R4.DEVCLOUD.09
FONDAMENTAUX DE LA CONTENEURISATION

BUT 2 RT

Description

La conteneurisation des services et des applications est l'un des outils majeurs du Cloud. Elle offre une grande flexibilité dans la conception et la maintenance des infrastructures. Elle facilite également la portabilité et la gestion du cycle de vie des services et des applications

- Découvrir le principe de conteneur et ses usages
- Infrastructures basées sur des conteneurs
- Plateforme de partage et de stockage
- Création de conteneur et configuration

Compétences et Apprentissages critiques

- Compétences:
  - Devcloud 1 : coordonner des infrastructures modulaires
  - Devcloud 2 : accompagner le développement d’application
- Apprentissages critiques :
  - AC24.01 DevCloud : proposer une solution cloud adapté à l’entreprise
  - AC24.02 DevCloud : virtualiser son environnement
  - AC24.03 DevCloud : utiliser les services du cloud
  - AC25.03 DevCloud : programmer son réseau par le code

Conteneur ?

- Réflexion issue du monde du transport.
  - Avant les conteneurs multimodaux, les opérations de chargement/ déchargement étaient longues, compliquées et généraient des pertes (casses, vols, oublis)
  - Avec les conteneurs, le chargement est pris en charge par la société, ensuite le conteneur est acheminé jusqu’a sa destination

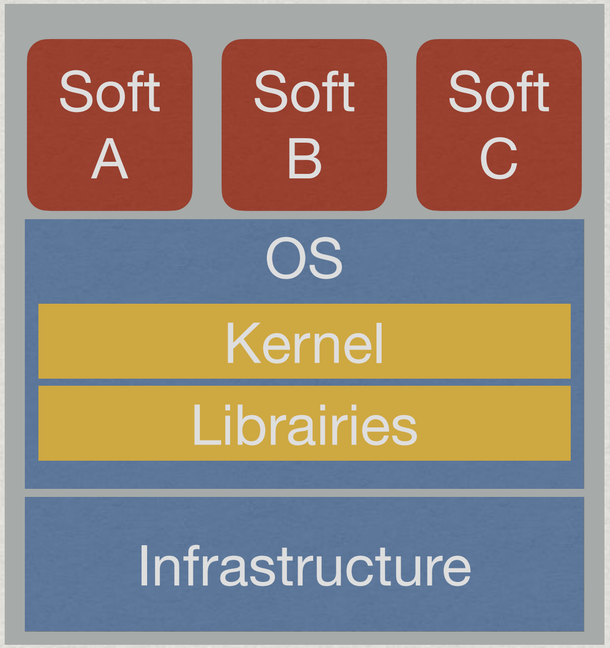
Gain de sécurité,
les responsabilités sont bien définies

Conteneur dans le monde du logiciel

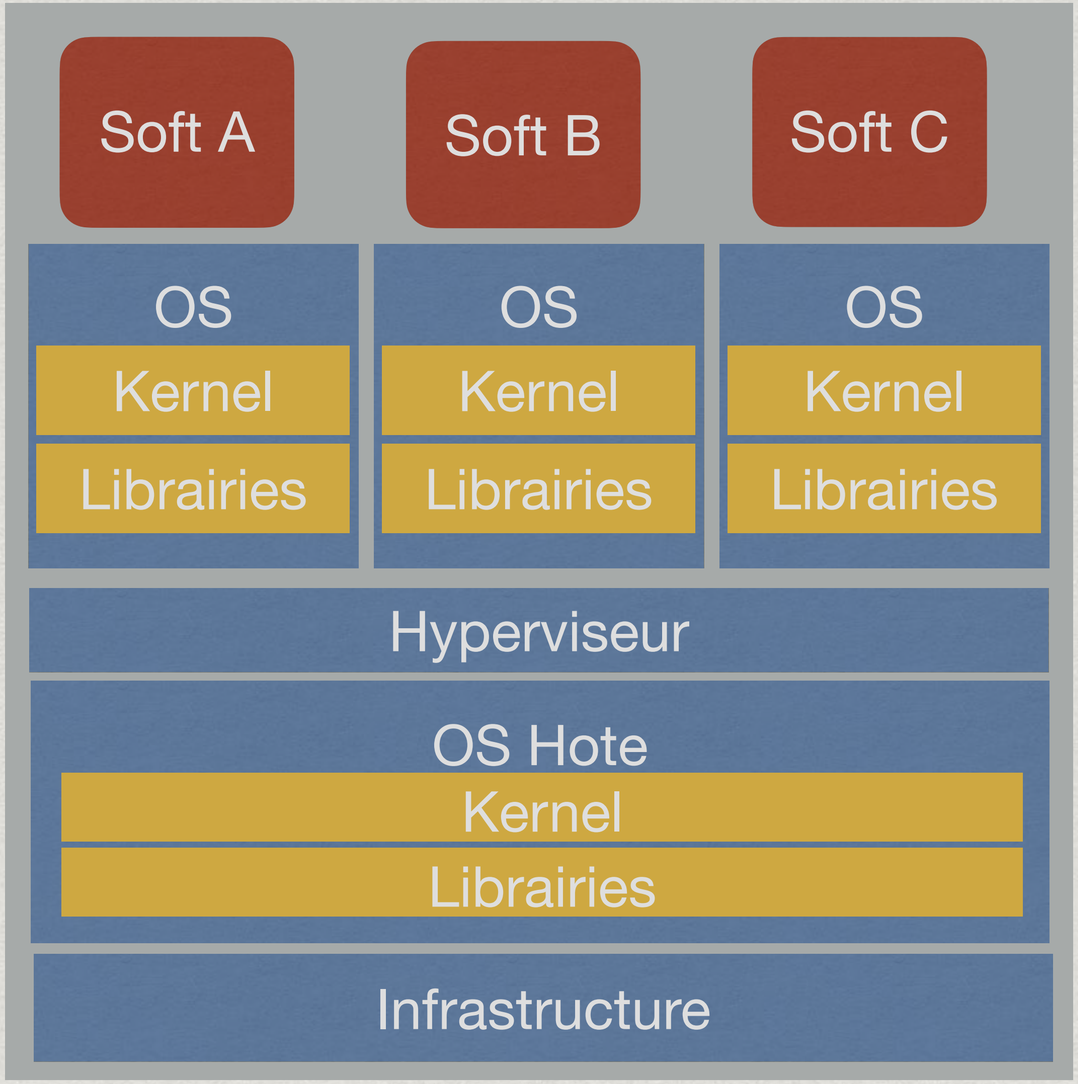
- La responsabilité se répartie sur 2 acteurs:
  - Le développeur
  - L’exploitant (responsable de l’infrastructure)
- Mais sans conteneur, certains pré-requis sont flous (OS, dépendance, ressources)
- Les conteneurs permettent de répartir les responsabilités
  - Le développeur est responsable de son application, des dépendances jusqu’à l’OS
  - L’exploitant est responsable du réseaux , des ressources, du stockage

virtualisation

Installation classique

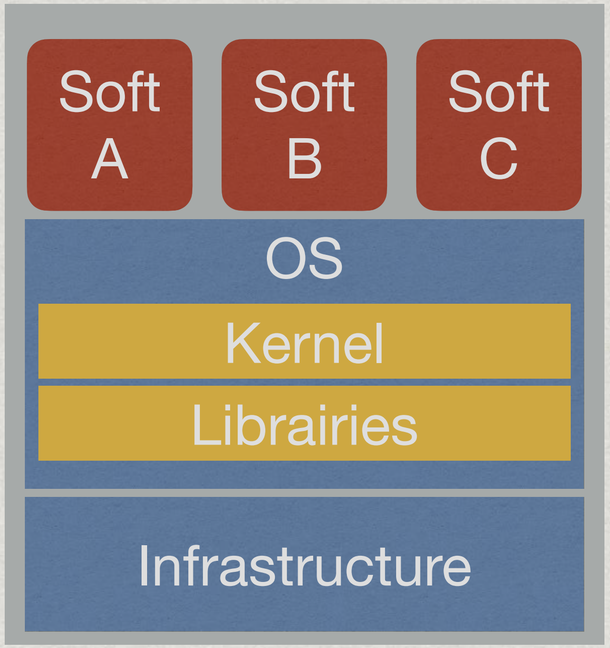


Installation virtualisée (une VM par soft)

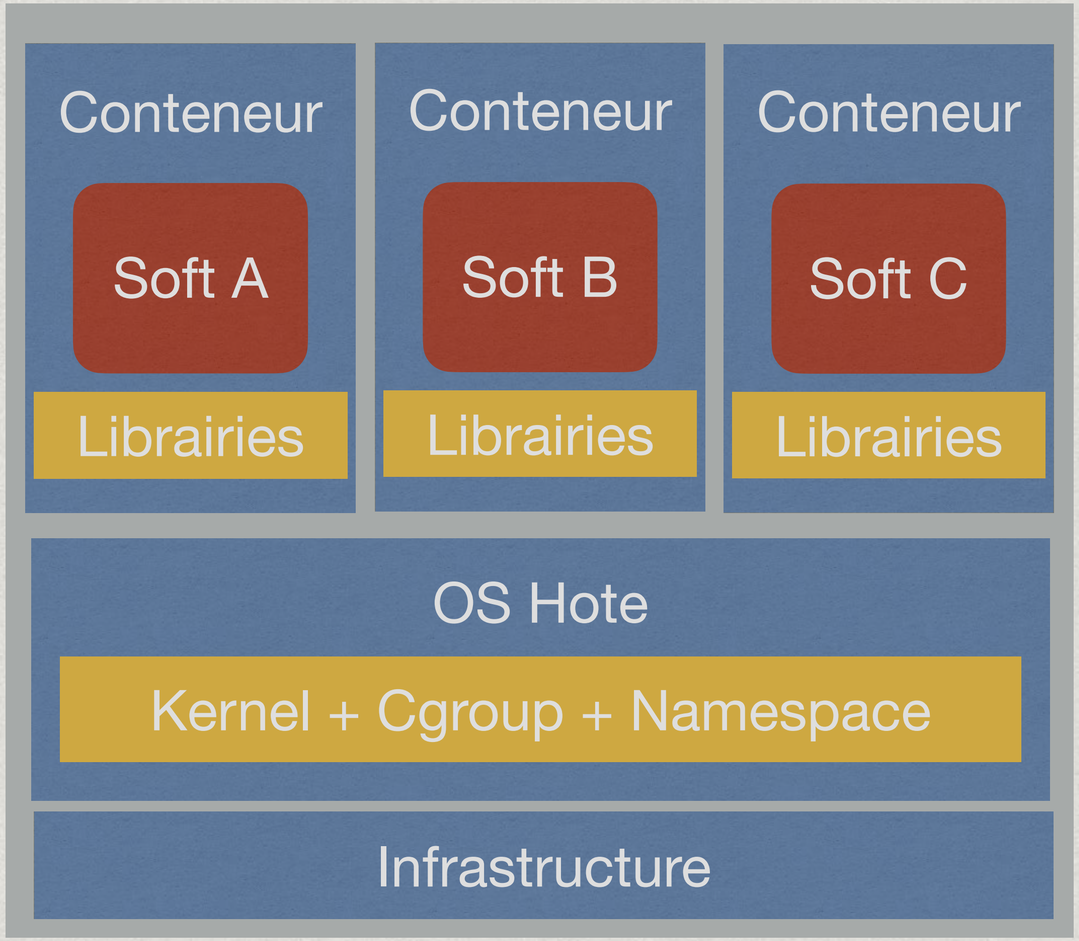


Conteneur

Installation classique



Installation à base de conteneur



Historique de la conteneurisation

- 1979 : chroot => isolation du système de fichier en redirigeant la racine du système vers un autre emplacement. Fonctionnalité Unix et BSD
- 2000 : BSD Jail : permet d'isoler le système de fichiers mais aussi les utilisateurs, les processus et le réseaux
- 2002 : Linux Namespaces: isolations, utilisateurs, processus, réseaux, … sous linux
- 2005 openVZ : solution opensource de conteneurs, mais pas d'intégration dans le noyaux linux
- 2006 : process containers lancé par google pour limité les usages des ressources matérielles => remplacé par les *control Groups* (CGROUP) inclus dans le noyaux linux en 2008
- 2008 : LinuX Containers (LXC) combine namespace et cgroup en utilisant des fonctionnalités du noyaux
- 2013 : Docker : fourni un CLI, le dockerfile pour concevoir de nouveau conteneur, et le docker registre pour stocker et diffuser des images

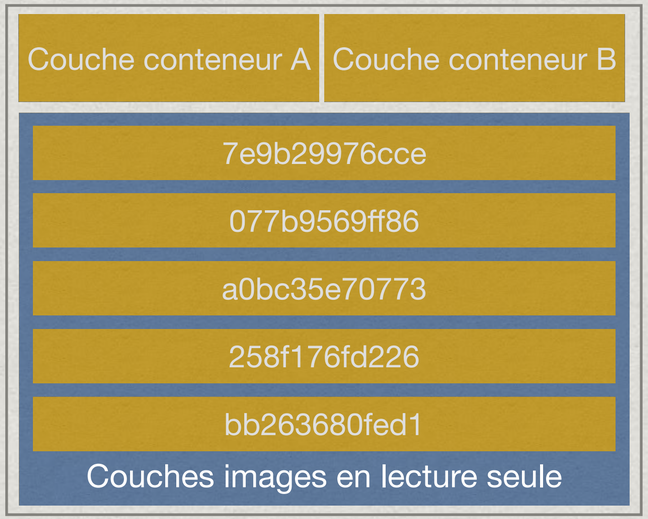
Docker

- Docker s’exécute nativement sous Linux
- Depuis 2016, exécution possible sous windows et Mac dans un environnement virtuel (VM hyperV sous windows et sous Mac démarrant un linux light)
- Une image Docker est un empilement de couches réutilisables car disponibles en lecture seule => Union File System (UFS)
- Seule la dernière couche de l’image est en écriture.

UFS

- Une image docker est constituées d'une succession d'images en lecture seule apportant les différentes couche nécessaire à notre application

```
root@hikaru16:/var/lib/docker/containers# docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx
bb263680fed1: Pull complete
258f176fd226: Pull complete
a0bc35e70773: Pull complete
077b9569ff86: Pull complete
3082a16f3b61: Pull complete
7e9b29976cce: Pull complete
Digest: sha256:6650513efd1d27c1f8a5351cbd33edf85cc7e0d9d0fcb4ffb23d8fa89b601ba8
Status: Downloaded newer image for nginx:latest
docker.io/library/nginx:latest
root@hikaru16:/var/lib/docker/containers#
```



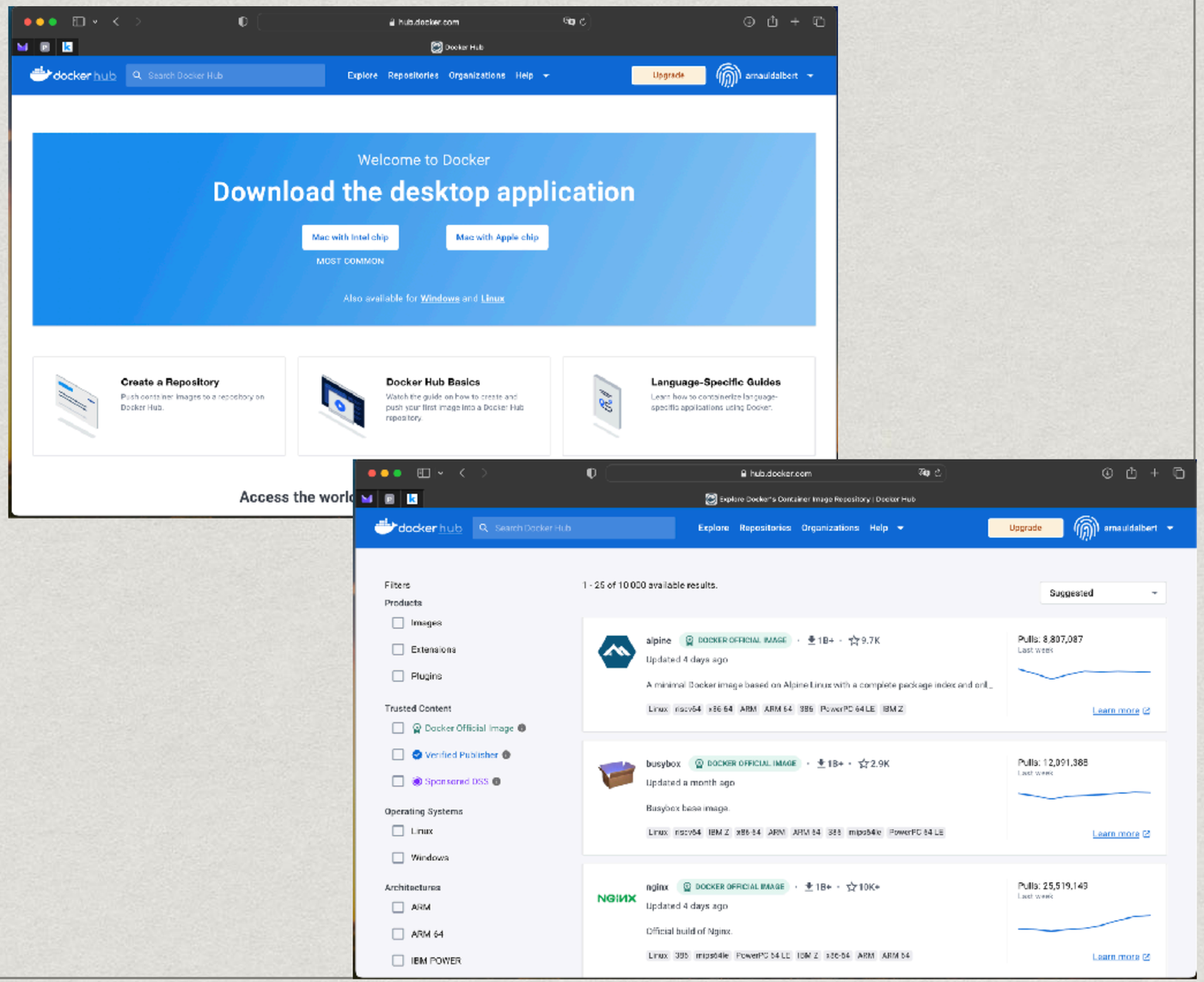
- L'intérêt de ces multiples couches en lecture seule, réside en leur réutilisation possible.

Copy on write, Persistence des données

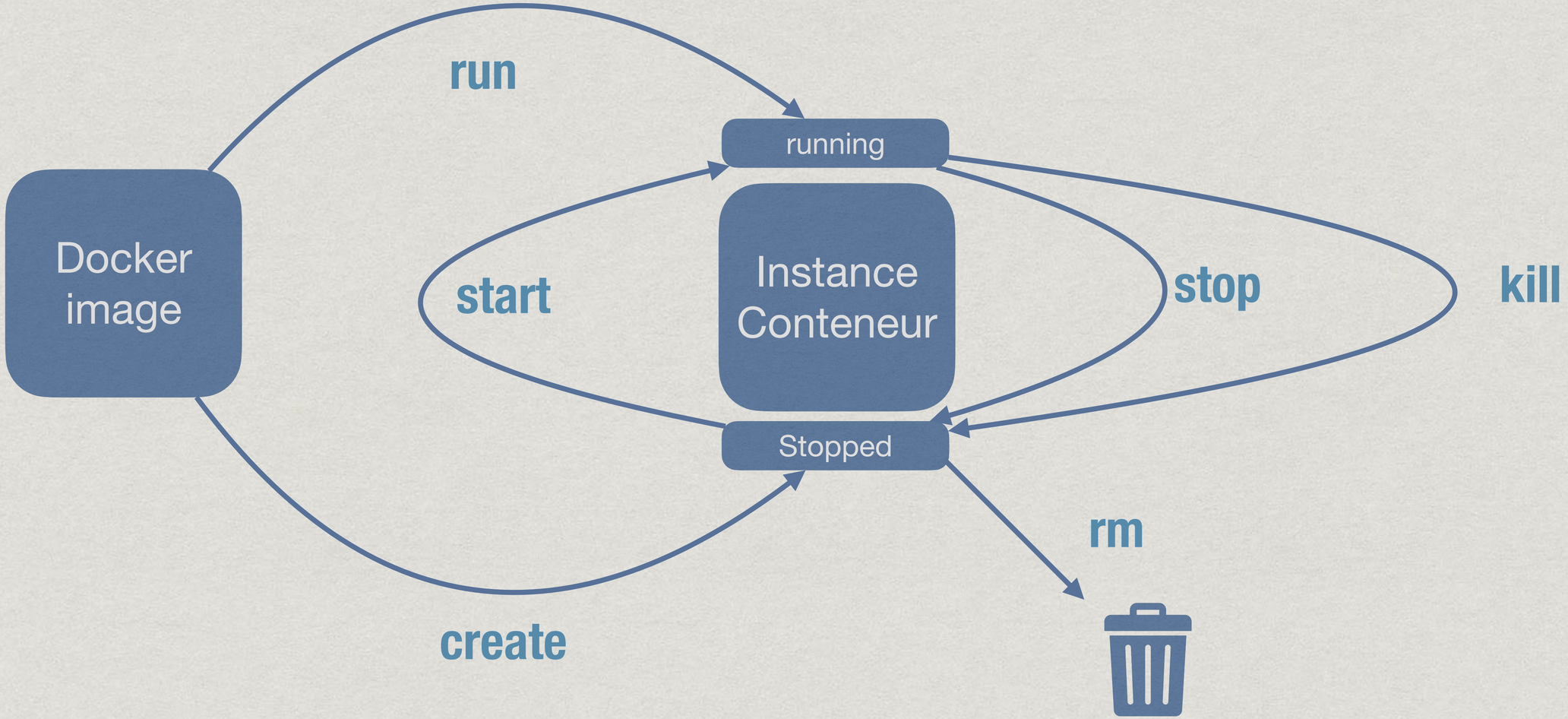
- Quand 2 conteneurs sont crées à partir de la même image et que des modifications sont opérés sur les fichiers, ces modifications sont écrites dans la couche du conteneur.
- Toutes ces modifications sont perdues à l'arrêt du conteneur
- Pour persister des données, il faut associer un volume au conteneur :
  - Point de montage du système de fichier local dans le conteneur (utilisé pour les bases de données, les services web ,…)

Docker Hub et registry

- Des images docker sont stockées dans des registry publics ou privés
- Par défaut la registry docker hub.docker.com est utilisée pour charger des images (nécessite la création d'un compte)
- Une registry locale peut être utilisée pour stocker les images crées



Cycle de vie d’un conteneur



Commandes docker

- *docker pull* **#nom-image** : télécharge l'image depuis le registry

```
nono@hikaru16:~$ docker pull alpine
Using default tag: latest
latest: Pulling from library/alpine
63b65145d645: Pull complete
Digest: sha256:69665d02cb32192e52e07644d76bc6f25abeb5410edc1c7a81a10ba3f0efb90a
Status: Downloaded newer image for alpine:latest
docker.io/library/alpine:latest
```

- *docker images* ou *docker image ls* : liste toutes les images disponibles localement

```
nono@hikaru16:~$ docker image ls
REPOSITORY   TAG       IMAGE ID       CREATED       SIZE
alpine       latest    b2aa39c304c2   8 days ago    7.05MB
```

- *docker image rm* **#nom-image** ou *docker image rm* **#image-id**

```
nono@hikaru16:~$ docker image rm alpine
Untagged: alpine:latest
Untagged: alpine@sha256:69665d02cb32192e52e07644d76bc6f25abeb5410edc1c7a81a10ba3f0efb90a
Deleted: sha256:b2aa39c304c27b96c1fef0c06bee651ac9241d49c4fe34381cab8453f9a89c7d
Deleted: sha256:7cd52847ad775a5ddc4b58326cf884beee34544296402c6292ed76474c686d39
```

  - On ne peut effacer une image que si elle n'est pas utilisé par un conteneur
  - Les commandes associées aux images commencent toujours pas **image**

Docker : commandes

- *docker login* (IP:port) permet de se connecter au registry (par défaut le docker hub mais nous pouvons spécifier le registry IP:port). Les options de cette commande sont -p pour le Password et -u pour le username. Sinon ils sont demandés en interactif

```
nono@hikaru16:~$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, he
ad over to https://hub.docker.com to create one.
Username: arnauldalbert
Password:
WARNING! Your password will be stored unencrypted in /home/nono/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

- *docker run* **#nom-image :** si l'image n'est pas dans le registry local, elle est téléchargée depuis le hub docker

```
nono@hikaru16:~$ docker run alpine
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
63b65145d645: Pull complete
Digest: sha256:69665d02cb32192e52e07644d76bc6f25abeb5410edc1c7a81a10ba3f0efb90a
Status: Downloaded newer image for alpine:latest
```

Docker ps

- La commande docker run lance un conteneur, qui bloque la console jusqu'à la fin d'exécution du processus du conteneur
- *docker ps* ou *docker container ls* permet de lister les conteneurs actifs
- L'option -a permet d'afficher tous les conteneurs

```
nono@hikaru16:~$ docker run alpine
nono@hikaru16:~$ docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
nono@hikaru16:~$ docker ps -a
CONTAINER ID   IMAGE     COMMAND     CREATED          STATUS                      PORTS     NAMES
bf0266045b6a   alpine    "/bin/sh"   8 seconds ago   Exited (0) 8 seconds ago             pedantic_wu
nono@hikaru16:~$
```

- Docker affecte nom automatiquement au conteneur, ainsi qu'un ID

Docker run

- Un docker non interactif exécute le processus qu'il embarque et s'arrête directement à la fin de ce processus.
- Pour lancer un conteneur sans bloquer le terminal et pouvoir exécuter des commandes au sein du conteneur : *docker run -t -i -d* **#nom-image**
  - L'option -t associe un tty au conteneur
  - L'option -d (- - detach) permet de libérer le terminal
  - L'option -i permet l'exécution interactive de commande
  - L'option - -rm permet de supprimer le conteneur à l'issue de son exécution

Gestion des conteneurs

- Gestion des états:
  - *docker stop* **#container-id**
  - *docker start* **#container-id**
  - *docker kill* **#container-id**
  - *docker restart* **#container-id**

Docker execution interactive

- *docker exec -t -i* **#conteneur-id *commande*** : exécute une commande dans un conteneur qui fonctionne en mode interactif

```
nono@hikaru16:~$ docker run -tid --rm alpine
a66c8a772689a92e22b8960373cfbccc05897708244008e124a52f6312f818bc
nono@hikaru16:~$ docker ps
CONTAINER ID   IMAGE     COMMAND     CREATED          STATUS          PORTS     NAMES
a66c8a772689   alpine    "/bin/sh"   6 seconds ago    Up 4 seconds              keen_napier
nono@hikaru16:~$ docker exec -ti a66c8a772689 /bin/sh
/ # uname -a
Linux a66c8a772689 5.10.0-21-amd64 #1 SMP Debian 5.10.162-1 (2023-01-21) x86_64 Linux
/ # ps ux
PID   USER     TIME  COMMAND
    1 root      0:00 /bin/sh
    7 root      0:00 /bin/sh
   14 root      0:00 ps ux
/ #
```

Docker commandes de copie

- Copie d’un fichier du conteneur vers l’host
  - *docker cp* **#container-id**:src_path dest_path
- Copie un fichier de l’host vers le conteneur
  - d*ocker cp* src_path **#container-id**:dest_path

Persistence des données et docker

- Afin d’assurer la persistence des données, il faut pouvoir écrire ces données dans un endroit qui n’est pas détruit à chaque arrêt
- La couche d’écriture d’un container lui est propre et disparait à la fin de l’exécution du container.
- Pour pallier cela on peut monter un volume, c’est à dire créer un lien entre un répertoire du système et un point de montage dans le container.

Persistence des données : volumes

- Pour ajouter un volume à un conteneur lors de la création, on ajoute l'option -v à la commande *docker run* **-d -v** *path_conteneur* **#nom-image**
- Le volume est créé dans le système local dans /var/lib/docker/volume
- On peut spécifier le répertoire local *docker run* **-d -v** *path_local***:***path_conteneur* **#nom-image**

Persistence des données : Volumes

- Plutôt que de spécifier un point de montage manuellement, il va être plus simple de créer des volumes nommés. Nous gérerons ces volumes à l'aide des commandes préfixée **volume**
- Création d'un volume : *docker volume create* **#nom-volume**
- Liste des volumes : *docker volume ls*
- Détail d'un volume: *docker volume inspect* **#nom-volume**
- Effacer un volume: *docker volume rm* **#nom-volume**
- Utiliser un volume dans un conteneur : docker run -v **#nom-volume:conteneur_path:option #nom-image**

Docker : exposition des ports

- Quand votre conteneur doit rendre disponible certains port utile à votre application
- L'option - P de docker run permet d'attribuer de façon automatique des ports de l'hôte vers les ports du conteneur.
- L'option -p port hote:port conteneur permet de préciser le port désiré
- La commande *docker port* **#container-id** permet de donner la cartographie des ports associés à ce conteneur
- docker inspect **#container-id** contient une section dédiée au mapping de port

Docker et le réseau

- Par défaut, un réseau Bridge est mis en place, avec une affection automatique d'IP
  - l'interface réseaux dans le conteneur est eth0
  - L'interface dans la machine host est docker0
  - Le sous réseau utilisé par défaut est le 172.17.0.0/16
  - La première adresse du sous réseaux est réservé à la machine hôte

Docker et le réseau

- Commande réseaux
  - *docker network ls* : liste les réseaux disponible
  - *docker network create -d bridge - -subnet 172.24.0.0/16* **dock1** : crée le sous réseau bridge nommé dock1 sur le sous réseau 172.24.0.0/16.
  - *docker network inspect* **dock1** : permet de voir les conteneurs connectés sur ce réseau
  - *docker run -tid - -network* **dock1 #nom-image** : permet de démarrer un conteneur sur le sous réseau dock1.

Création d'un nouvelle image : le Dockerfile

- Démarrer une image et agir à l'intérieur c'est bien, mais il est préférable de préparer le maximum de choses en amont lors de la création de l'image.
- Exemple : serveur nginx avec un nouveau fichier index.html
- Il faudrait, charger une image docker de nginx, puis copier les fichiers html dans le répertoire actif du serveur web. Et cela à chaque création d'un conteneur.
- On peut aussi créer une nouvelle image à l'issue de la copie avec le conteneur ainsi créé.
  - *docker commit* **#container_id(ou name) #nom_nouvelle_image**
- Mais cela revient à spécialiser des images, ce qui n'est pas le but recherché.

Création d’image : le Dockerfile

- Le Dockerfile, décrit l’ensemble des opérations nécessaires pour créer la nouvelle image
- Chaque image docker disponible sur le hub est décrite par son Dockerfile

```
#
# NOTE: THIS DOCKERFILE IS GENERATED VIA "update.sh"
#
# PLEASE DO NOT EDIT IT DIRECTLY.
#
FROM debian:bullseye-slim

LABEL maintainer="NGINX Docker Maintainers <docker-maint@nginx.com>"

ENV NGINX_VERSION   1.23.3
ENV NJS_VERSION     0.7.9
ENV PKG_RELEASE     1~bullseye

RUN set -x \
# create nginx user/group first, to be consistent throughout docker variants
    && addgroup --system --gid 101 nginx \
    && adduser --system --disabled-login --ingroup nginx --no-create-home --home /nonexistent --gecos "nginx user" --shell /bin/false --uid 101 nginx \
    && apt-get update \
    && apt-get install --no-install-recommends --no-install-suggests -y gnupg1 ca-certificates \
    && \
    NGINX_GPGKEY=573BFD6B3D8FBC641079A6ABABF5BD827BD9BF62; \
    found=''; \
    for server in \
        hkp://keyserver.ubuntu.com:80 \
        pgp.mit.edu \
    ; do \
        echo "Fetching GPG key $NGINX_GPGKEY from $server"; \
        apt-key adv --keyserver "$server" --keyserver-options timeout=10 --recv-keys "$NGINX_GPGKEY" && found=yes && break; \
    done; \
    test -z "$found" && echo >&2 "error: failed to fetch GPG key $NGINX_GPGKEY" && exit 1; \
    apt-get remove --purge --auto-remove -y gnupg1 && rm -rf /var/lib/apt/lists/* \
    && dpkgArch="$(dpkg --print-architecture)" \
    && nginxPackages=" \
        nginx=${NGINX_VERSION}-${PKG_RELEASE} \
        nginx-module-xslt=${NGINX_VERSION}-${PKG_RELEASE} \
        nginx-module-geoip=${NGINX_VERSION}-${PKG_RELEASE} \
        nginx-module-image-filter=${NGINX_VERSION}-${PKG_RELEASE} \
        nginx-module-njs=${NGINX_VERSION}+${NJS_VERSION}-${PKG_RELEASE} \
    " \
    && case "$dpkgArch" in \
        amd64|arm64) \
# arches officialy built by upstream
            echo "deb https://nginx.org/packages/mainline/debian/ bullseye nginx" >> /etc/apt/sources.list.d/nginx.list \
            && apt-get update \
            ;; \
        *) \
# we're on an architecture upstream doesn't officially build for
# let's build binaries from the published source packages
            echo "deb-src https://nginx.org/packages/mainline/debian/ bullseye nginx" >> /etc/apt/sources.list.d/nginx.list \
            \
# new directory for storing sources and .deb files
            && tempDir="$(mktemp -d)" \
            && chmod 777 "$tempDir" \
# (777 to ensure APT's "_apt" user can access it too)
```

Commande Dockerfile

- Le fichier doit s'appeler Dockerfile,
- Il commence par une commande FROM qui indique l'image qui sert de base à la construction
- RUN permet d'exécuter des commandes à la création de l'image
- CMD permet d'exécuter des commandes au lancement du conteneur
- ENTRYPOINT définit la commande à exécuter en premier (processus initial du conteneur)
- EXPOSE permet de définir les ports à mapper. Il faut ajouter la commande -P à *docker run* pour que le mapping soit fait automatiquement
- COPY permet de copier un fichier local dans l'image.
- VOLUME permet d'ajouter un volume à notre conteneur. Ce volume sera créé automatiquement dans /var/lib/docker/volumes/ et lié au chemin indiqué. L'option -v host_path:container_path permettra de définir quel répertoire local sera utilisé.
- WORKDIR permet de définir le répertoire courant lors du démarrage du conteneur. Par défaut, c'est la racine du système qui est définie.

Docker et l’orchestration

- Dans une architecture micro-services, il est important de pouvoir construire un groupe de conteneurs regroupant l’ensemble des services à déployer et étant dépendants les uns des autres.
- Plutôt que de construire et démarrer les conteneurs à la main, nous pouvons définir une suite logique d’action permettant de récupérer les images, de les configurer et de les démarrer en respectant les relations entre elles.
- C’est le rôle de l’orchestration. Des logiciels d’orchestration sont Docker Swarm, Kubernetes, mais dans une moindre mesure, docker compose permet aussi de faire cela