particulière (ne réfléchir que sur une seule chose lorsqu'on opère sur un cluster)
- créer des limites de ressources (CPU, RAM, etc.) pour le namespace
- définir des rôles et permissions sur le namespace qui s'appliquent à toutes les ressources à l'intérieur.

Lorsqu'on lit ou créé des objets sans préciser le namespace, ces objets sont liés au namespace default.

Pour utiliser un namespace autre que default avec kubectl il faut :

- le préciser avec l'option -n : `kubectl get pods -n kube-system`
- créer une nouvelle configuration dans la kubeconfig pour changer le namespace par default.

Kubernetes gère lui-même ses composants internes sous forme de pods et services.

- Si vous ne trouvez pas un objet, essayez de lancer

la commande kubectl avec l'option -A ou --all-namespaces

## Les Pods

Un Pod est l'unité d'exécution de base d'une application Kubernetes que vous créez ou déployez. Un Pod représente des process en cours d'exécution dans votre Cluster.

Un Pod encapsule un conteneur (ou souvent plusieurs conteneurs), des ressources de stockage, **une IP réseau unique**, et des options qui contrôlent comment le ou les conteneurs doivent s'exécuter (ex: *restart policy*). Cette collection de conteneurs et volumes tournent dans le même environnement d'exécution mais les processus sont isolés.

Un Pod représente une unité de déploiement : un petit nombre de conteneurs qui sont étroitement liés et qui partagent :

- les mêmes ressources de calcul
- des volumes communs
- la même IP donc le même nom de domaine
- peuvent se parler sur localhost

- peuvent se parler en IPC
- ont un nom différent et des logs différents

Chaque Pod est destiné à exécuter une instance unique d'un workload donné. Si vous désirez mettre à l'échelle votre workload, vous devez multiplier le nombre de Pods avec un déploiement.

Pour plus de détail sur la philosophie des pods, vous pouvez consulter [ce bon article](#).

Kubernetes fournit un ensemble de commandes pour débugger des conteneurs :

- `kubectl logs <pod-name> -c <conteneur_name>` (le nom du conteneur est inutile si un seul)
- `kubectl exec -it <pod-name> -c <conteneur_name> -- bash`
- `kubectl attach -it <pod-name>`

Enfin, pour debugger la sortie réseau d'un programme on peut rapidement forwarder un port depuis un pods vers l'extérieur du cluster :

- `kubectl port-forward <pod-name> <port_interne>:<port_externe>`
- C'est une commande de debug seulement : pour

exposer correctement des processus k8s, il faut créer un service, par exemple avec NodePort.

Pour copier un fichier dans un pod on peut utiliser:

```
kubectl cp <pod-name>:</path/to/remote  
/file> </path/to/local/file>
```

Pour monitorer rapidement les ressources consommées par un ensemble de processus il existe les commandes `kubectl top nodes` et `kubectl top pods`

## Un manifeste de Pod

`rancher-demo-pod.yaml`

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: rancher-demo-pod  
spec:  
  containers:  
    - image: monachus/rancher-  
demo:latest  
      name: rancher-demo-container  
  ports:  
    - containerPort: 8080
```

```
        name: http
        protocol: TCP
    - image: redis
        name: redis-container
        ports:
            - containerPort: 6379
                name: http
                protocol: TCP
```

## Rappel sur quelques concepts

### Haute disponibilité

- Faire en sorte qu'un service ait un "uptime" élevé.  
On veut que le service soit tout le temps accessible même lorsque certaines ressources manquent :

- elles tombent en panne
- elles sont sorties du service pour mise à jour, maintenance ou modification

Pour cela on doit avoir des ressources multiples...

- Plusieurs serveurs
- Plusieurs versions des données
- Plusieurs accès réseau

Il faut que les ressources disponibles prennent automatiquement le relais des ressources indisponibles. Pour cela on utilise en particulier:

- des “load balancers” : aiguillages réseau intelligents
- des “healthchecks” : une vérification de la santé des applications

Nous allons voir que Kubernetes intègre automatiquement les principes de load balancing et de healthcheck dans l’orchestration de conteneurs

## Répartition de charge (load balancing)

- Un load balancer : une sorte d’**“aiguillage” de trafic réseau**, typiquement HTTP(S) ou TCP.
- Un aiguillage **intelligent** qui se renseigne sur plusieurs critères avant de choisir la direction.

Cas d’usage :

- Éviter la surcharge : les requêtes sont réparties sur différents backends pour éviter de les saturer.

L’objectif est de permettre la haute disponibilité : on veut que notre service soit toujours disponible, même en période de panne/maintenance.

- Donc on va dupliquer chaque partie de notre service et mettre les différentes instances derrière un load balancer.
- Le load balancer va vérifier pour chaque backend s'il est disponible (**healthcheck**) avant de rediriger le trafic.
- Répartition géographique : en fonction de la provenance des requêtes on va rediriger vers un datacenter adapté (+ proche).

## Healthchecks

Fournir à l'application une façon d'indiquer qu'elle est disponible, c'est-à-dire :

- qu'elle est démarrée (*liveness*)
- qu'elle peut répondre aux requêtes (*readiness*).

## Application microservices

- Une application composée de nombreux petits services communiquant via le réseau. Le calcul pour répondre à une requête est décomposé en différente parties distribuées entre les services. Par exemple:
- un service est responsable de la gestion des

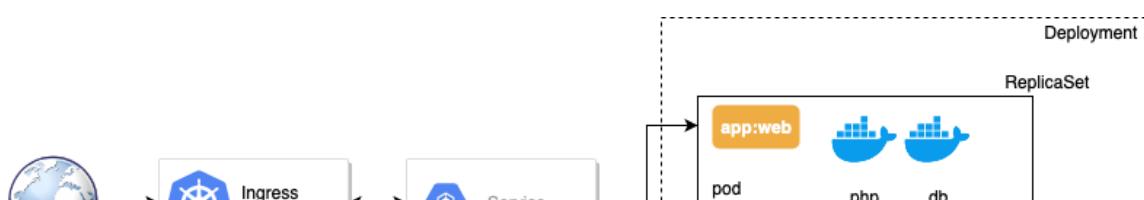
**clients** et un autre de la gestion des **commandes**.

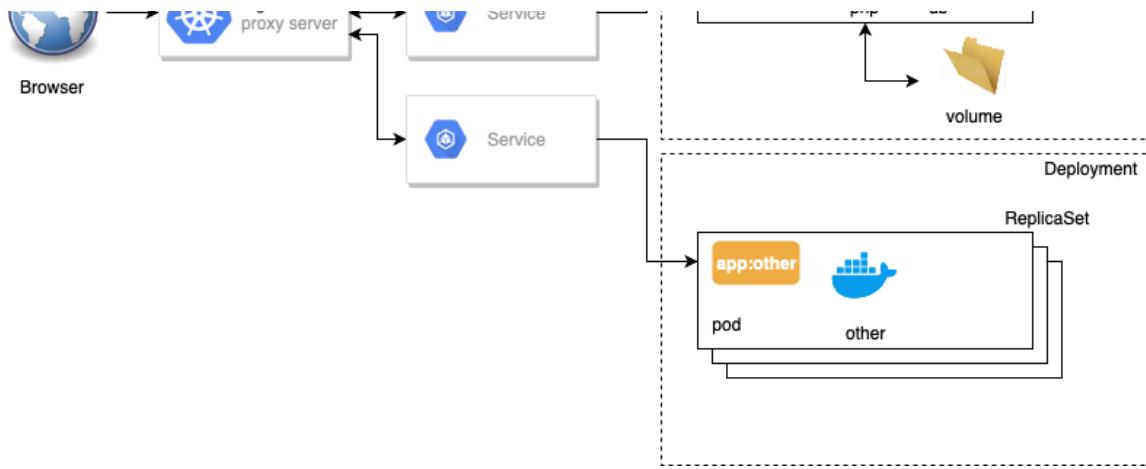
- Ce mode de développement implique souvent des architectures complexes pour être mis en oeuvre et kubernetes est pensé pour faciliter leur gestion à grande échelle.
- Imaginez devoir relancer manuellement des services vitaux pour une application en hébergeant des centaines d'instances : c'est en particulier à ce moment que kubernetes devient indispensable.

**2 exemples d'application microservices:**

- <https://github.com/microservices-patterns/ftgo-application> -> fonctionne avec le très bon livre **Microservices pattern** visible sur le readme.
- <https://github.com/GoogleCloudPlatform/microservices-demo> -> Exemple d'application microservice de référence de Google pour Kubernetes.

## **L'architecture découplée des services Kubernetes**





Comme nous l'avons vu dans le TP1, déployer une application dans kubernetes demande plusieurs étapes. En réalité en plus des **pods** l'ensemble de la gestion d'un service applicatif se décompose dans Kubernetes en 3 à 4 objets articulés entre eux:

- **replicaset**
- **deployment**
- **service**
- **(ingress)**

## Les Deployments (deploy)

Les déploiements sont les objets effectivement créés manuellement lorsqu'on déploie une application. Ce sont des objets de plus haut niveau que les **pods** et **replicaset** et les pilote pour gérer un déploiement applicatif.

## Deployment

*Updates and Rollback*

### ReplicaSet

*Self-healing, scalable, desired state*

Pod



Pod



...

Pod



*Les poupées russes Kubernetes : un Deployment contient un ReplicaSet, qui contient des Pods, qui contiennent des conteneurs*

Si c'est nécessaire d'avoir ces trois types de ressources c'est parce que Kubernetes respecte un principe de découplage des responsabilités.

La responsabilité d'un déploiement est de gérer la coexistence et le **tracking de versions** multiples d'une application et d'effectuer des montées de version automatiques en haute disponibilité en suivant une **RolloutStrategy** (CF. TP optionnel).

Ainsi lors des changements de version, un seul **deployment** gère automatiquement deux **replicaset**s contenant chacun **une version** de l'application : le découplage est nécessaire.

Un *deployment* implique la création d'un ensemble

de Pods désignés par une étiquette label et regroupé dans un **Replicaset**.

Exemple :

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

ports :

- containerPort: 80

- Pour les afficher : kubectl get deployments
- La commande kubectl run sert à créer un *deployment* à partir d'un modèle. Il vaut mieux utilisez apply -f.

## Les ReplicaSets (rs)

Dans notre modèle, les **ReplicaSet** servent à gérer et sont responsables pour:

- la réPLICATION (avoir le bon nombre d'instances et le scaling)
- la santé et le redémarrage automatique des pods de l'application (Self-Healing)
- kubectl get rs pour afficher la liste des replicas.

En général on ne les manipule pas directement (c'est déconseillé) même s'il est possible de les modifier et de les créer avec un fichier de ressource. Pour créer des groupes de conteneurs on utilise soit un **Deployment** soit d'autres formes de workloads (**DaemonSet**, **StatefulSet**, **Job**) adaptés à d'autres cas.

## Les Services

Dans Kubernetes, un **service** est un objet qui :

- Désigne un ensemble de pods (grâce à des tags) généralement géré par un déploiement.
- Fournit un endpoint réseau pour les requêtes à destination de ces pods.
- Configure une politique permettant d'y accéder depuis l'intérieur ou l'extérieur du cluster.

L'ensemble des pods ciblés par un service est déterminé par un **selector**.

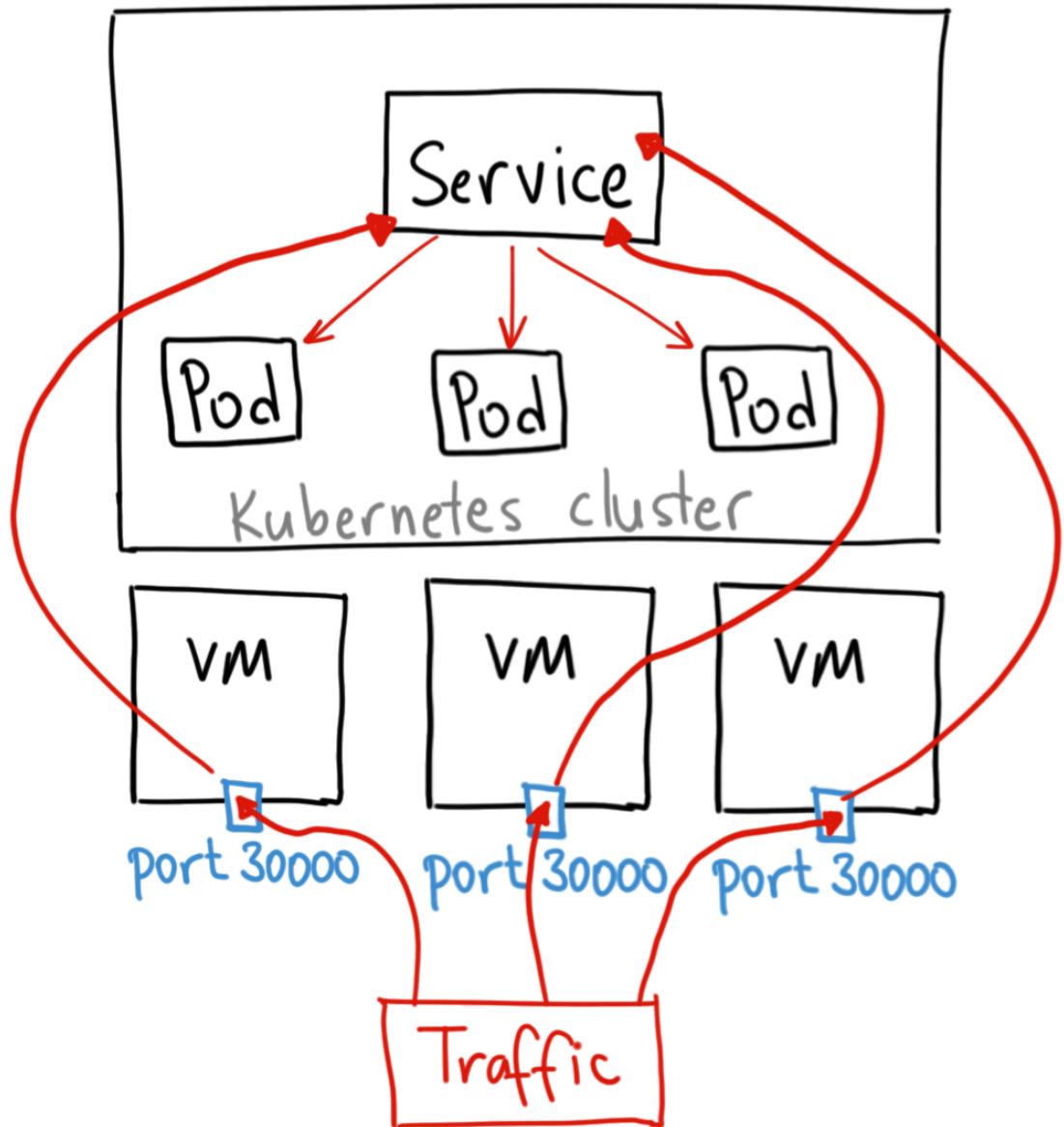
Par exemple, considérons un backend de traitement d'image (*stateless*, c'est-à-dire ici sans base de données) qui s'exécute avec 3 replicas. Ces replicas sont interchangeables et les frontends ne se soucient pas du backend qu'ils utilisent. Bien que les pods réels qui composent l'ensemble backend puissent changer, les clients frontends ne devraient pas avoir besoin de le savoir, pas plus qu'ils ne doivent suivre eux-mêmes l'état de l'ensemble des backends.

L'abstraction du service permet ce découplage : les clients frontend s'adressent à une seule IP avec

un seul port dès qu'ils ont besoin d'avoir recours à un backend. Les backends vont recevoir la requête du frontend aléatoirement.

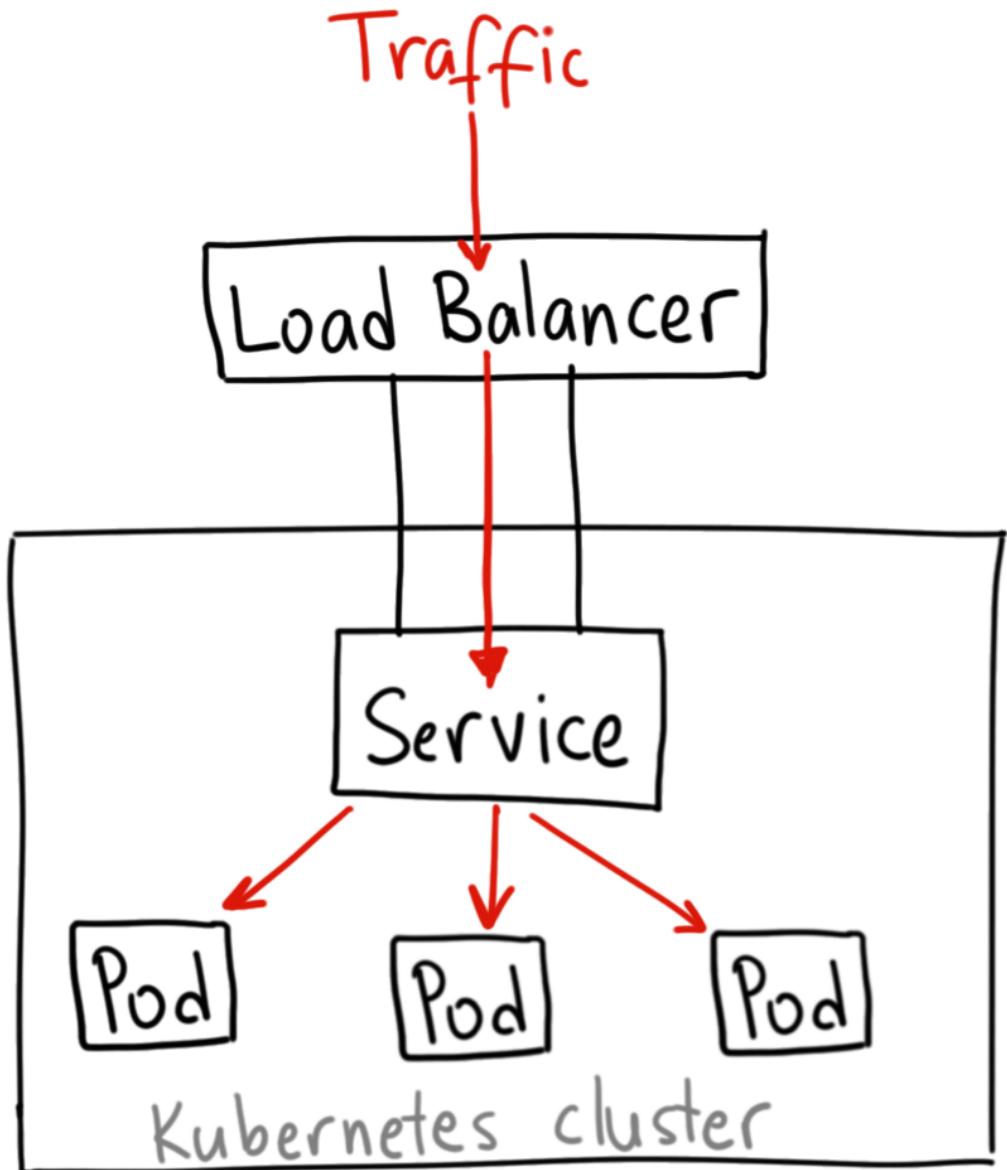
Les Services sont de trois types principaux :

- ClusterIP: expose le service **sur une IP interne** au cluster. Les autres pods peuvent alors accéder au service de l'intérieur du cluster, mais il n'est pas l'extérieur.
- NodePort: expose le service depuis l'IP de **chacun des noeuds du cluster** en ouvrant un port directement sur le nœud, entre 30000 et 32767. Cela permet d'accéder aux pods internes répliqués. Comme l'IP est stable on peut faire pointer un DNS ou Loadbalancer classique dessus.



*Crédits à [Ahmet Alp Balkan](#) pour les schémas*

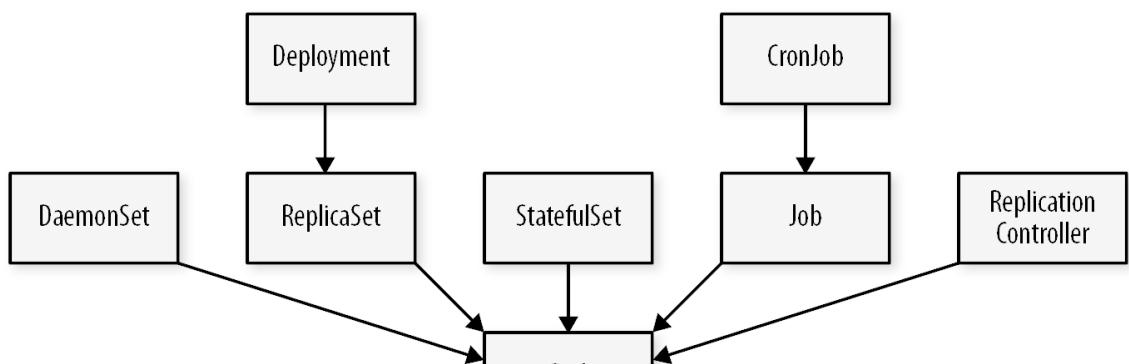
- LoadBalancer: expose le service en externe à l'aide d'un Loadbalancer de fournisseur de cloud. Les services NodePort et ClusterIP, vers lesquels le Loadbalancer est dirigé sont automatiquement créés.

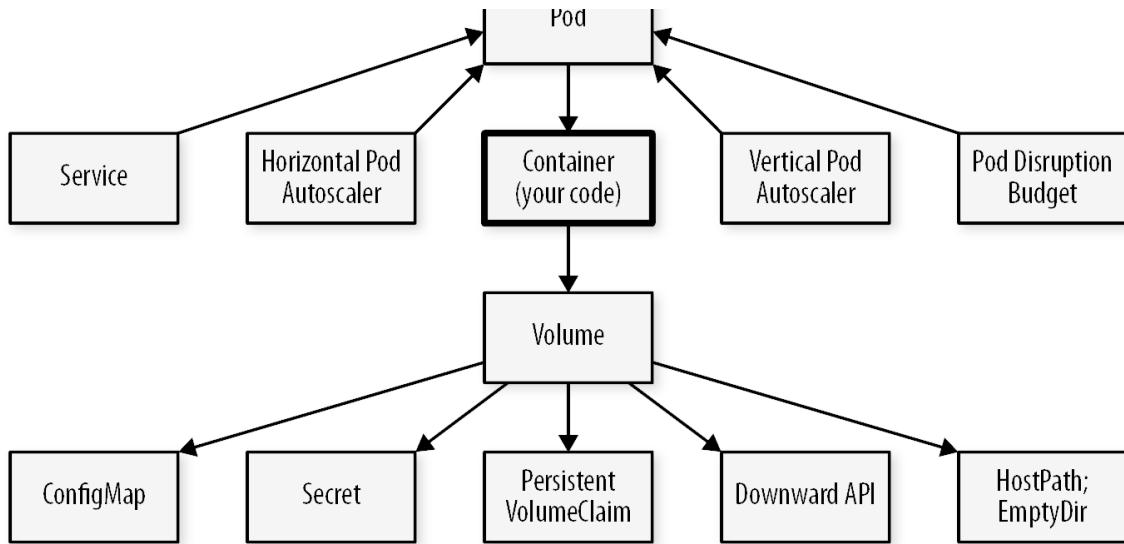


Crédits [Ahmet Alp Balkan](#)

## Les autres types de Workloads

### Kubernetes





En plus du déploiement d'un application, Il existe pleins d'autre raisons de créer un ensemble de Pods:

- **Le DaemonSet**: Faire tourner un agent ou démon sur chaque nœud, par exemple pour des besoins de monitoring, ou pour configurer le réseau sur chacun des nœuds.
- **Le Job** : Effectuer une tache unique de durée limitée et ponctuelle, par exemple de nettoyage d'un volume ou la préparation initiale d'une application, etc.
- **Le CronJob** : Effectuer une tache unique de durée limitée et récurrente, par exemple de backup ou de régénération de certificat, etc.

De plus même pour faire tourner une application, les déploiements ne sont pas toujours suffisants. En effet ils sont peu adaptés à des applications

statefull comme les bases de données de toutes sortes qui ont besoin de persister des données critiques. Pour celà on utilise un **StatefulSet** que nous verrons par la suite.

Étant donné les similitudes entre les DaemonSets, les StatefulSets et les Deployments, il est important de comprendre un peu précisément quand les utiliser.

Les **Deployments** (liés à des ReplicaSets) doivent être utilisés :

- lorsque votre application est complètement découplée du nœud
- que vous pouvez en exécuter plusieurs copies sur un nœud donné sans considération particulière
- que l'ordre de création des replicas et le nom des pods n'est pas important
- lorsqu'on fait des opérations *stateless*

Les **DaemonSets** doivent être utilisés :

- lorsqu'au moins une copie de votre application doit être exécutée sur tous les nœuds du cluster (ou sur un sous-ensemble de ces nœuds).

Les **StatefulSets** doivent être utilisés :

- lorsque l'ordre de création des replicas et le nom des pods est important
- lorsqu'on fait des opérations *stateful* (écrire dans une base de données)

## Jobs

Les jobs sont utiles pour les choses que vous ne voulez faire qu'une seule fois, comme les migrations de bases de données ou les travaux par lots. Si vous exécutez une migration en tant que Pod dans un deployment:

- Dès que la migration se finit le processus du pod s'arrête.
- Le **replicaset** qui détecte que l'"application" s'est arrêté va tenter de la redémarrer en recréant le pod.
- Votre tâche de migration de base de données se déroulera donc en boucle, en repeuplant continuellement la base de données.

## CronJobs

Comme des jobs, mais se lancent à un intervalle régulier, comme les cron sur les systèmes unix.

## 05 - TP 2 - Déployer en utilisant des fichiers ressource et Lens

Dans ce court TP nous allons redéployer notre application demonstration du TP1 mais cette fois en utilisant kubectl apply -f et en visualisant le résultat dans Lens.

N'hésitez pas aussi à observer les derniers évènements arrivés à votre cluster avec kubectl get events --watch.

- Changez de contexte pour k3s avec kubectl config use-context k3s ou kubectl config use-context default
- Chargez également la configuration de k3s dans Lens en cliquant à nouveau sur plus et en sélectionnant k3s ou default
- Commencez par supprimer les ressources demonstration et demonstration-service du TP1
- Créez un dossier TP2\_deploy\_using\_files\_and\_Lens sur le bureau de la machine distante et ouvrez le avec

VSCode.

Nous allons d'abord déployer notre application comme un simple **Pod** (non recommandé mais montré ici pour l'exercice).

- Créez un fichier `demo-pod.yaml` avec à l'intérieur le code d'exemple du cours précédent de la partie Pods.
- Appliquez le fichier avec `kubectl apply -f <fichier>`
- Constatez dans Lens dans la partie pods que les deux conteneurs du pod sont bien démarrés (deux petits carrés vert à droite de la ligne du pod)
- Modifiez le nom du pod dans la description précédente et réappliquez la configuration. Kubernetes mets à jour le nom.
- Modifier le nom du conteneur `rancher-demo` et réappliquez la configuration. Que se passe-t-il ?  
=> Kubernetes refuse d'appliquer le nouveau nom de conteneur car un pod est largement immutable.  
Pour changer d'une quelconque façon les conteneurs du pod il faut supprimer (`kubectl delete -f <fichier>`) et recréer le pod. Mais

ce travail de mise à jour devrait être géré par un déploiement pour automatiser et pour garantir la haute disponibilité de notre application démonstration.

Kubernetes fournit un ensemble de commandes pour débugger des conteneurs :

- `kubectl logs <pod-name> -c <conteneur_name>` (le nom du conteneur est inutile si un seul)
- `kubectl exec -it <pod-name> -c <conteneur_name> -- bash`
- `kubectl attach -it <pod-name>`
- Explorez le pod avec la commande `kubectl exec -it <pod-name> -c <conteneur_name> -- bash` écrite plus haut.
- Supprimez le pod.

## Avec un déploiement (méthode à utiliser)

- Créez un fichier `demo-deploy.yaml` avec à l'intérieur le code suivant à compléter:

```
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  name: demonstration
  labels:
    nom-app: demonstration
    partie: objet-deploiement
spec:
  selector:
    matchLabels:
      nom-app: demonstration
      partie: les-petits-pods-demo
  strategy:
    type: Recreate
  replicas: 1
  template:
    metadata:
      labels:
        nom-app: demonstration
        partie: les-petits-pods-demo
    spec:
      containers:
        - image: <image>
          name: <name>
      ports:
        - containerPort: <port>
```

```
name: demo-http
```

- Appliquez ce nouvel objet avec kubectl.
- Inspectez le déploiement dans Lens.
- Changez le nom d'un conteneur et réappliquez:  
Cette fois le déploiement se charge créer un nouveau pod avec les bonnes caractéristiques et de supprimer l'ancien.
- Changez le nombre de réplicats.

## Ajoutons un service en mode NodePort

- Créez un fichier demo-svc.yaml avec à l'intérieur le code suivant à compléter:

```
apiVersion: v1
kind: Service
metadata:
  name: demo-service
  labels:
    nom-app: demonstration
    partie: le-fameux-service-demo
spec:
  ports:
    - port: <port>
  selector:
```

```
nom-app: demonstration
partie: les-petits-pods-demo
type: NodePort
```

- Appliquez ce nouvel objet avec kubectl.
- Inspectez le service dans Lens.
- Visitez votre application avec l'Internal ip du noeud (à trouver dans les information du node) et le nodeport (port 3xxxx) associé au service, le nombre de réplicat devrait apparaître.
- Pour tester, changez le label du selector dans le **service** (lignes nom-app: demonstration et partie: les-petits-pods-demo à remplacer dans le fichier ) et réappliquez.
- Constatez que l'application n'est plus accessible dans le navigateur. Pourquoi ?
- Allez voir la section endpoints dans lens, constatez que quand l'étiquette est la bonne la liste des IPs des pods est présente et après la modification du selector la liste est vide (None)  
=> Les services kubernetes redirigent le trafic basés sur les étiquettes (labels) appliquées sur les pods du cluster. Il faut donc de même éviter

d'utiliser deux fois le même label pour des parties différentes de l'application.

## Solution

Le dépôt Git de la correction de ce TP est accessible ici :  
git clone -b correction\_k8s\_tp2 https://github.com/Uptime-Formation/corrections\_tp.git

## 06 - Rappels Docker

### Les Dockerfiles

- [Cours](#)
- [TP](#)

### Les volumes et les conteneurs

- [Cours](#)
- [TP](#)

Pour un exemple docker que nous allons réutiliser dans le TP3 vous pouvez cloner le code suivant:

```
git clone -b correction_k8s_tp2  
https://github.com/Uptime-Formation  
/corrections_tp.git
```

## 07 - TP 3 - Déployer des conteneurs de A à Z

Récupérez le projet de base en clonant la correction du TP2: `git clone -b exercice https://github.com/Uptime-Formation/tp3-k8s.git tp3`. On peut ouvrir une fenêtre VSCode directement dans le dossier qui nous intéresse avec : `code tp3`.

Ce TP va consister à créer des objets Kubernetes pour déployer une application microservices (plutôt simple) : `monsterstack`. Elle est composée :

- d'un front-end en Flask (Python) appelé `monstericon`,
- d'un service de backend qui génère des images (un avatar de monstre correspondant à une chaîne de caractères) appelé `dnmonster`
- et d'un datastore `redis` servant de cache pour les images de `monstericon`

Nous allons également utiliser le builder kubernetes `skaffold` pour déployer l'application en mode développement : l'image du frontend `monstericon` sera construite à partir du code

source présent dans le dossier app et automatiquement déployée dans minikube.

## **Etudions le code et testons avec docker-compose**

- Monstericon est une application web python (flask) qui propose un petit formulaire et lance une requête sur le backend pour chercher une image et l'afficher.
- Monstericon est construit à partir du Dockerfile présent dans le dossier TP3.
- Le fichier docker-compose.yml est utile pour faire tourner les trois services de l'application dans docker rapidement (plus simple que kubernetes)

Pour lancer l'application il suffit d'exécuter:

```
docker-compose up
```

Passons maintenant à Kubernetes.

## **Utiliser Kompose (facultatif)**

Explorer avec Kompose comment on peut traduire un fichier docker-compose.yml en ressources Kubernetes (ce sont les instructions à la page

suivante : <https://kubernetes.io/fr/docs/tasks/configure-pod-container/translate-compose-kubernetes/>).

En général il est recommandé de coder les ressources Kubernetes à la main comme nous allons le faire dans la partie suivante. Mais kompose peut être intéressant pour démarre un portage d'une application de docker vers kubernetes et pour bien comprendre l'équivalence des objets docker-compose et kubernetes.

Pour l'essayer installons d'abord Kompose :

```
# Linux
curl -L https://github.com/kubernetes/kompose/releases/download/v1.26.1/kompose-linux-amd64 -o kompose

chmod +x kompose
sudo mv ./kompose /usr/local/bin/kompose
```

Puis, utilisons la commande kompose convert et observons les fichiers générés. On peut ensuite faire kubectl apply avec les ressources créées à partir du fichier Compose.

# Déploiements pour le backend d'image dnmonster et le datastore redis

Maintenant nous allons également créer un déploiement pour dnmonster:

- créez dnmonster.yaml dans le dossier k8s-deploy-dev et collez-y le code suivant :

dnmonster.yaml :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dnmonster
  labels:
    app: monsterstack
spec:
  selector:
    matchLabels:
      app: monsterstack
      partie: dnmonster
  strategy:
    type: Recreate
  replicas: 5
  template:
    metadata:
```

```
labels:  
    app: monsterstack  
    partie: dnmonster  
spec:  
    containers:  
        - image: amouat/dnmonster:1.0  
          name: dnmonster  
        ports:  
            - containerPort: 8080  
              name: dnmonster
```

- Ensuite, configurons un deuxième deployment

`redis.yaml:`

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
    name: redis  
    labels:  
        app: monsterstack  
spec:  
    selector:  
        matchLabels:  
            app: monsterstack  
            partie: redis  
strategy:
```

```
    type: Recreate
replicas: 1
template:
  metadata:
    labels:
      app: monsterstack
      partie: redis
spec:
  containers:
    - image: redis:latest
      name: redis
    ports:
      - containerPort: 6379
        name: redis
```

- Installez skaffold en suivant les indications ici:  
<https://skaffold.dev/docs/install/>
- Appliquez ces ressources avec kubectl et vérifiez dans Lens que les 5 + 1 réplicats sont bien lancés.

## Déploiement du frontend monstericon

Ajoutez au fichier `monstericon.yaml` du dossier `k8s-deploy-dev` le code suivant:

```
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  name: monstericon
  labels:
    app: monsterstack
spec:
  selector:
    matchLabels:
      app: monsterstack
      partie: monstericon
  strategy:
    type: Recreate
  replicas: 3
  template:
    metadata:
      labels:
        app: monsterstack
        partie: monstericon
    spec:
      containers:
        - name: monstericon
          image: monstericon
          ports:
            - containerPort: 5000
```

L'image `monstericon` de ce déploiement n'existe

pas sur le Docker Hub, et notre Kubernetes doit pouvoir accéder à la nouvelle version de l'image construite à partir du Dockerfile. Nous allons utiliser `skaffold` pour cela. Il y a plusieurs possibilités :

- utiliser **minikube** : minikube a la capacité de se connecter au registry de notre installation Docker locale
- **sur k3s ou sur un cluster cloud** : pousser à chaque itération notre image sur un registry distant (Docker Hub)
  - pour ce faire, il faut éditer le fichier `skaffold.yaml` et le fichier de **Deployment** correspondant pour remplacer le nom de l'image `monstericon` pour faire référence à l'adresse à laquelle on souhaite pousser l'image sur le registry distant (ex:  
`docker.io/MON_COMPTE_DOCKER_HUB`  
`/monstericon`)
  - il est possible qu'il faille ajouter au même niveau que `artifacts` : dans le fichier `skaffold.yaml` ceci :
  - heureusement le mécanisme de layers des images

Docker ne nous oblige à uploader que les layers modifiés de notre image à chaque build

- (plus long) configurer un registry local (en Docker ou en Kubernetes) auquel Skaffold et Kubernetes peuvent accéder
- c'est plus long car il faut simplement configurer les certificats HTTPS ou expliciter que l'on peut utiliser un registry non sécurisé (HTTP)
- ensuite il suffit de déployer un registry tout simple (l'image officielle `registry:2`) ou plus avancé ([Harbour](#) par exemple)
- (plus avancé) utiliser Kaniko, un programme de Google qui permet de builder directement dans le cluster Kubernetes : <https://skaffold.dev/docs/pipeline-stages/builders/docker/#dockerfile-in-cluster-with-kaniko>
- Observons le fichier `skaffold.yaml`
- Lancez `skaffold run` pour construire et déployer l'application automatiquement (skaffold utilise ici le registry docker local et kubectl)

## Santé du service avec les Probes

- Ajoutons des healthchecks au conteneur dans le pod avec la syntaxe suivante (le mot-clé livenessProbe doit être à la hauteur du i de image):

```
livenessProbe:  
    tcpSocket: # si le socket est ouvert  
    c'est que l'application est démarrée  
    port: 5000  
    initialDelaySeconds: 5 # wait before  
    first probe  
    timeoutSeconds: 1 # timeout for the  
    request  
    periodSeconds: 10 # probe every 10  
    sec  
    failureThreshold: 3 # fail maximum 3  
    times  
readinessProbe:  
    httpGet:  
        path: /healthz # si l'application  
        répond positivement sur sa route  
        /healthz c'est qu'elle est prête pour  
        le traffic  
        port: 5000  
        httpHeaders:
```

```
- name: Accept  
  value: application/json  
  
initialDelaySeconds: 5  
timeoutSeconds: 1  
periodSeconds: 10  
failureThreshold: 3
```

La **livenessProbe** est un test qui s'assure que l'application est bien en train de tourner. S'il n'est pas rempli le pod est automatiquement supprimé et recréé en attendant que le test fonctionne.

Ainsi, k8s sera capable de savoir si notre conteneur applicatif fonctionne bien, quand le redémarrer.

C'est une bonne pratique pour que le **replicaset** Kubernetes sache quand redémarrer un pod et garantir que notre application se répare elle même (**self-healing**).

Cependant une application peut être en train de tourner mais indisponible pour cause de surcharge ou de mise à jour par exemple. Dans ce cas on voudrait que le pod ne soit pas détruit mais que le traffic évite l'instance indisponible pour être renvoyé vers un autre backend **ready**.

La **readinessProbe** est un test qui s'assure que l'application est prête à répondre aux requêtes en

train de tourner. S'il n'est pas rempli le pod est marqué comme non prêt à recevoir des requêtes et le service évitera de lui en envoyer.

## **Configuration d'une application avec des variables d'environnement simples**

- Notre application monstericon peut être configurée en mode DEV ou PROD. Pour cela elle attend une variable d'environnement CONTEXT pour lui indiquer si elle doit se lancer en mode PROD ou en mode DEV. Ici nous mettons l'environnement DEV en ajoutant (aligné avec la livenessProbe):

```
env:  
  - name: CONTEXT  
    value: DEV
```

## **Ajouter des indications de ressource nécessaires pour garantir la qualité de service**

- Ajoutons aussi des contraintes sur l'usage du CPU et de la RAM, en ajoutant à la même hauteur que env::

```
resources:  
  requests:
```

```
cpu: "100m" # 10% de proc  
memory: "50Mi"  
  
limits:  
  cpu: "300m" # 30% de proc  
  memory: "200Mi"
```

Nos pods auront alors **la garantie** de disposer d'un dixième de CPU (100/1000) et de 50 mégaoctets de RAM. Ce type d'indications permet de remplir au maximum les ressources de notre cluster tout en garantissant qu'aucune application ne prend toute les ressources à cause d'un fuite mémoire etc.

- Relancer `skaffold run` pour appliquer les modifications.
- Avec `kubectl describe deployment monsterverseicon`, lisons les résultats de notre `readinessProbe`, ainsi que comment s'est passée la stratégie de déploiement type : `Recreate`.

## Exposer notre stack avec des services

Les services K8s sont des endpoints réseaux qui balancent le trafic automatiquement vers un

ensemble de pods désignés par certains labels. Ils sont un peu la pierre angulaire des applications microservices qui sont composées de plusieurs sous parties elles même répliquées.

Pour créer un objet Service, utilisons le code suivant, à compléter :

```
apiVersion: v1
kind: Service
metadata:
  name: <nom_service>
  labels:
    app: monsterstack
spec:
  ports:
    - port: <port>
  selector:
    app: <app_selector>
    partie: <tier_selector>
  type: <type>
```

Ajoutez le code précédent au début de chaque fichier déploiement. Complétez pour chaque partie de notre application :

- le nom du service (name : dans metadata:) par le

nom de notre programme. En particulier, il faudra forcément appeler les services `redis` et `dnmonster` comme ça car cela permet à Kubernetes de créer les entrées DNS correspondantes. Le pod `monstericon` pourra ainsi les joindre en demandant à Kubernetes l'IP derrière `dnmonster` et `redis`.

- nom de la partie par le nom de notre programme (`monstericon`, `dnmonster` et `redis`)
- le port par le port du service
- les selectors `app` et `partie` par ceux du pod correspondant.

Le type sera : `ClusterIP` pour `dnmonster` et `redis`, car ce sont des services qui n'ont à être accédés qu'en interne, et `LoadBalancer` pour `monstericon`.

- Appliquez à nouveau avec `skaffold run`.
- Listez les services avec `kubectl get services`.
- Visitez votre application dans le navigateur avec `minikube service monstericon`.

- Supprimez l'application avec `skaffold delete`.

**Ajoutons un ingress (~ reverse proxy) pour exposer notre application en http**

- Pour **Minikube** : Installons le contrôleur Ingress Nginx avec `minikube addons enable ingress`.
- Pour les autres types de cluster (**cloud** ou **k3s**), lire la documentation sur les prérequis pour les objets Ingress et installez l'ingress controller appelé `ingress-nginx` : <https://kubernetes.io/docs/concepts/services-networking/ingress/#prerequisites>. Si besoin, aidez-vous du TP suivant sur l'utilisation de Helm.
- Avant de continuer, vérifiez l'installation du contrôleur Ingress Nginx avec `kubectl get svc -n ingress-nginx ingress-nginx-controller` : le service `ingress-nginx-controller` devrait avoir une IP externe.  
Il s'agit d'une implémentation de reverse proxy dynamique (car ciblant et s'adaptant directement aux objets services k8s) basée sur nginx configurée pour s'interfacer avec un cluster k8s.

- Repassez le service monstericon en mode ClusterIP. Le service n'est plus accessible sur un port. Nous allons utiliser l'ingress à la place pour afficher la page.
- Ajoutez également l'objet Ingress suivant dans le fichier `monster-ingress.yaml` :

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: monster-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
    - host: monsterstack.local # à changer si envie/besoin
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
```

```
service:  
  name: monstericon  
  port:  
    number: 5000
```

- Ajoutez ce fichier avec `skaffold run`.
- Récupérez l'ip de minikube avec `minikube ip`, (ou alors allez observer l'objet Ingress dans Lens dans la section Networking. Sur cette ligne, récupérez l'ip de minikube en `192.x.x.x`.).
- Ajoutez la ligne `<ip-minikube>` `monsterstack.local` au fichier `/etc/hosts` avec `sudo nano /etc/hosts` puis CRTL+S et CRTL+X pour sauver et quitter.
- Visitez la page `http://monsterstack.local` pour constater que notre Ingress (reverse proxy) est bien fonctionnel.

## Solution

Le dépôt Git de la correction de ce TP est accessible ici : `git clone -b tp3 https://github.com/Uptime-Formation/corrections_tp.git`

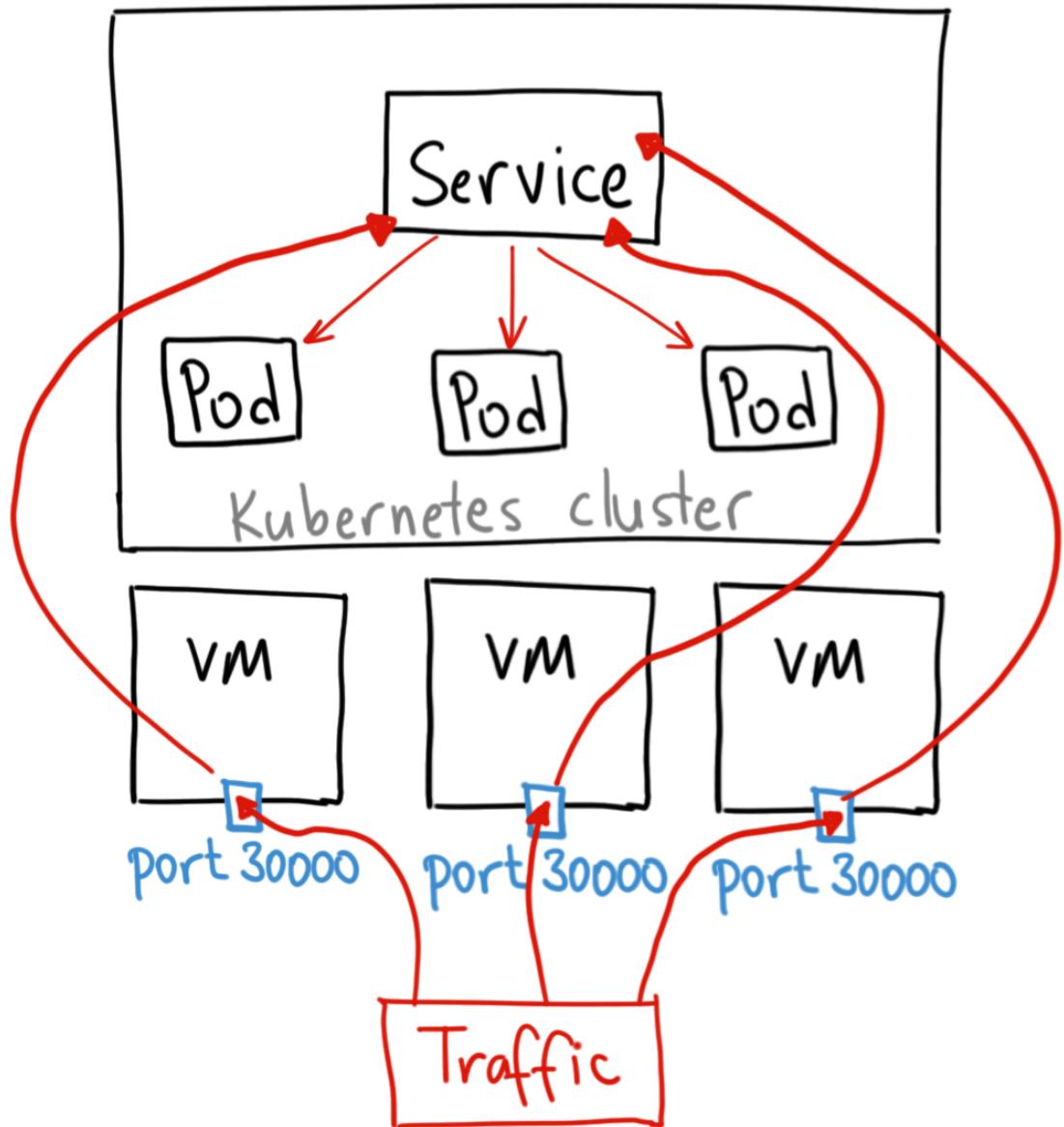
## 08 - Cours - Le réseau dans Kubernetes

Les solutions réseau dans Kubernetes ne sont pas standard. Il existe plusieurs façons d'implémenter le réseau.

### Rappel, les objets Services

Les Services sont de trois types principaux :

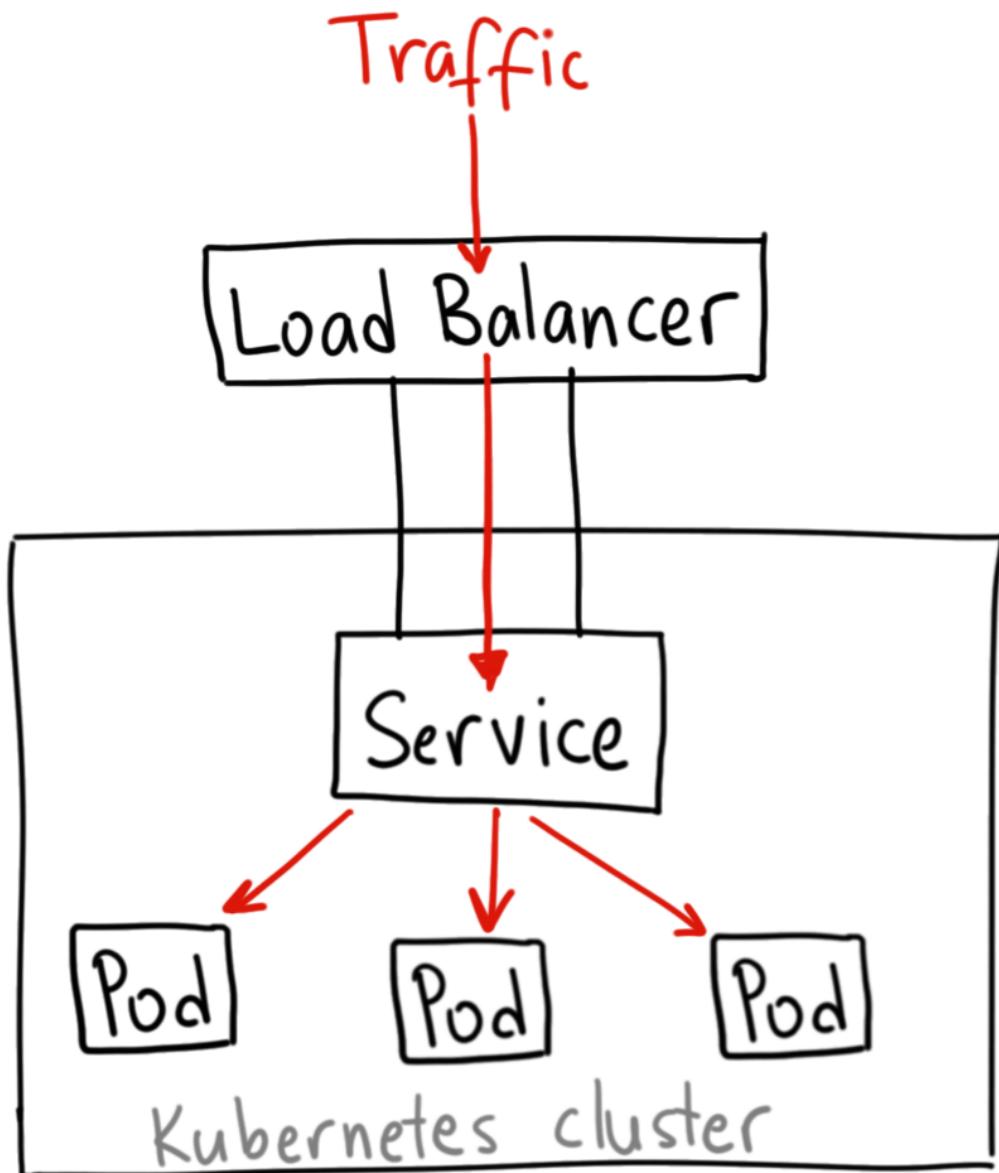
- ClusterIP: expose le service **sur une IP interne** au cluster appelée ClusterIP. Les autres pods peuvent alors accéder au service mais pas l'extérieur.
- NodePort: expose le service depuis l'IP publique de **chacun des noeuds du cluster** en ouvrant port directement sur le nœud, entre 30000 et 32767. Cela permet d'accéder aux pods internes répliqués. Comme l'IP est stable on peut faire pointer un DNS ou Loadbalancer classique dessus.
- Dans la pratique, on utilise très peu ce type de service.



*Crédits à [Ahmet Alp Balkan](#) pour les schémas*

- LoadBalancer: expose le service en externe à l'aide d'un Loadbalancer de fournisseur de cloud. Les services NodePort et ClusterIP, vers lesquels le Loadbalancer est dirigé sont automatiquement créés.
- Dans la pratique, on utilise que ponctuellement ce type de service, pour du HTTP/s on ne va pas exposer notre service (ce sera un service de type

ClusterIP) et on va utiliser à la place un objet Ingress (voir ci-dessous).



Crédits [Ahmet Alp Balkan](#)

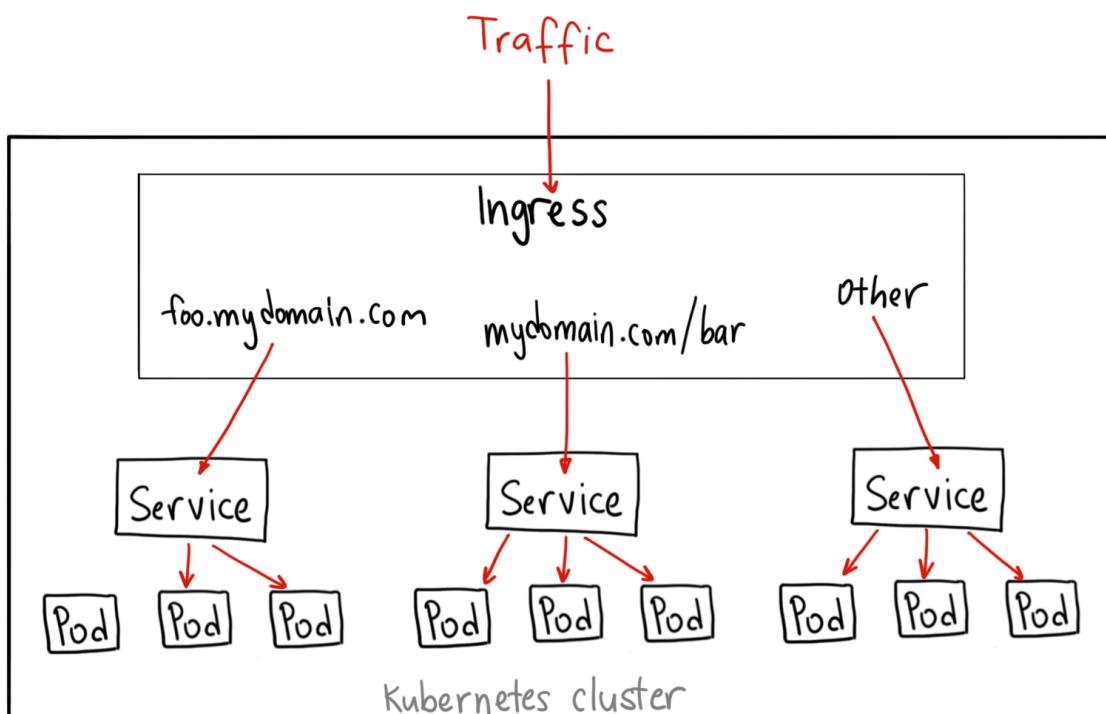
## Fournir des services LoadBalancer on premise avec MetallB

Dans un cluster managé provenant d'un fournisseur de cloud, la création d'un objet Service

Loadbalancer entraîne le provisionnement d'une nouvelle machine de loadbalancing à l'extérieur du cluster avec une IPv4 publique grâce à l'offre d'IaaS du fournisseur (impliquant des frais supplémentaires).

Cette intégration n'existe pas par défaut dans les clusters de dev comme minikube ou les clusters on-premise (le service restera pending et fonctionnera comme un NodePort). Le projet [MetaLB](#) cherche à y remédier en vous permettant d'installer un loadbalancer directement dans votre cluster en utilisant une connexion IP classique ou BGP pour la haute disponibilité.

## Les objets Ingresses



## *Crédits [Ahmet Alp Balkan](#)*

Un Ingress est un objet pour gérer dynamiquement le **reverse proxy** HTTP/HTTPS dans Kubernetes.

Documentation: <https://kubernetes.io/docs/concepts/services-networking/ingress/#what-is-ingress>

Exemple de syntaxe d'un ingress:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-wildcard-host
spec:
  rules:
    - host: "domain1.bar.com"
      http:
        paths:
          - pathType: Prefix
            path: "/bar"
        backend:
          service:
            name: service1
            port:
              number: 80
          - pathType: Prefix
```

```
    path: "/foo"
    backend:
        service:
            name: service2
            port:
                number: 80
- host: "domain2.foo.com"
  http:
    paths:
      - pathType: Prefix
        path: "/"
        backend:
            service:
                name: service3
                port:
                    number: 80
```

Pour pouvoir créer des objets ingress il est d'abord nécessaire d'installer un **ingress controller** dans le cluster:

- Il s'agit d'un déploiement conteneurisé d'un logiciel de reverse proxy (comme nginx) et intégré avec l'API de kubernetes
- Le contrôleur agit donc au niveau du protocole HTTP et doit lui-même être exposé (port 80 et 443)

à l'extérieur, généralement via un service de type LoadBalancer.

- Le contrôleur redirige ensuite vers différents services (généralement configurés en **ClusterIP**) qui à leur tour redirigent vers différents ports sur les pods selon l'URL de la requête.

Il existe plusieurs variantes d'**ingress controller**:

- Un ingress basé sur Nginx plus ou moins officiel à Kubernetes et très utilisé:

<https://kubernetes.github.io/ingress-nginx/>

- Un ingress Traefik optimisé pour k8s.
- il en existe d'autres : celui de payant l'entreprise Nginx, Contour, HAProxy...

Chaque provider de cloud et flavour de kubernetes est légèrement différent au niveau de la configuration du contrôleur ce qui peut être déroutant au départ:

- minikube permet d'activer l'ingress nginx simplement (voir TP)
- autre exemple: k3s est fourni avec traefik configuré par défaut
- On peut installer plusieurs ingress

controllers correspondant à plusieurs  
IngressClasses

Comparaison des contrôleurs: <https://medium.com/flant-com/comparing-ingress-controllers-for-kubernetes-9b397483b46b>

## Gestion dynamique des certificats à l'aide de certmanager

Certmanager est une application kubernetes (un operator) plus ou moins officielle capable de générer automatiquement des certificats TLS/HTTPS pour nos ingresses.

- Documentation d'installation: <https://cert-manager.io/docs/installation/kubernetes/>
- Tutorial pas à pas pour générer un certificat automatiquement avec un ingress et letsencrypt: <https://cert-manager.io/docs/tutorials/acme/ingress/>

Exemple de syntaxe d'un ingress utilisant certmanager:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: kuard
```

```
annotations:
  kubernetes.io/ingress.class:
"nginx"
  cert-manager.io/issuer:
"letsencrypt-prod"
spec:
  tls:
  - hosts:
    - example.example.com
      secretName: quickstart-example-tls
  rules:
  - host: example.example.com
    http:
      paths:
      - path: /
        pathType: Exact
      backend:
        service:
          name: kuard
          port:
            number: 80
```

## Le mesh networking et les *service meshes*

Un **service mesh** est un type d'outil réseau pour connecter un ensemble de pods, généralement les parties d'une application microservices de façon encore plus intégrée que ne le permet Kubernetes.

En effet opérer une application composée de nombreux services fortement couplés discutant sur le réseau implique des besoins particuliers en terme de routage des requêtes, sécurité et monitoring qui nécessite l'installation d'outils fortement dynamique autour des nos conteneurs.

Un exemple de service mesh est

<https://istio.io> qui, en ajoutant en conteneur “sidecar” à chacun des pods à supervisés, ajoute à notre application microservice un ensemble de fonctionnalités d'intégration très puissant.

## **CNI (container network interface) : Les implémentations du réseau Kubernetes**

Beaucoup de solutions de réseau qui se concurrencent, demandant un comparatif un peu fastidieux.

- plusieurs solutions très robustes
- diffèrent sur l'implémentation : BGP, réseau overlay

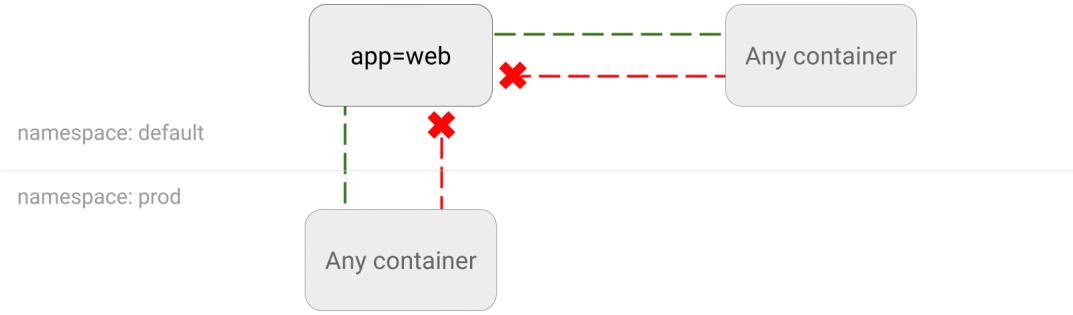
ou non (encapsulation VXLAN, IPinIP, autre)

- toutes ne permettent pas d'appliquer des **NetworkPolicies** : l'isolement et la sécurité réseau
- peuvent parfois s'hybrider entre elles (Canal = Calico + Flannel)
- ces implémentations sont souvent concrètement des *DaemonSets* : des pods qui tournent dans chacun des nodes de Kubernetes
- Calico, Flannel, Weave ou Cilium sont très employées et souvent proposées en option par les fournisseurs de cloud
- Cilium a la particularité d'utiliser la technologie eBPF de Linux qui permet une sécurité et une rapidité accrue

Comparaisons :

- <https://www.objectif-libre.com/fr/blog/2018/07/05/comparatif-solutions-reseaux-kubernetes/>
- <https://rancher.com/blog/2019/2019-03-21-comparing-kubernetes-cni-providers-flannel-calico-canal-and-weave/>

## **Les network policies : des firewalls dans le cluster**



*Crédits [Ahmet Alp Balkan](#)*

**Par défaut, les pods ne sont pas isolés au niveau réseau** : ils acceptent le trafic de n'importe quelle source.

Les pods deviennent isolés en ayant une NetworkPolicy qui les sélectionne. Une fois qu'une NetworkPolicy (dans un certain namespace) inclut un pod particulier, ce pod rejetttera toutes les connexions qui ne sont pas autorisées par cette NetworkPolicy.

- Des exemples de Network Policies : [Kubernetes Network Policy Recipes](#)

## Ressources sur le réseau

- Documentation officielle : <https://kubernetes.io/fr/docs/concepts/services-networking/service/>
- [An introduction to service meshes - DigitalOcean](#)
- [Kubernetes NodePort vs LoadBalancer vs Ingress?](#)

## When should I use what?

- Determine best networking option - Project Calico
- Doc officielle sur les solutions de networking

## **Vidéos**

Des vidéos assez complètes sur le réseau, faites par Calico :

- Kubernetes Ingress networking
- Kubernetes Services networking
- Kubernetes networking on Azure

Sur MetalLB, les autres vidéos de la chaîne sont très bien :

- Why you need to use MetalLB - Adrian Goins

## **09 - TP 4 - Déployer Wordpress Avec une base de donnée persistante**

## **Déployer Wordpress et MySQL avec du stockage et des Secrets**

Nous allons suivre ce tutoriel pas à pas :

<https://kubernetes.io/docs/tutorials/stateful-application/mysql-wordpress-persistent-volume/>

Il faut :

- Créez un projet TP4.
- Créer la `kustomization.yaml` avec le générateur de secret.
- Copier les 2 fichiers dans le projet.
- Les ajouter comme resources à la `kustomization.yaml`.

Commentons un peu le contenu des deux fichier `mysql-deployment.yaml` et `wordpress-deployment.yaml`.

- Vérifier que le stockage et le secret ont bien fonctionnés.
- Exposez et visitez le service avec minikube  
service `wordpress`. Faite la configuration de base de `wordpress`.

### **Observer le déploiement du secret à l'intérieur des pods**

- Entrez dans le pod de `mysql` grâce au terminal de Lens.
- Cherchez la variable d'environnement

`MYSQL_ROOT_PASSWORD` à l'aide des commandes  
`env | grep MYSQL`. Le conteneur mysql a utilisé cette variable accessible de lui seul pour se configurer.

## Observez la persistence

- Supprimez uniquement les deux déploiements.
- redéployez à nouveau avec `kubectl apply -k ..`, les deux déploiements sont recréés.
- En rechargeant le site on constate que les données ont été conservées.
- Allez observer la section stockage dans Lens.  
Commentons ensemble.
- Supprimer tout avec `kubectl delete -k ..`  
Que s'est-il passé ? (côté storage)

En l'état les `PersistentVolumes` générés par la combinaison du `PersistentVolumeClaim` et de la `StorageClass` de minikube sont également supprimés en même temps que les PVCs. Les données sont donc perdues et au chargement du site on doit relancer l'installation.

Pour éviter cela il faut avec une `Reclaim Policy`

à retain (conserver) et non delete comme suit <https://kubernetes.io/docs/tasks/administer-cluster/change-pv-reclaim-policy/>. Les volumes sont alors conservées et les données peuvent être récupérées manuellement. Mais les volumes ne peuvent pas être reconnectés à des PVCs automatiquement.

- Pour récupérer les données on peut monter le PV manuellement dans un pod
- Utiliser la nouvelle fonctionnalité de clone de volume

## **Pour aller plus loin**

- 

# **10 - Cours - Objets Kubernetes Partie 2.**

## **Le stockage dans Kubernetes**

### **Les Volumes Kubernetes**

Comme dans Docker, Kubernetes fournit la possibilité de mondes volumes virtuels dans les conteneurs de nos pod. On liste séparément les volumes de notre pod puis on les monte une ou

plusieurs dans les différents conteneurs Exemple:

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
    - image: k8s.gcr.io/test-webserver
      name: test-container
      volumeMounts:
        - mountPath: /test-pd
          name: test-volume
  volumes:
    - name: test-volume
      hostPath:
        # chemin du dossier sur l'hôte
        path: /data
        # ce champ est optionnel
        type: Directory
```

La problématique des volumes et du stockage est plus compliquée dans kubernetes que dans docker car k8s cherche à répondre à de nombreux cas d'usages. [doc officielle](#). Il y a donc de nombreux types de volumes kubernetes correspondants à des usages de base et aux solutions proposées par les

principaux fournisseurs de cloud.

Mentionnons quelques d'usage de base des volumes:

- `hostPath`: monte un dossier du noeud où est planifié le pod à l'intérieur du conteneur.
- `local`: comme `hostPath` mais conscient de la situation physique du volume sur le noeud et à combiner avec les placements de pods avec `nodeAffinity`
- `emptyDir`: un dossier temporaire qui est supprimé en même temps que le pod
- `configMap`: pour monter des fichiers de configurations provenant du cluster à l'intérieur des pods
- `secret`: pour monter un secret (configuration) provenant du cluster à l'intérieur des pods
- `cephfs`: monter un volume ceph provenant d'un ceph installé sur le cluster
- etc.

En plus de la gestion manuelle des volumes avec les options précédentes, Kubernetes permet de provisionner dynamiquement du stockage en

utilisant des plugins de création de volume grâce à 3 types d'objets: **StorageClass**, **PersistentVolume** et **PersistentVolumeClaim**.

## **Les types de stockage avec les StorageClasses**

Le stockage dynamique dans Kubernetes est fourni à travers des types de stockage appelés **StorageClasses** :

- dans le cloud, ce sont les différentes offres de volumes du fournisseur,
- dans un cluster auto-hébergé c'est par exemple des opérateurs de stockage comme `rook.io` ou `longhorn(Rancher)`.

[doc officielle](#)

## **Demander des volumes et les liens aux pods : PersistentVolumes et PersistentVolumeClaims**

Quand un conteneur a besoin d'un volume, il crée une **PersistentVolumeClaim** : une demande de

volume (persistant). Si un des objets *StorageClass* est en capacité de le fournir, alors un *PersistentVolume* est créé et lié à ce conteneur : il devient disponible en tant que volume monté dans le conteneur.

- les *StorageClasses* fournissent du stockage
- les conteneurs demandent du volume avec les *PersistentVolumeClaims*
- les *StorageClasses* répondent aux *PersistentVolumeClaims* en créant des objets *PersistentVolumes* : le conteneur peut accéder à son volume.

### [doc officielle](#)

Le provisioning de volume peut être manuelle (on crée un objet *PersistentVolume* ou non la *PersistentVolumeClaim* mène directement à la création d'un volume persistant si possible)

### **Des déploiements plus stables et précautionneux : les *StatefulSets***

L'objet *StatefulSet* est relativement récent dans Kubernetes.

On utilise les Statefulsets pour répliquer un ensemble de pods dont l'état est important : par exemple, des pods dont le rôle est d'être une base de données, manipulant des données sur un disque.

Un objet StatefulSet représente un ensemble de pods dotés d'identités uniques et de noms d'hôtes stables. Quand on supprime un StatefulSet, par défaut les volumes liés ne sont pas supprimés.

Les StatefulSets utilisent un nom en commun suivi de numéros qui se suivent. Par exemple, un StatefulSet nommé web comporte des pods nommés web-0, web-1 et web-2. Par défaut, les pods StatefulSet sont déployés dans l'ordre et arrêtés dans l'ordre inverse (web-2, web-1 puis web-0).

En général, on utilise des StatefulSets quand on veut :

- des identifiants réseau stables et uniques
- du stockage stable et persistant
- des déploiements et du scaling contrôlés et dans un ordre défini

- des rolling updates dans un ordre défini et automatisées

Article récapitulatif des fonctionnalités de base pour applications stateful: <https://medium.com/capital-one-tech/conquering-statefulness-on-kubernetes-26336d5f4f17>

## Paramétrer ses Pods

### Les ConfigMaps

D'après les recommandations de développement [12factor](#), la configuration de nos programmes doit venir de l'environnement. L'environnement est ici Kubernetes.

Les objets ConfigMaps permettent d'injecter dans des pods des ensemble clés/valeur de configuration en tant que volumes/fichiers de configuration ou variables d'environnement.

### les Secrets

Les Secrets se manipulent comme des objets ConfigMaps, mais ils sont chiffrés et faits pour stocker des mots de passe, des clés privées, des certificats, des tokens, ou tout autre élément de

config dont la confidentialité doit être préservée. Un secret se créé avec l'API Kubernetes, puis c'est au pod de demander à y avoir accès.

Il y a 3 façons de donner un accès à un secret :

- le secret est un fichier que l'on monte en tant que volume dans un conteneur (pas nécessairement disponible à l'ensemble du pod). Il est possible de ne jamais écrire ce secret sur le disque (volume `tmpfs`).
- le secret est une variable d'environnement du conteneur.

Pour définir qui et quelle app a accès à quel secret, on peut utiliser les fonctionnalités “RBAC” de Kubernetes.

## **Lier utilisateurs et autorisations: Le Role-Based Access Control (RBAC)**

Kubernetes intègre depuis quelques versions un système de permissions fines sur les ressources et les namespaces. Il fonctionne en liant des ensembles de permissions appelées Roles à des identités/comptes humains appelés User ou des comptes de services pour vos programmes appelés

## ServiceAccount.

Exemple de comment générer un certificat à créer un nouvel utilisateur dans minikube:

<https://docs.bitnami.com/tutorials/configure-rbac-in-your-kubernetes-cluster/>

Doc officielle: <https://kubernetes.io/docs/reference/access-authn-authz/authentication/>

## Roles et ClusterRoles + bindings

Une **role** est un objet qui décrit un ensemble d'actions permises sur certaines ressources et s'applique sur **un seul namespace**. Pour prendre un exemple concret, voici la description d'un roles qui autorise la lecture, création et modification de pods et de services dans le namespace par défaut:

```
kind: Role
apiVersion:
rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-and-services
rules:
- apiGroups: [""]
```

```
resources: ["pods", "services"]
verbs: ["create", "delete", "get",
"list", "patch", "update", "watch",
"proxy"]
```

- Un role est une liste de règles rules
- Les rules sont décrites à l'aide de 8 verbes différents qui sont ceux présent dans le role d'exemple au dessus qu'ont associe à une liste d'objets.
- Le role **ne fait rien par lui même** : il doit être appliqué à une identité ie un User ou ServiceAccount.
- Classiquement on crée des Roles comme admin ou monitoring qui désignent un ensemble de permission consistante pour une tâche donnée.
- Notre role exemple est limité au namespace default. Pour créer des permissions valable pour tout le cluster on utilise à la place un objet appelé un ClusterRole qui fonctionne de la même façon mais indépendamment des namespace.
- Les Roles et ClusterRoles sont ensuite appliqués aux ServicesAccounts à l'aide respectivement de RoleBinding et

ClusterRoleBinding comme l'exemple suivant:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  namespace: default
  name: pods-and-services
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: alice
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: mydevs
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: pod-and-services
```

En plus des rôles que vous pouvez créer pour les utilisateur·ices et processus de votre cluster, il existe déjà dans kubernetes un ensemble de ClusterRoles prédéfinis qui sont affichables avec :

```
kubectl get clusterroles
```

La plupart de ces rôles intégrés sont destinés au `kube-system`, c'est-à-dire aux processus internes du cluster.

Cependant quatre rôles génériques existent aussi par défaut :

- Le rôle `cluster-admin` fournit un accès complet à l'ensemble du cluster.
- Le rôle `admin` fournit un accès complet à un espace de noms précis.
- Le rôle `edit` permet à un·e utilisateur·ice de modifier des choses dans un espace de noms.
- Le rôle `view` permet l'accès en lecture seule à un espace de noms.

La commande `kubectl auth can-i <verb> <type_de_resource>` permet de déterminer selon le profil utilisé (défini dans votre `kubeconfig`) les permissions actuelles de l'user sur les objets Kubernetes.

## **11 - Cours - Helm, le gestionnaire de paquets Kubernetes et les Opérateurs**

Nous avons vu que dans Kubernetes la

configuration de nos services / applications se fait généralement via de multiples fichiers YAML.

Les kustomizations permettent de rassembler ces descriptions en dossier de code et ont pas mal d'avantages mais on a vite besoin de quelque chose de plus puissant.

- Pour s'adapter à plein de paramétrages différents de notre application
- Pour éviter la répétition de code

C'est donc "trop" déclaratif en quelque sorte, et il faut se concentrer sur les quelques propriétés que l'on souhaite créer ou modifier,

## **Helm**

Pour pallier ce problème, il existe un utilitaire appelé Helm, qui produit les fichiers de déploiement que l'on souhaite.

Helm est le package manager recommandé par Kubernetes, il utilise les fonctionnalités de templating du langage Go.

Helm permet donc de déployer des applications / stacks complètes en utilisant un système de templating et de dépendances, ce qui permet

d'éviter la duplication et d'avoir ainsi une arborescence cohérente pour nos fichiers de configuration.

Mais Helm propose également :

- la possibilité de mettre les Charts dans un répertoire distant (Git, disque local ou partagé...), et donc de distribuer ces Charts publiquement.
- un système facilitant les Updates et Rollbacks de vos applications.

Il existe des sortes de *stores* d'applications Kubernetes packagées avec Helm, le plus gros d'entre eux est [Kubeapps Hub](#), maintenu par l'entreprise Bitnami qui fournit de nombreuses Charts assez robustes.

Si vous connaissez Ansible, un chart Helm est un peu l'équivalent d'un rôle Ansible dans l'écosystème Kubernetes.

## Concepts

Les quelques concepts centraux de Helm :

- Un package Kubernetes est appelé **Chart** dans Helm.
- Un Chart contient un lot d'informations nécessaires

pour créer une application Kubernetes :

- la **Config** contient les informations dynamiques concernant la configuration d'une **Chart**
- Une **Release** est une instance existante sur le cluster, combinée avec une **Config** spécifique.

### Quelques commandes Helm:

Voici quelques commandes de bases pour Helm :

- `helm repo add bitnami https://charts.bitnami.com/bitnami`: ajouter un repo contenant des charts
- `helm search repo bitnami`: rechercher un chart en particulier
- `helm install my-release my-chart --values=myvalues.yaml` : permet d'installer le chart my-chart avec le nom my-release et les valeurs de variable contenues dans myvalues.yaml (elles écrasent les variables par défaut)
- `helm upgrade my-release my-chart` : permet de mettre à jour notre release avec une nouvelle version.
- `helm ls`: Permet de lister les Charts installés sur

votre Cluster

- `helm delete my-release`: Permet de désinstaller la release `my-release` de Kubernetes

## **La configuration d'un Chart: des templates d'objets Kubernetes**

Visitons un exemple de Chart : [minecraft](#)

On constate que Helm rassemble des fichiers de descriptions d'objets k8s avec des variables (moteur de templates de Go) à l'intérieur, ce qui permet de factoriser le code et de gérer puissamment la différence entre les versions.

## **Kubernetes API et extension par APIgroups**

Tous les types de resources Kubernetes correspondent à un morceau (un sous arbre) d'API REST de Kubernetes. Ces chemins d'API pour chaque ressources sont classés par groupe qu'on appelle des `apiGroups`:

- On peut lister les resources et leur groupes d'API avec la commande `kubectl api-resources -o wide`.
- Ces groups correspondent aux préfixes indiqué

dans la section `apiVersion` des descriptions de ressources.

- Ces groupes d'API sont versionnés sémantiquement et classés en `alpha` `beta` et `stable`. `beta` indique déjà un bon niveau de stabilité et d'utilisabilité et beaucoup de ressources officielles de kubernetes ne sont pas encore en `api` `stable`. Exemple: les CronJobs viennent de se stabiliser au début 2021.
- N'importe qui peut développer ses propres types de resources appelées `CustomResourceDefinition` (voir ci dessous) et créer un `apiGroup` pour les ranger.

Documentation: [https://kubernetes.io  
/docs/reference/using-api/](https://kubernetes.io/docs/reference/using-api/)

## **Operators et Custom Resources Definitions (CRD)**

Un opérateur est :

- un morceau de logique opérationnelle de votre infrastructure (par exemple: la mise à jour votre logiciel de base de donnée stateful comme cassandra ou elasticsearch) ...

- ... implémenté dans kubernetes par un/plusieurs conteneur(s) “controller” ...
- ... contrôlé grâce à une extension de l’API Kubernetes sous forme de nouveaux type d’objets kubernetes personnalisés (de haut niveau) appelés *CustomResourcesDefinition* ...
- ... qui crée et supprime des resources de base Kubernetes comme résultat concrète.

Les opérateurs sont un sujet le plus *méta* de Kubernetes et sont très à la mode depuis leur démocratisation par Red Hat pour la gestion automatique de base de données.

- Ils peuvent être développés avec un framework Go ou Ansible
- Ils sont généralement répertoriés sur le site:  
<https://operatorhub.io/>

Exemples :

- L’opérateur Prometheus permet d’automatiser le monitoring d’un cluster et ses opérations de maintenance.
- La chart officielle de la suite Elastic (ELK) définit des objets de type `elasticsearch`

- KubeVirt permet de rajouter des objets de type VM pour les piloter depuis Kubernetes
- Azure propose des objets correspondant à ses ressources du cloud Azure, pour pouvoir créer et paramétrer des ressources Azure directement via la logique de Kubernetes.



## **Limites des opérateurs**

Il est possible de développer soit même des opérateurs mais il s'agit de développement complexes qui devraient être entrepris par les développeurs du logiciel et qui sont surtout utiles pour des applications distribuées et stateful. Les opérateurs n'ont pas forcément vocation à remplacer les Charts Helm comme on l'entend parfois.

Voir : <https://thenewstack.io/kubernetes-when-to-use-and-when-to-avoid-the-operator-pattern/>

## 12 - TP 5 - Déployer Wordpress avec Helm et ArgoCD

Helm est un “gestionnaire de paquet” ou vu autrement un “outil de templating avancé” pour k8s qui permet d’installer des applications sans faire des copier-coller pénibles de YAML :

- Pas de duplication de code
- Possibilité de créer du code générique et flexible avec pleins de paramètres pour le déploiement.
- Des déploiements avancés avec plusieurs étapes

Inconvénient: Helm ajoute souvent de la complexité non nécessaire car les Charts sur internet sont très paramétrables pour de multiples cas d’usage (plein de code qui n’est utile que dans des situations spécifiques).

Helm ne dispense pas de maîtriser l’administration de son cluster.

### Installer Helm

- Pour installer Helm sur Ubuntu, utilisez : sudo

```
sudo snap install helm --classic
```

- Suivez le Quickstart : <https://helm.sh/docs/intro/quickstart/>

## Autocomplete

```
helm completion bash | sudo tee  
/etc/bash_completion.d/helm et relancez  
votre terminal.
```

## Utiliser un chart Helm pour installer Wordpress

- Cherchez Wordpress sur <https://artifacthub.io/>.
- Prenez la version de **Bitnami** et ajoutez le dépôt avec la première commande à droite (ajouter le dépôt et déployer une release).
- Installer une “**release**” wordpress-tp de cette application (ce chart) avec `helm install wordpress-tp bitnami/wordpress`
- Des instructions sont affichées dans le terminal pour trouver l’IP et afficher le login et password de notre installation. La commande pour récupérer l’IP ne fonctionne que dans les cluster proposant une intégration avec un loadbalancer et fournissant

donc des IP externe. Dans minikube (qui ne fournit pas de loadbalancer) il faut à la place lancer `minikube service wordpress-tp` pour y accéder avec le NodePort.

- Notre Wordpress est prêt. Connectez-vous-y avec les identifiants affichés (il faut passer les commandes indiquées pour récupérer le mot de passe stocké dans un secret k8s).

Vous pouvez constater que l'utilisateur est par default `user` ce qui n'est pas très pertinent. Un chart prend de nombreux paramètres de configuration qui sont toujours listés dans le fichier `values.yaml` à la racine du Chart.

On peut écraser certains de ces paramètres dans un nouveau fichier par exemple `myvalues.yaml` et installer la release avec l'option `--values=myvalues.yaml`.

- Désinstallez Wordpress avec `helm uninstall wordpress-tp`

## **Utiliser la fonction `template` de Helm pour étudier les ressources d'un Chart**

- Visitez le code des charts de votre choix en clonant

le répertoire Git des Charts officielles Bitnami et en l'explorant avec VSCode :

```
git clone https://github.com/bitnami  
/charts/  
code charts
```

- Regardez en particulier les fichiers templates et le fichier de paramètres values.yaml.
- Comment modifier l'username et le password wordpress à l'installation ? il faut donner comme paramètres le yaml suivant:

```
wordpressUsername: <votrenom>  
wordpressPassword: <easytoguesspasswd>
```

- Nous allons paramétriser plus encore l'installation. Créez un dossier TP5 avec à l'intérieur un fichier values.yaml contenant:

```
wordpressUsername: <stagiaire> #  
replace  
wordpressPassword: myunsecurepassword  
wordpressBlogName: Kubernetes example  
blog  
  
replicaCount: 1
```

```
service:
  type: ClusterIP

ingress:
  enabled: true
  hostname: wordpress.
<stagiaire>.formation.dopl.uk #
replace with your hostname pointing on
the cluster ingress loadbalancer IP
  tls: true
  certManager: true
  annotations:
    cert-manager.io/cluster-issuer:
letsencrypt-prod
    kubernetes.io/ingress.class: nginx
```

- En utilisant ces paramètres, plutôt que d'installer le chart, nous allons faire le rendu (templating) des fichiers ressource générés par le chart:  
helm template wordpress-tp  
bitnami/wordpress --values=values.yaml  
> wordpress-tp-manifests.yaml.  
  
On peut maintenant lire dans ce fichier les objets kubernetes déployés par le chart et ainsi apprendre de nouvelles techniques et syntaxes. En le

parcourant on peut constater que la plupart des objets abordés pendant cette formation y sont présent plus certains autres.

## Installer le ingress NGINX et ArgoCD

Voir le [TP Gitlab et ArgoCD](#).

### ArgoCD pour installer et visualiser en live les ressources de notre chart

Argocd permet d'installer des applications qui peuvent être soit des dossiers de manifestes kubernetes simple, soit des dossiers contenant une kustomization.yaml soit des charts Helm. Une application Argocd peut être créée dans l'interface web ou être déclarée elle-même grâce à un fichier manifeste de type:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
```

Ce n'est pas une ressource de base mais bien une CustomResourceDefinition car ArgoCD est un opérateur d'applications. Nous allons créer un tel manifeste.

- Ouvrez le fichier wordpress-chart-argocd-

app.yaml et collez à l'intérieur:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: wordpress
  namespace: argocd
spec:
  destination:
    namespace: default
    server:
      https://kubernetes.default.svc
  project: default
  source:
    repoURL:
      https://charts.bitnami.com/bitnami
    chart: wordpress
    targetRevision: 11.0.5
    helm:
      values:
        wordpressUsername: elie
        wordpressPassword:
          myunsecurepassword
        wordpressBlogName: Kubernetes
example blog
```

```
replicaCount: 1

service:
  type: ClusterIP

ingress:
  enabled: true
  hostname:
    wordpress.elie.formation.dopl.uk
    pathType: Prefix
    tls: true
    certManager: true
  annotations:
    cert-manager.io/cluster-
  issuer: letsencrypt-prod

kubernetes.io/ingress.class: nginx
```

- Appliquez ce fichier avec kubectl apply -f.
- Visitez la page <https://argocd.<votrelogin>.formation.dopl.uk>.  
Une application wordpress est apparue
- Visitez la, en particulier les desired manifests de quelques resources.

- Synchronisez l'application pour installer le chart avec Sync.

En une minute ou deux, l'application est installée et l'ingress avec son certificat devrait être généré.

Vous pouvez visiter le blog à l'adresse:

`https://wordpress.<votrelogin>.formation.dop1.uk`

## Solution

Le dépôt Git contenant la correction de ce TP et des précédents est accessible avec cette commande : `git clone -b all_corrections https://github.com/Uptime-Formation/corrections_tp.git`

## TP opt. - CI/CD avec Gitlab et ArgoCD

### Installation d'un cluster avec argoCD

ArgoCD est un outil de GitOps extrêmement pratique et puissant.

Qu'est-ce que le GitOps: <https://www.objectif-libre.com/fr/blog/2019/12/17/gitops-tour-horizon-pratiques-outils/>

Mais, dans notre cas, avec un ingress NGINX, il nécessite un accès HTTPS et donc la génération d'un certificat. Une solution est de l'installer dans un cluster accessible publiquement (avec un IP publique) pour pouvoir générer un certificat avec cert-manager et un Challenge ACME HTTP 101. C'est le cas de notre cluster k3s.

Vos serveurs VNC qui sont aussi désormais des clusters k3s ont déjà plusieurs sous-domaines configurés: <votrelogin>. <sousdomaine>.dopl.uk et \*.<votrelogin>. <sousdomaine>.dopl.uk. Le sous domaine argocd.<login>.<sousdomaine>.dopl.uk pointe donc déjà sur le serveur (Wildcard DNS).

Ce nom de domaine va nous permettre de générer un certificat HTTPS pour notre application web argoCD grâce à un ingress nginx, le cert-manager de k8s et letsencrypt (challenge HTTP101).

## Installer le ingress NGINX dans k3s

- Installer l'ingress nginx avec la commande:

```
kubectl apply -f
```

```
https://raw.githubusercontent.com
```

/kubernetes/ingress-nginx/controller-v1.1.0/deploy/static/provider/cloud/deploy.yaml (pour autres méthodes ou problèmes voir : <https://kubernetes.github.io/ingress-nginx/deploy/>)

- Vérifiez l'installation avec kubectl get svc -n ingress-nginx ingress-nginx-controller : le service ingress-nginx-controller devrait avoir une IP externe.

## Installer Cert-manager dans k3s

- Pour installer cert-manager lancez : kubectl apply -f <https://github.com/jetstack/cert-manager/releases/download/v1.6.1/cert-manager.yaml>
- Il faut maintenant créer une ressource de type ClusterIssuer pour pourvoir émettre (to issue) des certificats.
- Créez une ressource comme suit (soit dans Lens avec + soit dans un fichier à appliquer ensuite avec kubectl apply -f):

```
apiVersion: cert-manager.io/v1
```

```
kind: ClusterIssuer
metadata:
  name: letsencrypt-prod
spec:
  acme:
    # You must replace this email
    address with your own.
    # Let's Encrypt will use this to
    contact you about expiring
    # certificates, and issues related
    to your account.
    email: cto@doxx.fr
    server: https://acme-
v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      # Secret resource that will be
      used to store the account's private
      key.
      name: letsencrypt-prod-account-
key
      # Add a single challenge solver,
      HTTP01 using nginx
      solvers:
        - http01:
          ingress:
```

```
class: nginx
```

## Installer Argocd

- Effectuer l'installation avec la première méthode du getting started : [https://argo-cd.readthedocs.io/en/stable/getting\\_started/](https://argo-cd.readthedocs.io/en/stable/getting_started/)
- Il faut maintenant créer l'ingress (reverse proxy) avec une configuration particulière que nous allons expliquer.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: argocd-server-ingress
  namespace: argocd
  annotations:
    cert-manager.io/cluster-issuer:
      letsencrypt-prod
    kubernetes.io/ingress.class: nginx
    kubernetes.io/tls-acme: "true"
    nginx.ingress.kubernetes.io/ssl-passthrough: "true"
      # If you encounter a redirect loop
      or are getting a 307 response code
      # then you need to force the nginx
```

ingress to connect to the backend using HTTPS.

```
#
```

```
nginx.ingress.kubernetes.io/backend-
protocol: "HTTPS"
spec:
  tls:
    - hosts:
        - argocd.<yoursubdomain>
          secretName: argocd-secret # do not
change, this is provided by Argo CD
      rules:
        - host: argocd.<yoursubdomain>
          http:
            paths:
              - path: /
                pathType: Prefix
            backend:
              service:
                name: argocd-server
            port:
              number: 443
```

- Créez et appliquez cette ressource Ingress.
- Vérifiez dans Lens que l'ingress a bien généré un

certificat (cela peut prendre jusqu'à 2 minutes)

- Chargez la page argocd.<votre sous domaine> dans un navigateur. exp  
argocd.stagiaire1.docker.dopl.uk
- Pour se connecter utilisez le login admin et récupérez le mot de passe admin en allant chercher le secret argocd-initial-admin-secret dans Lens (Config > Secrets avec le namespace argocd activé).

## **Récupérer le corrigé du TP et le pousser sur Gitlab**

- Récupérer le corrigé à compléter du TP CICD gitlab argocd avec git clone -b k8s\_gitlab\_argocd\_correction  
[https://github.com/Uptime-Formation/corrections\\_tp.git](https://github.com/Uptime-Formation/corrections_tp.git)  
k8s\_gitlab\_argocd\_correction
- Ouvrez le projet dans VSCode
- Créer un nouveau projet vide sur gitlab
- Remplacez dans tout le projet, les occurrences de <sousdomain>.dopl.uk par votre sous domaine

par exemple stagiaire1.docker.dopl.uk

- Remplacez également partout gitlab.com/e-lie/cicd\_gitlab\_argocd\_corrections par l'url de votre dépôt Gitlab (sans le https:// ou git@).
- Poussez ce projet dans la branche k8s\_gitlab\_argocd\_correction du dépôt créé précédemment:

```
git remote add gitlab <votre dépôt  
gitlab>  
git push gitlab
```

- Observons le fichier .gitlab-ci.yml.
- Allons voir le pipeline dans l'interface CI/CD de gitlab. Les deux premier stages du pipeline devraient s'être bien déroulés.

## Déploiement de l'application dans argoCD

Expliquons un peu le reste du projet.

- Créez un token de déploiement dans Gitlab > Settings > Repository > Deploy Tokens. Ce token va nous permettre de donner l'autorisation à ArgoCD de lire le dépôt gitlab

(facultatif si le dépôt est public cela ne devrait pas être nécessaire). Complétez ensuite **2 fois** le token dans le fichier k8s/argocd-apps.yaml comme suit :

`https://<nom_token>:`

`<motdepasse_token>@gitlab.com/<votre depo>.git` dans les deux sections `repoURL` : des deux applications.

- Créez les deux applications `monstericon-dev` et `monstericon-prod` dans argocd avec `kubectl apply -f k8s/argocd-apps.yaml`.
- Allons voir dans l'interface d'ArgoCD pour vérifier que les applications se déploient bien sauf le conteneur `monstericon` dont l'image n'a pas encore été buildée avec le bon tag. Pour cela il va falloir que notre pipeline s'exécute complètement.

Les deux étapes de déploiement (dev et prod) du pipeline nécessitent de pousser automatiquement le code du projet à nouveau pour déclencher le redéploiement automatique dans ArgoCD (en mode pull depuis gitlab). Pour cela nous avons besoin de créer également un token utilisateur:

- Allez dans Gitlab > User Settings (en haut à droite dans votre profil) >

Access Tokens et créer un token avec  
read\_repository write\_repository  
read\_registry write\_registry activés.  
Sauvegardez le token dans un fichier.

- Allez dans Gitlab > Settings > CI/CD > Variables pour créer deux variables de pipelines: CI\_USERNAME contenant votre nom d'utilisateur gitlab et CI\_PUSH\_TOKEN contenant le token précédent. Ces variables de pipelines nous permettent de garder le token secret dans gitlab et de l'ajouter automatiquement aux pipeline pour pouvoir autoriser la connexion au dépôt depuis le pipeline (git push).
- Nous allons maintenant tester si le pipeline s'exécute correctement en commitant et poussant à nouveau le code avec git push gitlab.
- Debuggons les pipelines s'ils sont en échec.
- Allons voir dans ArgoCD pour voir si l'application dev a été déployée correctement. Regardez la section events et logs des pods si nécessaire.
- Une fois l'application dev complètement healthy (des coeurs verts partout). On peut visiter l'application en mode dev à l'adresse

[https://monster-dev.<votre\\_sous\\_domaine>](https://monster-dev.<votre_sous_domaine>).

- On peut ensuite déclencher le stage deploy-prod manuellement dans le pipeline, vérifier que l'application est healthy dans ArgoCD (debugger sinon) puis visiter [https://monster.<votre\\_sous\\_domaine>](https://monster.<votre_sous_domaine>).

## Idées d'amélioration

- Déplacer le code de déploiement dans un autre dépôt que le code d'infrastructure. Le pipeline devra cloner le dépôt d'infrastructure, templater avec kustomize la bonne version de l'image dans le bon environnement. Pousser le code d'infrastructure sur le dépôt d'infrastructure. Corriger l'application ArgoCD pour montrer le dépôt d'infrastructure.
- Mutualiser le code de déploiement k8s avec des overlays kustomize
- Utiliser une stratégie de blue/green ou A/B déploiement avec Argo Rollouts ou Istio avec vérification de réussite du déploiement et rollback en cas d'échec.

- Ajouter plus d'étapes réalistes de CI/CD en se basant par exemple sur le livre GitOps suivant.
- Gérer la création des ressources gitlab automatiquement avec Terraform et gérer les secrets (tokens gitlab) consciencieusement.

## Bibliographie

- 2021 - GitOps and Kubernetes Continuous Deployment with Argo CD, Jenkins X, and Flux
- Billy Yuen, Alexander Matyushentsev, Todd Ekenstam, Jesse Suen (z-lib.org)

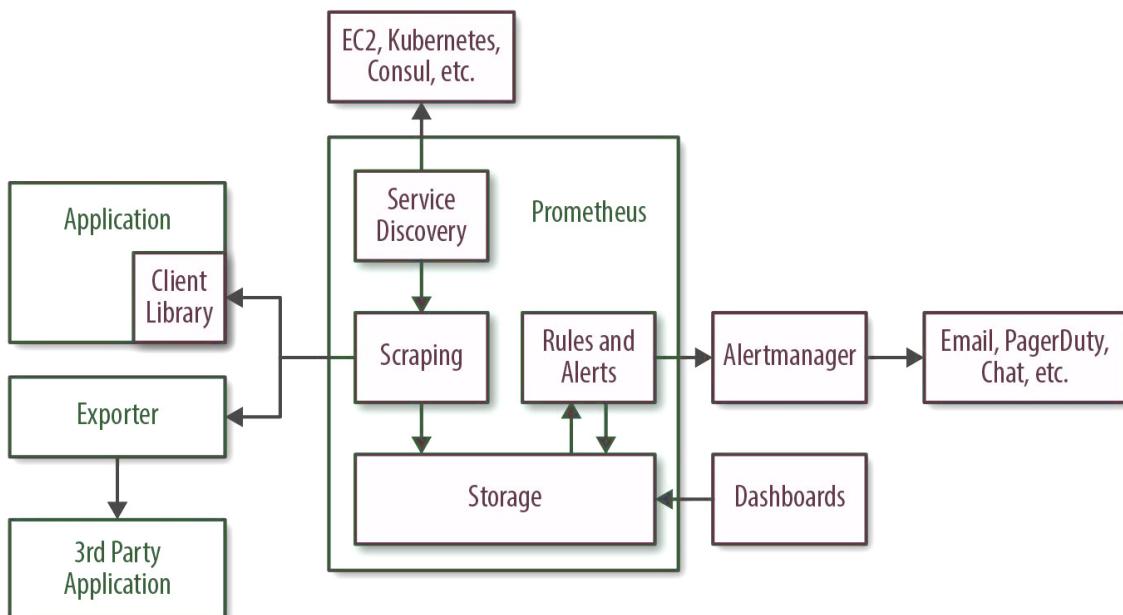
## TP optionnel - Stratégies de déploiement et monitoring

### Installer Prometheus pour monitorer le cluster Minikube

Pour comprendre les stratégies de déploiement et mise à jour d'application dans Kubernetes (deployment and rollout strategies) nous allons installer puis mettre à jour une application d'exemple et observer comment sont gérées les requêtes vers notre application en fonction de la stratégie de déploiement choisie.

Pour cette observation on peut utiliser un outil de monitoring. Nous utiliserons ce TP comme prétexte pour installer une des stack les plus populaires et intégrée avec kubernetes : Prometheus et Grafana. Prometheus est un projet de la Cloud Native Computing Foundation.

Prometheus est un serveur de métriques c'est à dire qu'il enregistre des informations précises (de petite taille) sur différents aspects d'un système informatique et ce de façon périodique en effectuant généralement des requêtes vers les composants du système (metrics scraping).



## Installer Prometheus avec Helm

Installez Helm si ce n'est pas déjà fait. Sur Ubuntu :

```
sudo snap install helm --classic
```

- Créons un namespace pour prometheus et grafana : `kubectl create namespace monitoring`
- Ajoutez le dépôt de chart **Prometheus** et **kube-state-metrics**: `helm repo add prometheus-community https://prometheus-community.github.io/helm-charts` puis `helm repo add kube-state-metrics https://kubernetes.github.io/kube-state-metrics` puis mise à jours des dépôts `helm helm repo update`.
- Installez ensuite le chart prometheus :

```
helm install \
  --namespace=monitoring \
  --version=13.2.1 \
  --set=service.type=NodePort \
  prometheus \
  prometheus-community/prometheus
```

## **kube-state-metrics et le monitoring du cluster**

Le chart officiel installe par défaut en plus de Prometheus, kube-state-metrics qui est une intégration automatique de kubernetes et prometheus.

Une fois le chart installé vous pouvez visualisez les informations dans Lens, dans la première section du menu de gauche Cluster.

## **Déployer notre application d'exemple (goprom) et la connecter à prometheus**

Nous allons installer une petite application d'exemple en go.

- Téléchargez le code de l'application et de son déploiement depuis github: `git clone https://github.com/e-lie/k8s-deployment-strategies`

Nous allons d'abord construire l'image docker de l'application à partir des sources. Cette image doit être stockée dans le registry de minikube pour pouvoir être ensuite déployée dans le cluster. En mode développement Minikube s'interface de façon très fluide avec la ligne de commande Docker grâce à quelques variables d'environnement :

`minikube docker-env`

- Changez le contexte de docker cli pour pointer vers minikube avec `eval` et la commande précédente.
- Allez dans le dossier `goprom_app` et “construisez”

l'image docker de l'application avec le tag  
`uptimeformation/goprom`.

- Allez dans le dossier de la première stratégie `recreate` et ouvrez le fichier `app-v1.yml`. Notez que `image:` est à `uptimeformation/goprom` et qu'un paramètre `imagePullPolicy` est défini à `Never`. Ainsi l'image sera récupéré dans le registry local du docker de minikube ou sont stockées les images buildées localement plutôt que récupéré depuis un registry distant.
- Appliquez ce déploiement kubernetes:

### **Observons notre application et son déploiement kubernetes**

- Explorez le fichier de code go de l'application `main.go` ainsi que le fichier de déploiement `app-v1.yml`. Quelles sont les routes http exposées par l'application ?
- Faites un forwarding de port Minikube pour accéder au service `goprom` dans votre navigateur.
- Faites un forwarding de port pour accéder au service `goprom-metrics` dans votre navigateur

(c'est sur la route /metrics). Quelles informations récupère-t-on sur cette route ?

- Pour tester le service `prometheus-server` nous avons besoin de le mettre en mode NodePort (et non ClusterIP par défaut). Modifiez le service dans Lens pour changer son type.
- Exposez le service avec Minikube (n'oubliez pas de préciser le namespace monitoring).
- Vérifiez que Prometheus récupère bien les métriques de l'application avec la requête PromQL :  
`sum(rate(http_requests_total{app="goprom"}) [5m]) by (version)`.
- Quelle est la section des fichiers de déploiement qui indique à Prometheus où récupérer les métriques ?

## **Installer et configurer Grafana pour visualiser les requêtes**

Grafana est une interface de dashboard de monitoring facilement intégrable avec Prometheus. Elle va nous permettre d'afficher un histogramme en temps réel du nombre de requêtes vers l'application.

Créez un secret Kubernetes pour stocker le loging admin de grafana.

```
cat <<EOF | kubectl apply -n monitoring -f -
apiVersion: v1
kind: Secret
metadata:
  namespace: monitoring
  name: grafana-auth
type: Opaque
data:
  admin-user: $(echo -n "admin" |
base64 -w0)
  admin-password: $(echo -n "admin" |
base64 -w0)
EOF
```

Ensuite, installez le chart Grafana en précisant quelques paramètres:

```
helm repo add grafana
https://grafana.github.io/helm-charts
helm repo update
helm install \
--namespace=monitoring \
--version=6.1.17 \
```

```
--set=admin.existingSecret=grafana-auth \
--set=service.type=NodePort \
--set=service.nodePort=32001 \
grafana \
grafana/grafana
```

Maintenant Grafana est installé vous pouvez y accéder en forwardant le port du service grâce à Minikube:

```
$ minikube service grafana
```

Pour vous connecter utilisez, username: admin, password: admin.

Il faut ensuite connecter Grafana à Prometheus, pour ce faire ajoutez une DataSource:

```
Name: prometheus
Type: Prometheus
Url: http://prometheus-server
Access: Server
```

Créer une dashboard avec un Graphe. Utilisez la requête Prometheus (champ query suivante):

```
sum(rate(http_requests_total{app="gopr [5m]})) by (version)
```

Pour avoir un meilleur aperçu de la version de

l'application accédée au fur et à mesure du déploiement, ajoutez `{ {version} }` dans le champ legend.

## Observer un basculement de version

Ce TP est basé sur l'article suivant:

<https://blog.container-solutions.com/kubernetes-deployment-strategies>

Maintenant que l'environnement a été configuré :

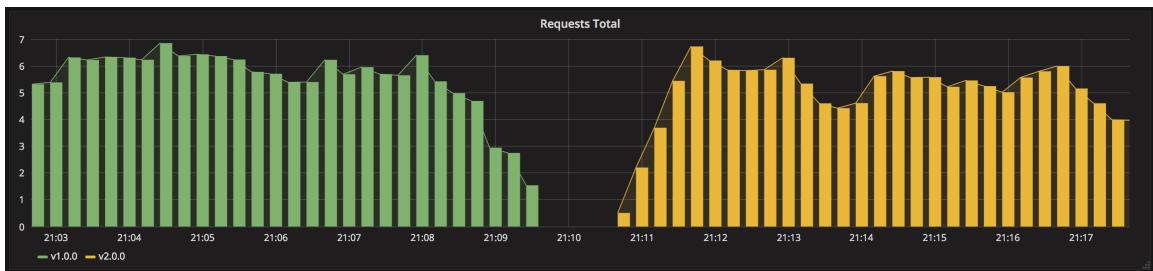
- Lisez l'article.
- Vous pouvez testez les différentes stratégies de déploiement en lisant leur README . md.
- En résumé, pour les plus simple, on peut:
- appliquer le fichier app-v1 . yml pour une stratégie.
- lancer la commande suivante pour effectuer des requêtes régulières sur l'application:

```
service=$(minikube service goprom --url) ; while sleep 0.1; do curl "$service"; done
```
- Dans un second terminal (pendant que les requêtes tournent) appliquer le fichier app-v2 . yml

correspondant.

- Observez la réponse aux requêtes dans le terminal ou avec un graphique adapté dans graphana (Il faut configurer correctement le graphique pour observer de façon lisible la transition entre v1 et v2). Un aperçu en image des histogrammes du nombre de requêtes en fonction des versions 1 et 2 est disponible dans chaque dossier de stratégie.
- supprimez le déploiement+service avec `delete -f` ou dans Lens.

Par exemple pour la stratégie **recreate** le graphique donne:



## Facultatif : Installer Istio pour des scénarios plus avancés

Pour des scénarios plus avancés de déploiement, on a besoin d'utiliser soit un *service mesh* comme Istio (soit un plugin de rollout comme Argo Rollouts mais pas ce que nous proposons ici).

1. Sur k3s, supprimer la release Helm du Ingress

Controller Traefik (ou le ingress Nginx) pour le remplacer par l'ingress Istio.

2. Installer Istio, créer du trafic vers l'ingress de l'exemple et afficher le graphe de résultat dans le dashboard Istio : <https://istio.io/latest/docs/setup/getting-started/>

3. Utiliser ces deux ressources pour appliquer une stratégie de déploiement de type A/B testing poussée :

- <https://istio.io/latest/docs/tasks/traffic-management/request-routing/>
- <https://github.com/ContainerSolutions/k8s-deployment-strategies/tree/master/ab-testing>

## Contenu intégral

### Exporter les supports en pdf

Pour exporter correctement les TPs et autres pages de ce site au format pdf, utilisez la fonction **imprimer** de Google Chrome ou Firefox (vous pouvez aussi activer le Mode Lecture de Firefox en cliquant Affichage > Passer en Mode Lecture) **en ouvrant la page suivante** : [Contenu intégral](#).

