Introduction à Docker

1 - Virtualisation

La virtualisation :

- Un concept très ancien (années 60, mainframes IBM)
- Démocratisation récente (années 2000, PC et serveurs)

Le principe :

- Émuler un ordinateur physique dans un autre
- Emulation = logiciel “Hyperviseur”

Terminologie :

- Hôte (Host) = la machine physique qui héberge les Invités
- Invité (Guest) = la machine logique (VM) hébergée sur l'Hôte
- Emulation = reproduction exacte d'un système dans un autre système (un ordinateur dans un ordinateur)

Simulation = reproduction d'un système qui a l'apparence (extérieure) d'un autre système mais est totalement incompatible avec lui

Avant la virtualisation :

- 1 service = souvent 1 serveur physique
- Installation d'un OS + installation du/des services
- Maintenance matérielle
- Maintenance logicielle (de l'OS)
- Changement de matériel =
  - un long temps de réinstallation et de préparation
  - un temps d'indispo lors de la migration
- "Scalabilité" compliquée

Avec la virtualisation :

- 1 gros serveur physique
- Plusieurs VMs
- 1 service = 1 serveur logique dans une VM
- Installation d'un OS + installation du/des services
- Pas de maintenance matérielle pour la VM
- Maintenance logicielle (de l'OS)
- Changement de matériel / Scalabilité =
  - un temps de migration très court
  - un temps d'indispo très limité

Intérêts :

- Plusieurs VMs (machines virtuelles) sur un seul ordi
- Cloisonnement des VMs
- Administration et maintenances simplifiées (pb matériel -> déplacement VM, qq minutes)
- Coûts mieux maîtrisés :
  - Seules les VMs utiles sont lancées (une VM arrêtée ne coûte rien)
  - Ressources (humaines, financières) concentrées sur moins de matériels et d'infrastructure.

Intérêts :

- Assurance d'un matériel standard du point de vue de la VM : le matériel est émulé, la VM ne voit pas le matériel physique. Migration simplifiée.
- Meilleures performances : affectation des ressources (CPU, RAM) en fonction des besoins, parfois ponctuels (pics de montée en charge)

Inconvénients :

- Une perte de performance avec la virtualisation du matériel :
  - CPU
  - Accès disque,
  - Réseau
  - Hyperviseur

  Note : impact très modéré
- Une perte de performance avec la multiplication des OS (hôte + invités)

2 - Conteneurisation

La conteneurisation :

- Des balbutiements (chroot fin 70)
- Un aboutissement assez récent (2008 pour Linux avec LXC)
- Décollage (2013 pour Docker)

Le principe :

- Une “sorte” de virtualisation mais en beaucoup plus léger.

Comparable à la virtualisation :

- Une machine hôte (physique) qui accueille des conteneurs (sortes de “machines logiques”)
- Cloisonnement : environnement d’exécution séparé de l’hôte et des autres conteneurs

Des différences fondamentales :

- Pas d'émulation matérielle
- Pas d'OS (kernel) sur les invités
- Un cloisonnement par Name Spaces (Process, Mount, IPC, User et Network)
- Conteneurs ne voient pas les processus hôte
- Hôte voit les processus des conteneurs !

Un conteneur = un processus du point de vue de l'hôte.

Avantages :

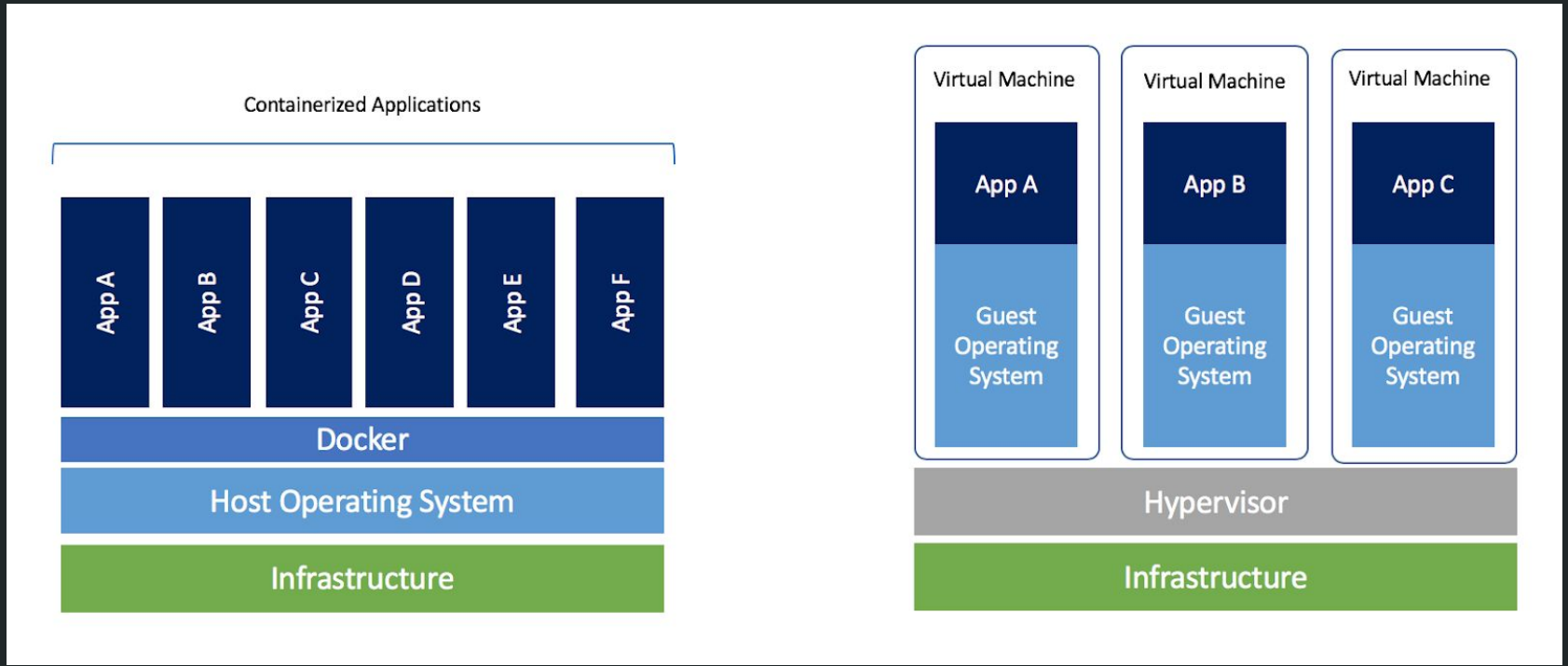
- Une archi extrêmement légère
- Pas de surcharge CPU (car conteneur ~= le processus qui tourne dans le conteneur)
- Une consommation de ressources maîtrisée (pas d'allocation réservée en CPU, RAM, Disque)
- Un démarrage très rapide (quelques secondes)
- Pas de pollution de l'hôte (installation de packages)
- Elimination des incompatibilités (plusieurs versions d'un même package)

Inconvénients :

- Le même OS (kernel) pour tous les conteneurs = pas possible d'avoir des versions de noyau différentes
- Une gestion des droits plus complexe
- Nécessité d'avoir des droits admin pour certaines actions ou certaines configurations
- Pour Docker : uniquement Linux, mais des solutions existent pour Windows et macOS

3 - Docker

Un dessin vaut 1000 mots...



En standard, un conteneur c'est :

- Un processus (celui qui tourne dans le conteneur)
- Un noyau partagé avec l'hôte (normal puisque c'est un processus)
- Une arborescence privée
- Un réseau privé local
- Une vision privée des processus (pas de vue sur les processus du système) + remappage des PIDs
- Un remappage des Users (root dans conteneur != root hôte)

En option, un conteneur peut :

- Partager un bout de l'arbo du FS avec son hôte
- Avoir un VPN partagé avec d'autres conteneurs ciblés
- Partager la même pile TCP/IP que l'hôte
- Avoir root dans conteneur = root hôte

La sécurité, le cloisonnement et le partage de certains Name Spaces est géré par le système hôte.

L'environnement est aussi sécurisé (et donc aussi faible) que l'est l'hôte. Penser aux mises à jour des patches de sécurité (OS et Docker).

Il existe d'autres techno de conteneurisation (sous Unix et Windows)

A partir de maintenant, on ne s'intéressera qu'à Docker.

4 - Terminologie

Image

Un modèle de départ pour les conteneurs.
Construction par couches (Layers)
Layers ~= Calques (papier ou à la “Photoshop”)
Un calque modifie la pile de calques du dessous
Tous les calques sont en lecture seule

Une image :

- Se base sur une autre image (ou image “scratch”)
- Apporte ses adaptations (ajout de layers)
- Peut servir de base à d’autres images
- Layers en lecture seule = immuables -> pas de risque de casser des images qui l’utilisent

Images publiques : Docker Hub (hub.docker.com)
Images privées : Hub privé ou machine perso/Serveur privé

Couche (Layer)

- Une étape dans la construction d'une image = 1 ligne du Dockerfile (vu plus tard)
- Généralement un layer = modification (ajout, changement, suppression) dans le File System
- C'est aussi une sorte d'image "intermédiaire"

Conteneur (container)

Une instanciation d'une image.
Utilise les layers de l'image (RO = Lecture seule)
Ajoute un layer supplémentaire (RW = Lecture/Écriture)
= layer de travail du conteneur (contient les créa, modifs des objets des layers RO du FS -> Copy On Write)
-> Layers des images (RO) sont partagés entre conteneurs.

Un conteneur est volatile :

- Par défaut les données sont dans le conteneur
- Suppression conteneur = données perdues

Pour persister les données : les volumes (voir plus loin)

Hub (Dépôt)

- Images officielles : produites par Docker Inc.
- Images vérifiées : produites par les éditeurs (“Verified publishers”).
- Images publiques : tout le monde peut publier des images. Attention à la sécurité
- Dépôt officiel : hub.docker.com
- Dépôts tiers : publics ou privés (“Registries”)
- IUT : docker.iutlan.etu.univ-rennes1.fr

Par image :

- Historique de versions = tags
- Tag “latest” = Dernière version
- Conservation des versions
- Possibilité de cibler une version spécifique = garantie de stabilité pour l’utilisateur
- Accès aux sources de construction des images (voir Dockerfile)
- Souvent lié à un dépôt Gitlab/Github

Compte gratuit :

- 200 “images pulls” / jour
- Dépôt public illimité

Comptes payants :

- Limites beaucoup plus larges, voire illimitées
- Pour les entreprises avec gros besoins

Volumes

Données d'un conteneur = stockage dans conteneur
Conteneur supprimé = perte du stockage
Persistance = volumes
Hôte et Conteneur lisent et écrivent les mêmes fichiers

2 types :

- Volumes mappés
- Volumes managés

Volumes mappés (bind mounts) :

- Pratique pour partager des données présentes sur le FS de l'hôte (ex : un dossier html)
- Forte dépendance du File System hôte (droits etc)
- Performance moins bonnes

Volumes managés (managed volumes) :

- Possible à migrer entre plateformes (compatibilité)
- Meilleures performances

Networks

Par défaut un conteneur :

- voit son hôte , voit les autres conteneurs configurés aussi par défaut, accède au monde extérieur (Internet)
- n'accepte pas de connexion de l'extérieur (ports ouverts uniquement dans le conteneur)

Conteneur peut être attaché à un ou plusieurs réseaux :

- Ca change le mode par défaut (plus d'accès à l'hôte, plus d'accès aux conteneurs configurés par défaut)
- Échanges entre conteneurs attachés au(x) mêmes réseau(x)
- Attachement possible à un réseau à chaud (en live)

5 - Commande docker

Manipulation de l'environnement Docker.

Types :

- image
- container
- network
- volume
- divers autres

Syntaxe : docker <type> <commande>

Aide : docker --help

Aide sur un type donné :

docker <type> --help

Exemples :

- docker image --help
- docker container --help

Aide sur une commande donnée :

docker <type> <commande> --help

Exemples :

- docker image pull --help
- docker container run --help

run

Instancie un conteneur à partir d'une image et démarre son exécution (i.e. lance le processus du conteneur)

Syntaxe : docker container run <image>[:tag]

Exemple : docker container run hello-world

Syntaxe : docker container run <image>[:tag]

Si tag pas spécifié => latest

Utiliser latest (par défaut) = pas toujours une bonne idée. Privilégier une version (un tag) connue pour fonctionner comme attendu.

Cycle de vie

Types de conteneurs :

- Sans fin : ne s'arrêtent pas seuls (ex : serveur Web)
- Avec fin : traitement fini -> fin du processus (ex : un convertisseur d'images)

Quand le processus dans un conteneur s'arrête, le conteneur s'arrête aussi. Rappel : un conteneur = un processus (celui qui tourne "dans le conteneur")

Conteneur démarré = ajouté dans une liste

Conteneur (processus) terminé = gardé dans la liste

Liste = possibilité de faire “revivre” un conteneur

Chaque docker container run => nouveau conteneur => ajouté dans la liste => Liste grandit !

ps

Listing des conteneurs actifs, arrêtés et terminés.

Syntaxe : docker container ps [-a]

Sans -a : seulement les conteneurs actifs (processus en cours d'exécution). Avec -a : tous (a = all)

Syntaxe : docker container ps [-a]

Sans -a : seulement les conteneurs actifs (processus en cours d'exécution). Avec -a : tous (a = all)

NB : ps et ls sont des alias
docker container ps == docker container ls

ID

Tous les objets (conteneurs, images, networks, volumes etc.) dans Docker ont un ID unique.

Exemple : docker container ps

Les commandes créant des objets affichent ID créé : docker container run hello-world -> Affiche ID du conteneur créé

ID = très longue valeur hexadécimale (64 cars)

OK pour utiliser un début d'ID tant que pas d'ambiguïté

docker container ps = généralement les 12 premiers cars. Possibilité d'afficher plus si besoin. Proba conflit sur 12 cars ~= nulle.

start/stop

Sur conteneur actif :

- Arrêt du conteneur (SIGTERM) :
  docker container stop <ID>

Sur conteneur arrêté :

- Redémarrage du conteneur :
  docker container start <ID>

kill

Sur conteneur actif, arrêt du conteneur (SIGKILL) :

docker container kill <ID>

kill vs stop = SIGKILL vs SIGTERM => KILL garanti, TERM ça dépend…

start/kill : données conservées dans le conteneur

rm

Sur conteneur arrêté, supprime le conteneur :

docker container rm <ID>

Supprime aussi les données du conteneur mais pas les volumes.

Detached

Un conteneur sans fin (daemon, style serveur Web) = par défaut est attaché au Terminal => Problème.

Pour détacher (laisser tourner en background) :

docker container run -d <image>[:tag]

Exemple : docker container run -d nginx

exec

Sur conteneur actif, exécute une commande DANS le conteneur :

docker container exec <ID> <commande>

Exemple :
docker container run -d nginx
docker container exec <ID> ls /

exec (interactif)

Pour commandes interactive (qui lisent au clavier),
besoin d'attacher un TTY (TeleTYpe = Terminal)

docker container exec -t -i <ID> <commande>

Exemple :
docker container run -d nginx
docker container exec -ti <ID> bash

Ports (mappage)

Mappage port hôte <-> port conteneur

Syntaxe :
docker container run -p port_hote:port_conteneur <image>

Exemple : docker container run -p 9999:80 nginx
Navigateur : http://localhost:9999

Bind mount (volume mappé)

Mappage dossier hôte <-> dossier conteneur
Syntaxe :
docker container run -v
dossier_hote:dossier_conteneur <image>

Exemple :
docker container run -v $(pwd):/usr/share/nginx/html -p 8888:80
-d nginx
Navigateur : http://localhost:8888

https://commons.wikimedia.org/wiki/File:Thats_all_folks.svg