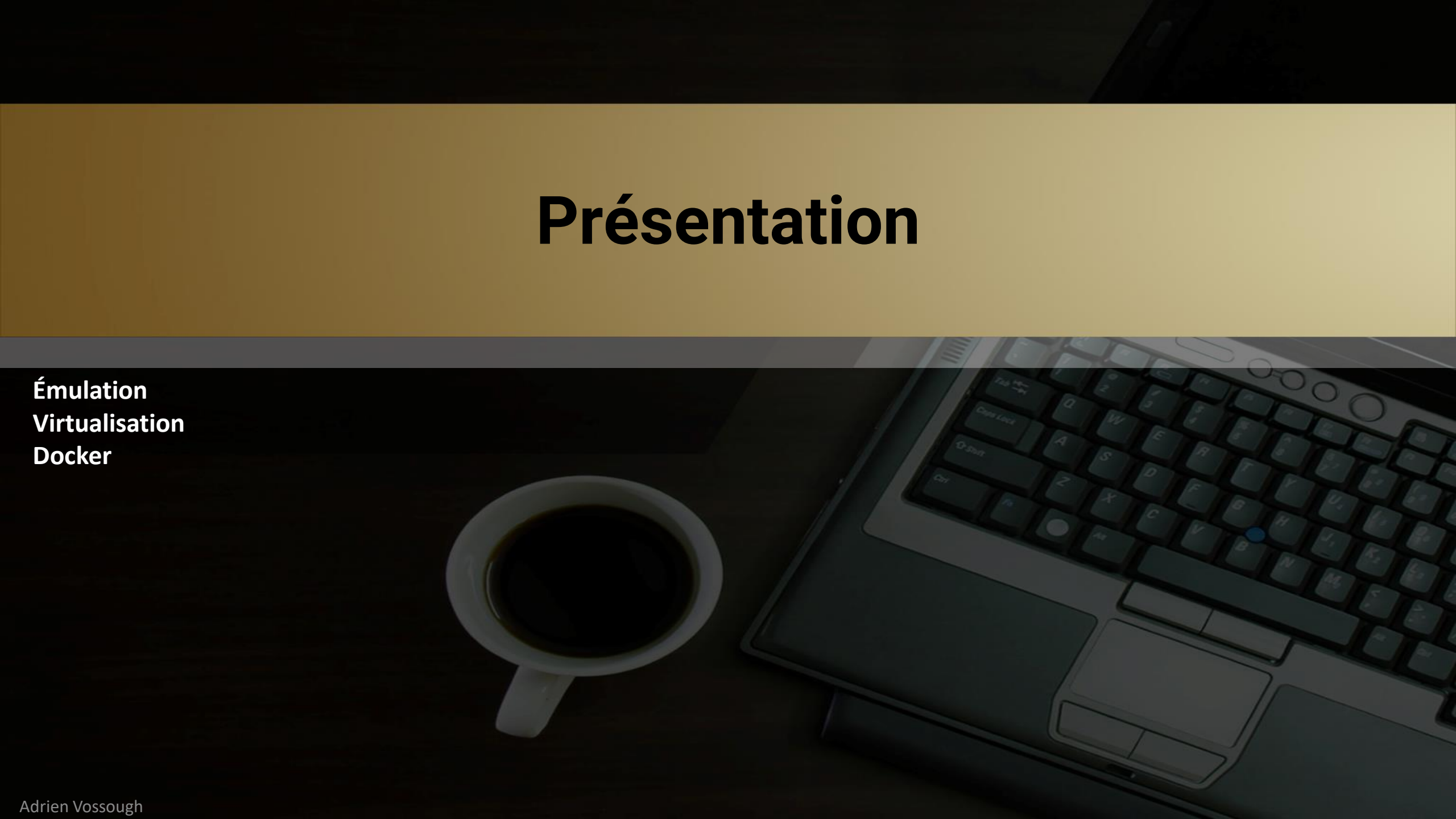
Adrien Vossough

Présentation



Émulation
Virtualisation
Docker

Adrien Vossough

Chaque processeur a sa propre architecture : langage, instructions, organisation de la mémoire, etc.

Il existe des **"familles" de processeurs**, c'est-à-dire une architecture relativement proche : X86, AMD64 (compatible X86), ARM, PowerPC, etc.

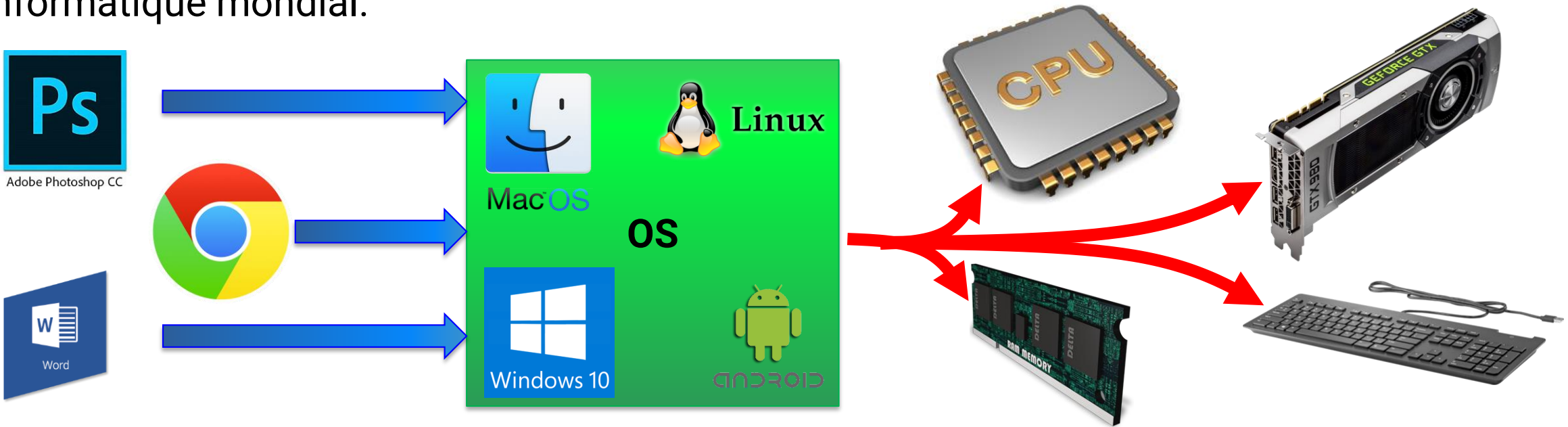
Un **programme** est développé puis **compilé** (traduit) pour une **famille de processeur** et ne fonctionnera donc pas pour d'autres types s'ils ne sont pas compatibles.

En plus de cela, lors de la phase de compilation, des fonctionnalités d'un OS peuvent être utilisé, ce qui rend un programme dépendant de l'OS et de la famille de processeur pour lequel il a été compilé.

Chaque processeur a ses propres instructions, chaque périphérique a son propre protocole.

Le système d'exploitation (OS) est une application dont le but premier est de communiquer avec le matériel informatique (périphérique + mémoire + unité de calcul).

L'OS est une interface entre une application et le matériel, car il est quasiment impossible de créer un programme qui puisse communiquer avec l'ensemble du matériel du parc informatique mondial.

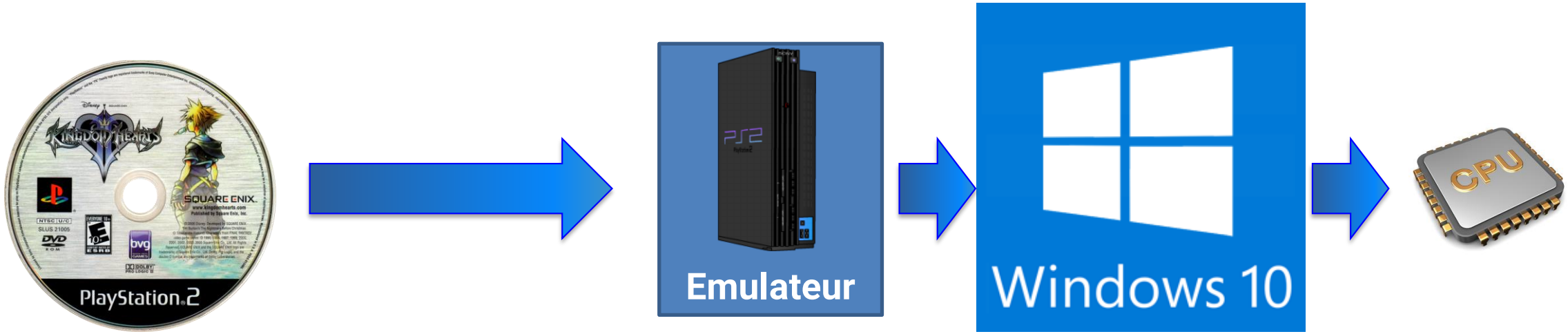


L'architecture de processeur de la PS2 se nomme : MIPS

La PS2 n'utilise pas d'OS mais une API système

Un jeu PS2 est un programme compilé pour le processeur de la PS2.
Il ne fonctionnera pas sur un autre système

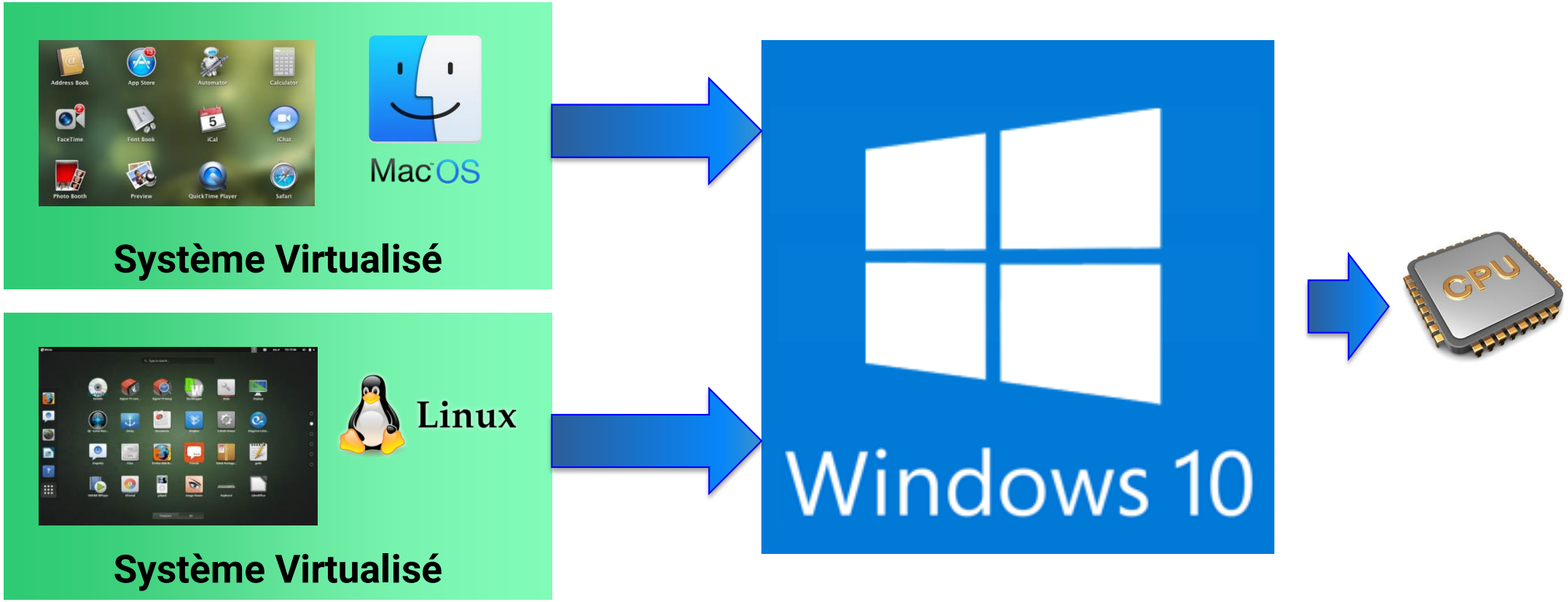
L'émulateur est une application qui va simuler un système pour faire fonctionner une application pour ce même système.



L'émulation est extrêmement gourmande en terme de puissance de calcul car elle ne fait pas que traduire un système vers un autre mais simule l'ensemble d'un système, OS compris.

A titre d'exemple, pour simuler la Super Nintendo dont le CPU est de 20Mhz, il faut un processeur 10 fois plus puissant.

La virtualisation permet de faire fonctionner sur une même machine physique plusieurs systèmes comme s'ils se trouvaient sur des machines physiques distinctes.

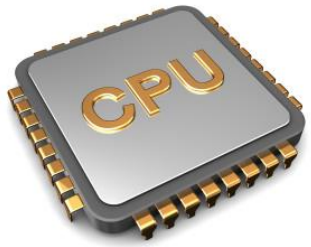


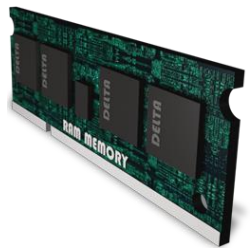
La virtualisation contrairement à l'émulation ne simule pas l'architecture hardware mais encapsule un OS et ses applications.

L'OS et ses applications sont encapsulés dans **des machines virtuelles**.

Le comportement est le même que sur un ordinateur physique.

Hyperviseur





L'hyperviseur est une application qui gère les machines virtuelles et leurs ressources (mémoire/CPU)

La virtualisation

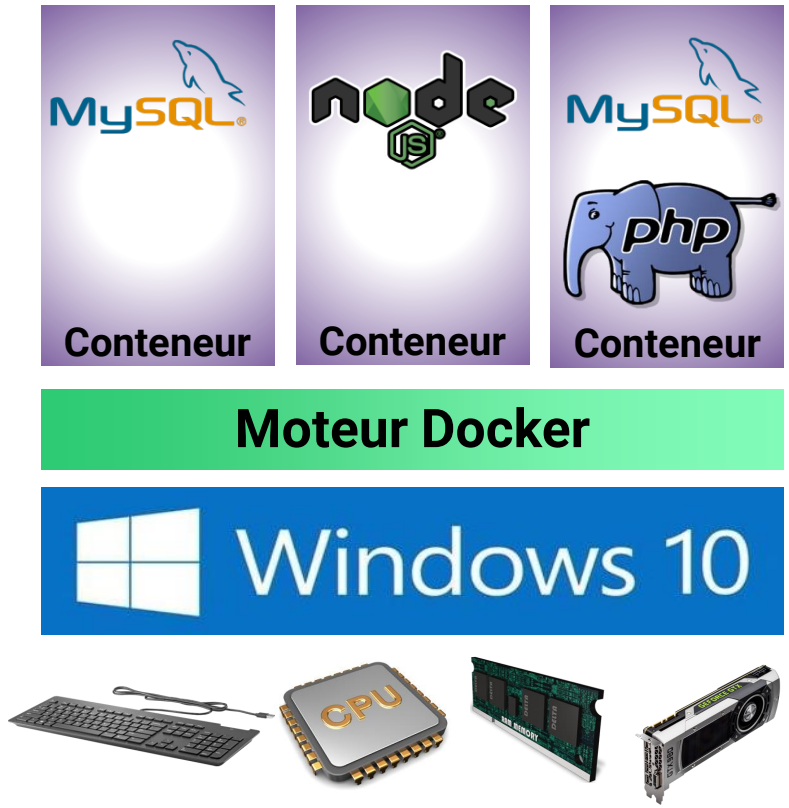
- Est énormément utilisé pour enfermer des systèmes sur un même ordinateur sans qu'aucun ne puisse interférer avec les autres.

- Est plus rapide que l'émulation, avec certains hyperviseurs les performances sont presque à 100% de celle de l'ordinateur,

- L'OS virtualisé consomme autant de mémoire, de CPU et d'espace disque que s'il était directement installé sur la machine physique.

Docker n'émule ni ne virtualise un système, il permet l'encapsulation d'applications ; nous parlons de virtualisation légère.

Il n'y a pas d'installation d'OS car les applications sont dépendantes de l'OS déjà présent sur le système.



Si nous sommes sous Windows, seules des applications Windows peuvent être installées.

Avec Docker, l'absence d'OS allège la virtualisation (RAM, CPU, Disque Dur).

Le moteur docker / docker engine / docker daemon est le cœur de Docker

- exécute les conteneurs
- gère l'isolation
- sécurise des conteneurs
- gère les ressources des conteneurs : mémoire, CPU

Le client docker pour se connecter au moteur et lui soumettre des actions.

- Peut être installé localement ou à distance.
- Existe en ligne de commande ou sous forme graphique (Windows/Mac)

Image Docker : Fournit les applications à installer dans un conteneur

- lecture seulement.
- peut être téléchargée ou fabriquée par nous
- doivent être présentes sur le serveur qui exécute docker.
- distribuées par des registres

registres ou des serveurs d'images

- le serveur par défaut est : **hub.docker.com**

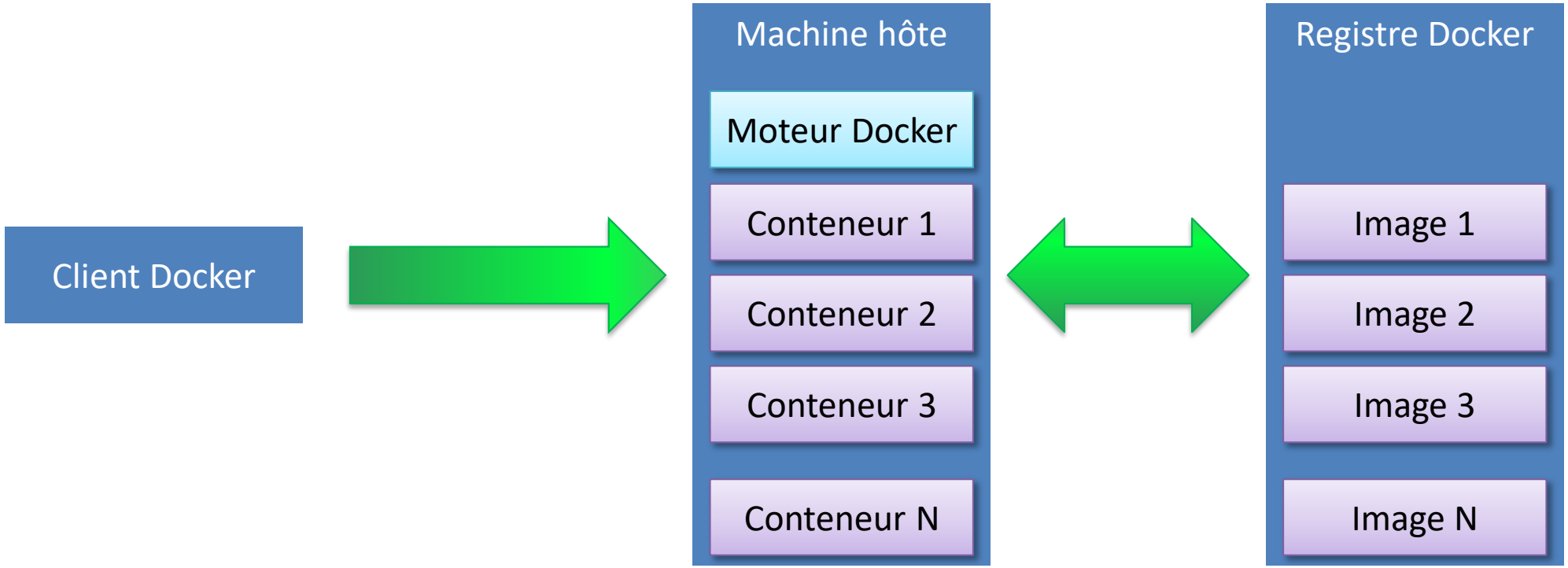
Un conteneur est un espace d'exécution, isolé et étanche par rapport aux autres.

- contient l'application à exécuter et les dépendances pour exécuter cette application.
- complet, indépendant et facile à redéployer sur un autre serveur docker.
- Un conteneur est une instance d'une image.

L'image égale binaire, conteneur égale processus.

L'accès aux images :

- peut être ouvert et donc public sans authentification
- peut être privé et demande à s'authentifier pour accéder à l'image ou récupérer l'image, ou pour pouvoir aussi la modifier.

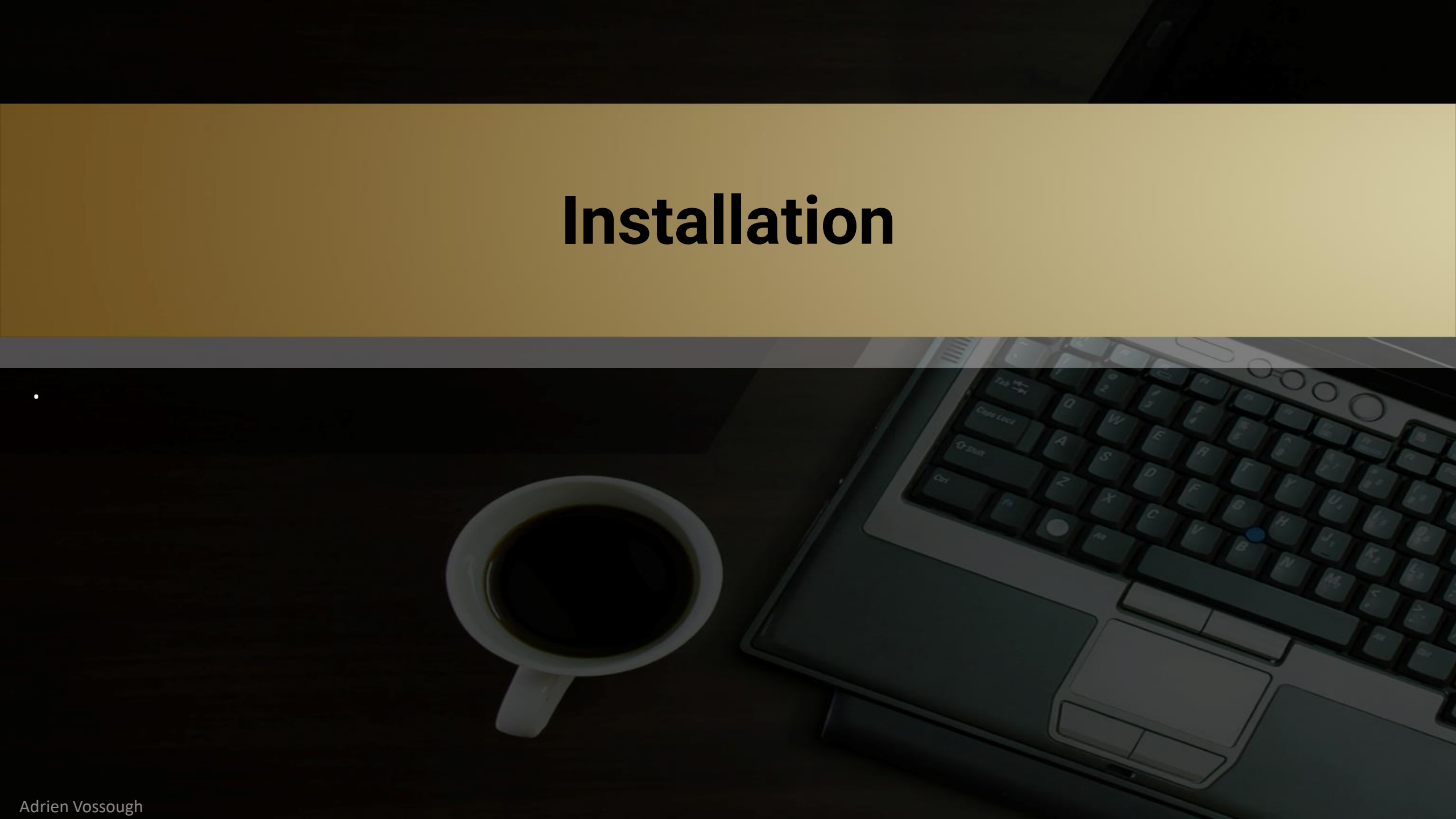


Le client Docker se connecte à l'hôte et lui soumet des requêtes comme fabriquer un conteneur à partir d'une image.

Le Docker Daemon vérifie que l'image est stockée localement, sinon va la récupérer depuis le Docker Registry et se charge de démarrer et arrêter les conteneurs.

Le conteneur est dépendant de **l'image,** cette dernière ne peut donc pas être supprimée

Installation



Adrien Vossough

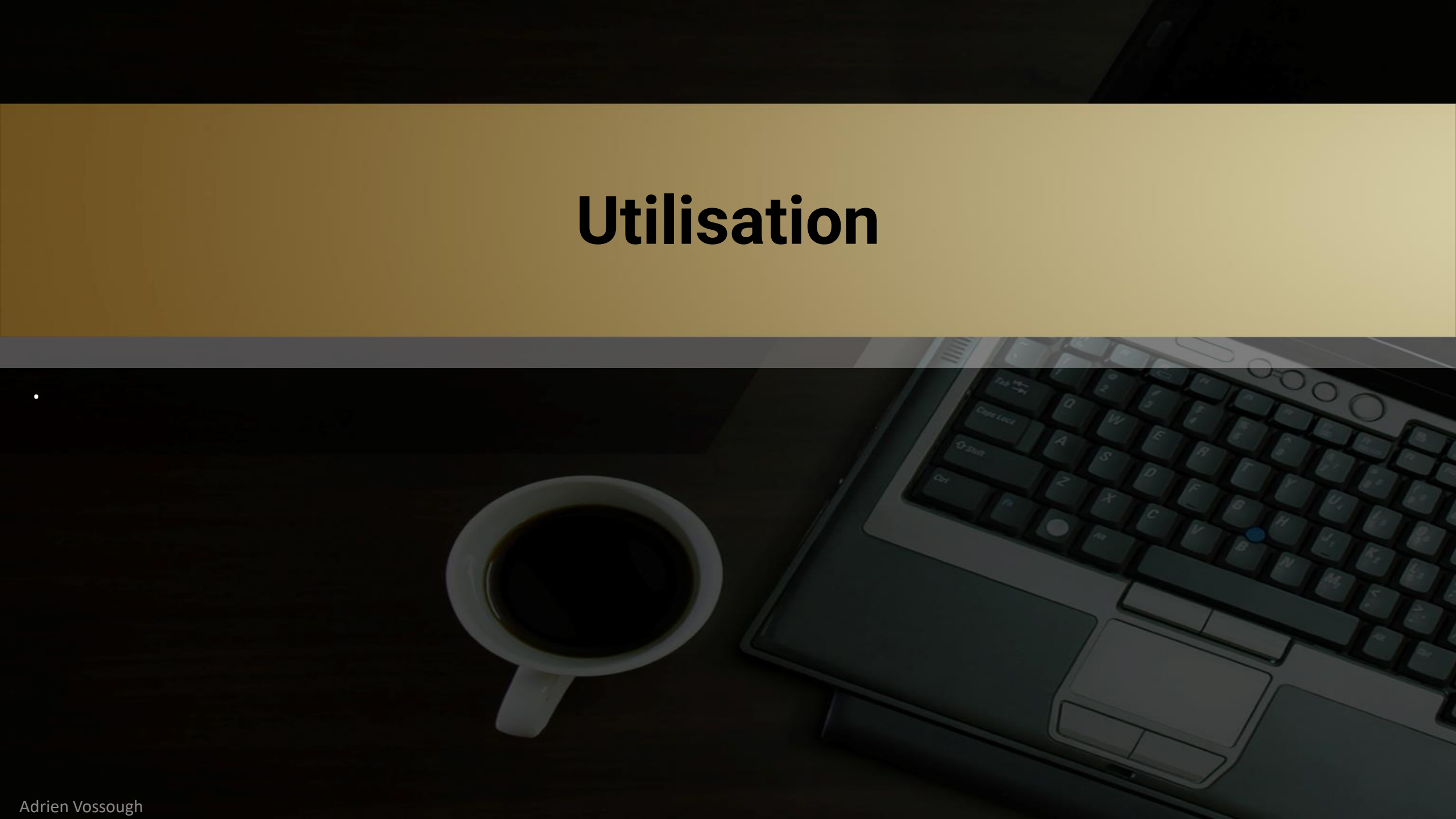
Installer Docker Toolbox pour **Windows 10 Family ou autre** **puis suivre la page d'installation** : https://docs.docker.com/toolbox/toolbox_install_windows/

Télécharger et installer Docker Community pour **Windows 10 PRO**:
https://store.docker.com/editions/community/docker-ce-desktop-windows

- Docker Daemon est démarré.
- Lancer un PowerShell

- Taper la commande : **`docker version`**
  Elle donnera la version de docker
- Taper la commande : **`docker run hello-world`**
  Va télécharger une image de test et lancer le conteneur
- Taper la commande : **`docker ps -a`**
  Indique qu'un conteneur a été créé puis supprimé

Utilisation



Après le chargement d'une image dans un conteneur :

- commande : **`docker images`**
  Affiche la liste des images. REPOSITORY nom de l'image ou du dépôt, TAG, version

- Pour rechercher une image sur le hub de Docker :
  `docker search nginx`
  NAME donne le nom de l'image, un nom sans préfixe est une image officielle, elle est dans le namespace [root]. Si le nom est préfixé , l'identifiant est un utilisateur ou une organisation qui propose l'image.
  Stars : renommée de l'image
  AUTOMATED indique si l'image est générée automatiquement grâce à un gestionnaire de code source (github, bitbucket, etc)
- Description non tronqué : **`docker search nginx --no-trunc`**

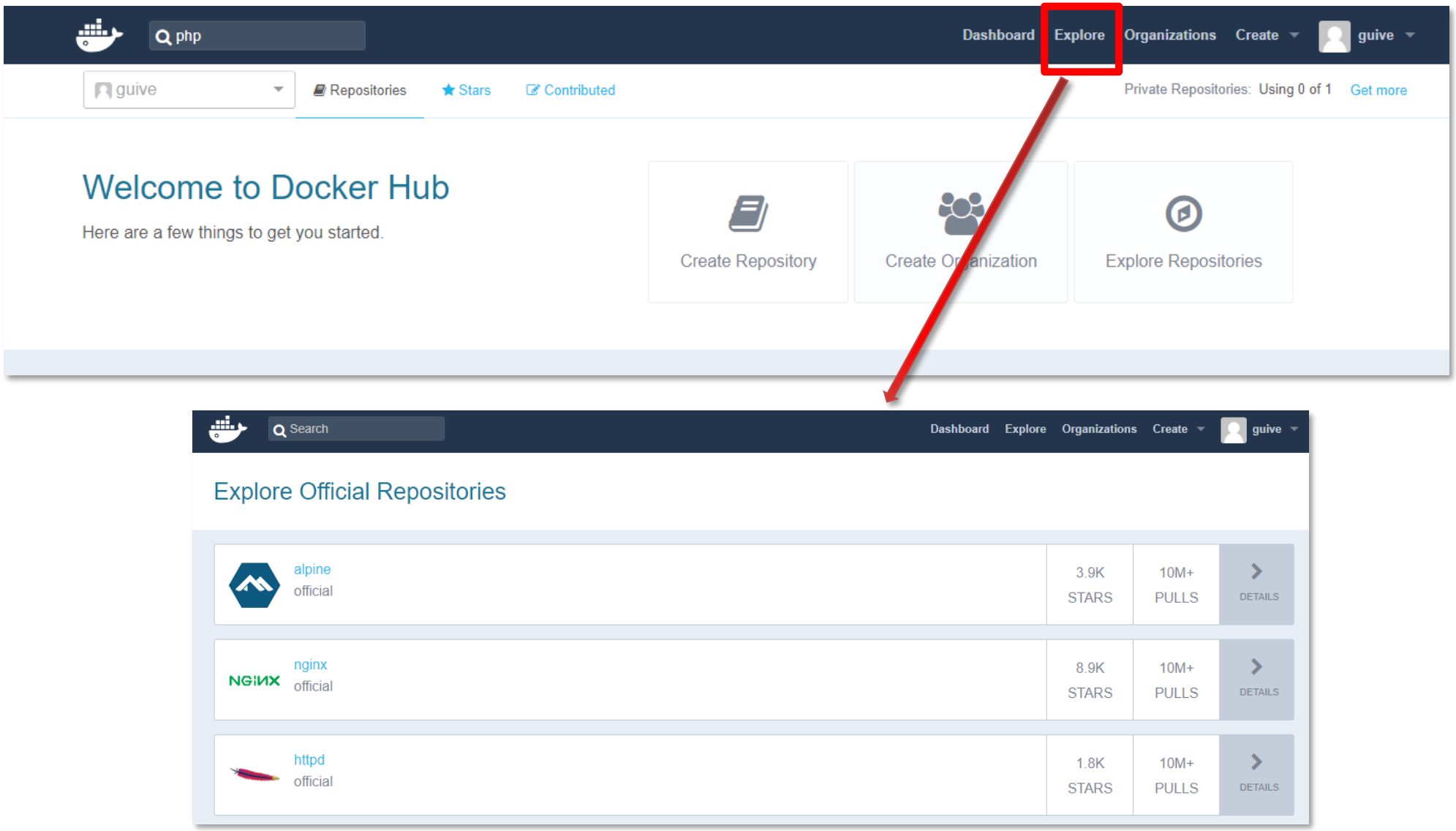
- Télécharger l'image nginx : **`docker pull nginx`**
  Sans indication, la version sera "latest" soit la dernière version sortie
  `docker pull nginx:1.12` Va installer la version 1.12

- Pour supprimer une image : **`docker rmi nginx`**

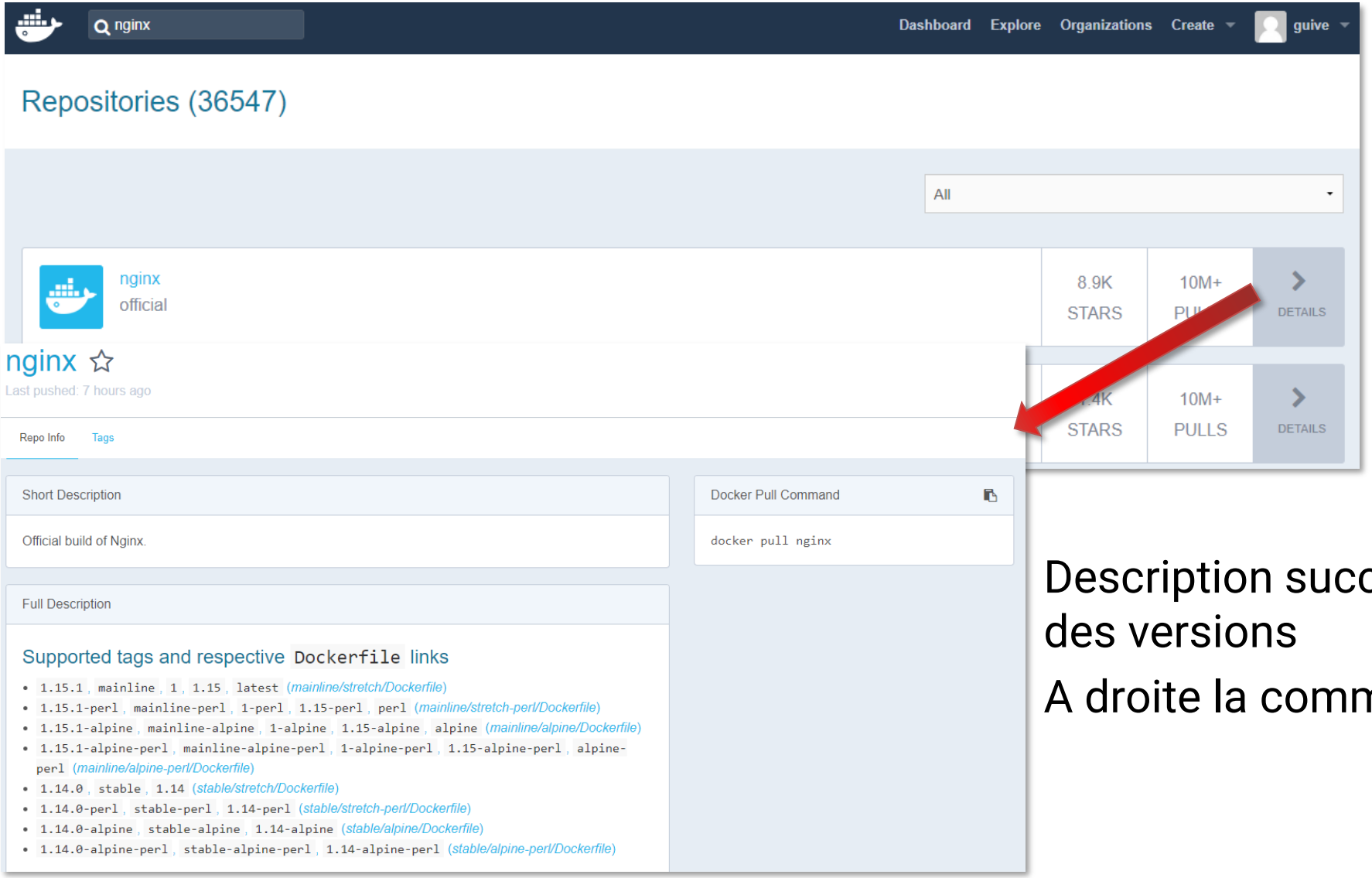
- Conteneur actif : **`docker ps`**
  Voir tous les conteneurs utilisés : **`docker ps -a`**

- Si nous tentons de supprimer une image qui a servi pour un conteneur, un conflit apparait : **`docker rmi hello-world`**
  Si nous tapons **`docker ps -a`**, un trace subsiste
  Dans ce cas, pour supprimer l'image, option force : **`docker rmi -f hello-world`**
  Attention, le conteneur ne pourra plus redémarrer

Le hub permet de trouver facilement des images : https://hub.docker.com/



Le hub permet de trouver facilement des images : https://hub.docker.com/



Stars : Renommée

Pulls : nbr de téléchargement

Description succincte à gauche et la liste des versions

A droite la commande pour télécharge

- Aide : **`docker help`**

```
Management Commands:
  config      Manage Docker configs
  container   Manage containers
  image       Manage images
  network     Manage networks
  node        Manage Swarm nodes
  plugin      Manage plugins
  secret      Manage Docker secrets
  service     Manage services
  swarm       Manage Swarm
  system      Manage Docker
  trust       Manage trust on Docker images
  volume      Manage volumes

Commands:
  attach      Attach local standard input, output, and error streams to a running container
  build       Build an image from a Dockerfile
  commit      Create a new image from a container's changes
  cp          Copy files/folders between a container and the local filesystem
  create      Create a new container
  diff        Inspect changes to files or directories on a container's filesystem
  events      Get real time events from the server
  exec        Run a command in a running container
```

Les commandes s'applique sur un contexte (liste dans management commands)

Exemple : container, network, volume sera associé à une commande attach, build etc.

- Aide des commande associé aux conteneurs : **`docker container -help`**

```
Commands:
  attach      Attach local standard input, output, and error streams to a running container
  commit      Create a new image from a container's changes
  cp          Copy files/folders between a container and the local filesystem
  create      Create a new container
  diff        Inspect changes to files or directories on a container's filesystem
  exec        Run a command in a running container
  export      Export a container's filesystem as a tar archive
  inspect     Display detailed information on one or more containers
  kill        Kill one or more running containers
  logs        Fetch the logs of a container
  ls          List containers
  pause       Pause all processes within one or more containers
  port        List port mappings or a specific mapping for the container
  prune       Remove all stopped containers
  rename      Rename a container
  restart     Restart one or more containers
```

Le contexte conteneur est celui par défaut :

`docker ps` et **`docker container ps`** donne la même chose

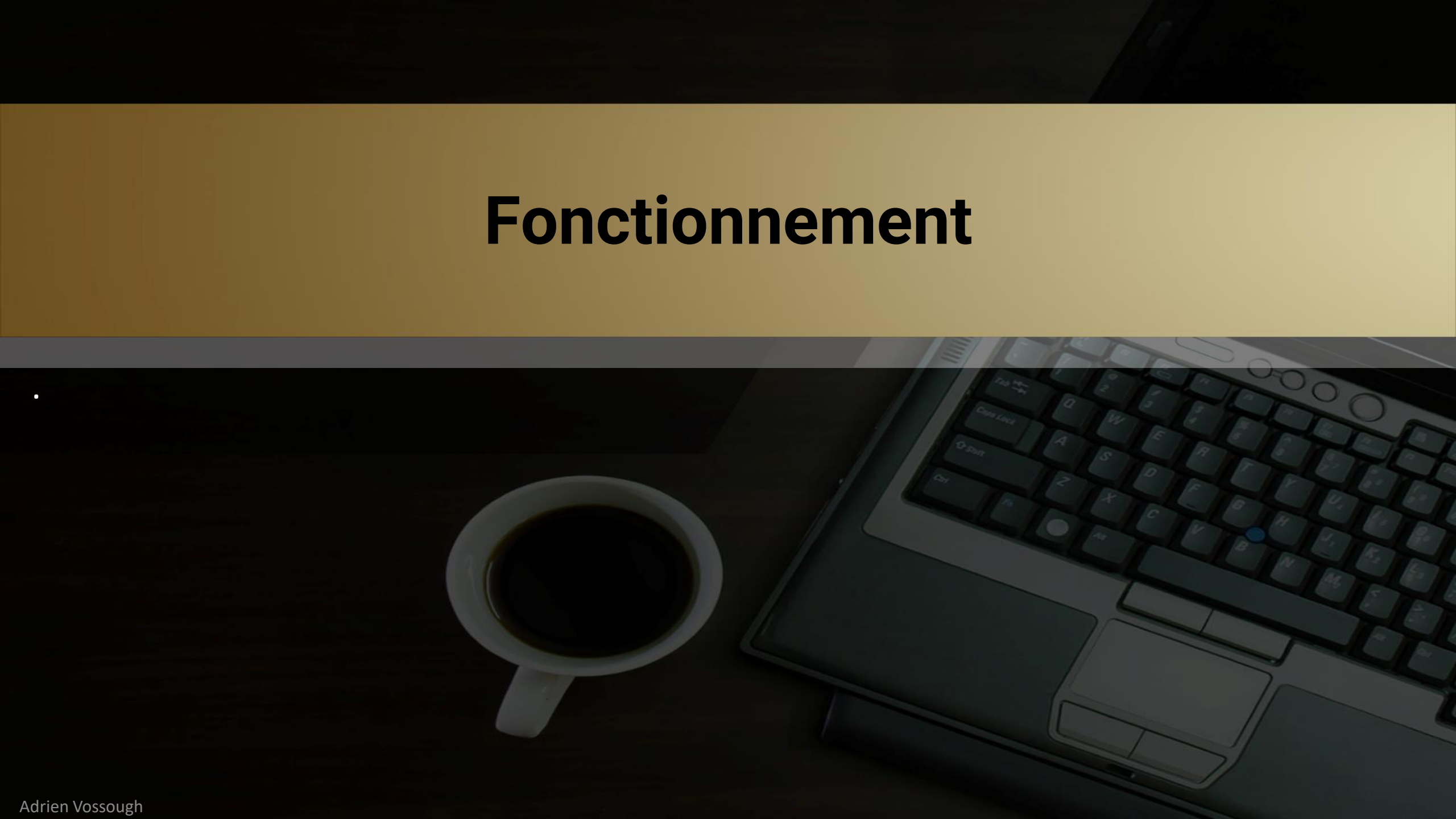
Aide sur une commande lié à un contexte (ici conteneur) : **`docker container run --help`**

```
Usage:  docker container run [OPTIONS] IMAGE [COMMAND] [ARG...]

Run a command in a new container

Options:
      --add-host list                Add a custom host-to-IP mapping
                                     (host:ip)
  -a, --attach list                  Attach to STDIN, STDOUT or STDERR
      --blkio-weight uint16          Block IO (relative weight),
                                     between 10 and 1000, or 0 to
                                     disable (default 0)
      --blkio-weight-device list     Block IO weight (relative device
                                     weight) (default [])
      --cap-add list                 Add Linux capabilities
      --cap-drop list                Drop Linux capabilities
      --cgroup-parent string         Optional parent cgroup for the
```

Nous avons une liste d'option pour une commande donnée

Parfois les options ont un raccourcis : -a pour --attach

Fonctionnement



Il existe deux cycles de vie de conteneur :

Le cycle de vie de base, utilisé par exemple, pour les compilations ou un traitement spécifique

- le conteneur est lancé à partir d'une image
- Il est créé
- Exécute un processus spécifique.
- La tâche terminée, le conteneur se termine aussi
- Le conteneur persiste sur le système avec l'état terminé et n'est plus en mémoire
- La tâche réalisée, le conteneur peut être détruit avec docker rm, ou lors du lancement, ajouter l'option - - rm pour le détruire automatiquement lorsqu'il se termine.

Le cycle de vie avancé :

- Application sous forme de service comme un serveur
- le conteneur est créé et exécute un daemon ou un service tel que HTTPD

`docker run` : commande pour lancer un conteneur

Pour utiliser des images de type système comme Ubuntu :

- **`docker run ubuntu`**
  Il ne contient pas un noyau Ubuntu mais un environnement et ne contient que la partie binaire et la partie library stockées dans une arborescence au format Ubuntu.

- On remarque que le container s'est arrêté car il lance un script bash mais n'a pas trouvé le terminal
  `docker ps -a` : colonne commands "/bin/bash"

- Il faut donc ajouter les options -ti qui demande de démarrer un terminal et de se connecter de dessus (i pour interactif)
  `docker run -ti ubuntu`
  - Taper **`ps`** qui est une commande Bash pour voir les processus
  - **`hostname`** : retourne le nom d'hôte qui est ici l'ID du conteneur car il se comporte comme un serveur

- Pour quitter le mode terminal, commande "exit

- Il est possible de lancer une commande bash directement :
  `docker run -ti ubuntu ps`

- **`docker run --name=cont01 --hostname=hostest -it ubuntu`**
  permet de définir un nom de conteneur et son hostname
  sortir avec "**`exit`**" et lancer "**`docker ps -a`**" pour vérifier le "name"

- Par défaut, Docker lance un conteneur en mode attaché, sauf avec l'option -d
  `docker run -d -ti ubuntu ps`

  Pour récupérer la sortie standard (logs ne sont pas des journaux) :
  `docker logs idContainer`

- En mode attaché :
  `docker run -a -ti ubuntu ps`
  ou **`docker run -ti ubuntu ps`**

- Pour pouvoir passer du mode attaché au mode détaché, on utilise la combinaison
  Ctrl+P+Q

Nous allons mettre notre docker Ubuntu à jour :

- **`docker run -ti  ubuntu`**
  Démarre le terminal en mode interactif (on se connecte dessus)
- **`apt-get update`**
  On met à jour le gestionnaire de dépendance
- **`apt-get install iputils-ping`**
  Nous installons la commande "ping"

Lançons un ping sur le conteneur lui-même :

- **`ping 127.0.0.1`**

Nous sommes en mode attaché, pour passer du mode attaché au mode détaché, utiliser la combinaison Ctrl+P+Q

`docker ps`, le conteneur est toujours exécuté mais en mode détaché

Pour se rattacher :

- **`docker attach idDuConteneur`**

- **`docker run -d -P nomConteneur`**
  -d en mode detach,
  -P permet de **mapper les ports** du conteneur sur ceux du hôte : les clients externes peuvent se connecter au conteneur en passant par l'hôte
  nginx est un serveur http (comme Apache)

- **`docker stop Idconteneur`** ou **`docker kill Idconteneur`**
  arrête un conteneur
  La commande stop envoie un signal de terminaison ou PID1, qui est le premier processus de la hiérarchie des processus à l'intérieur du conteneur. C'est un arrêt propre/normal.
- **`docker kill --signal=9 Idconteneur`**
  Arrêt sale, force la fermeture du conteneur s'il est bloqué.
  Le statut donné par docker ps -a donnera : Exited (137) About a minute ago

- **`docker start IDconteneur`**
  Redémarre un conteneur arrêté

- **`docker exec -it IDConteneur bash`**
  exec permet de lancer un **processus en parallèle** sur un conteneur.
  On se connecte à un terminal en lançant la commande bash

- **`docker inspect IDConteneur`**
  Retourne un objet JSON, **descriptif du conteneur** (nom, réseau/mapping, IP)

- **`docker inspect --format='{{.State.Status}}' IDConteneur`**
  Retourne **un champ du descriptif**, ici, le champ Status de l'objet State

- **`docker inspect --format='{{.State}}' centos0`**
  Retourne **l'objet State du descriptif** mais ne contient que les valeurs

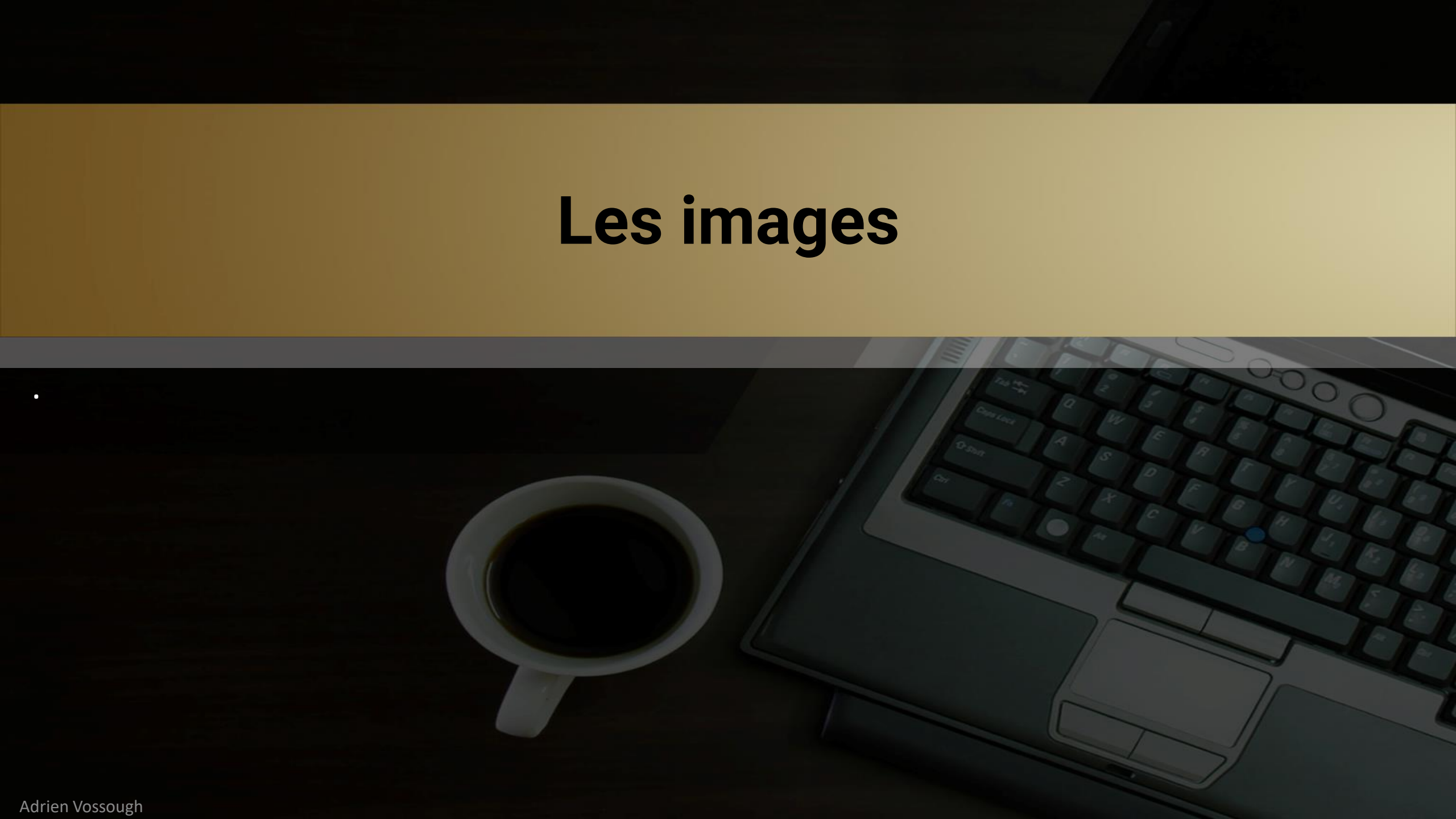
- **`docker inspect --format='{{json .State}}' centos0`**
  le mot clé json indique que l'objet State s'affichera au **format JSON, clé valeur**

- **`docker ps`**
  Affiche les conteneurs actifs
- **`docker ps -a`**
  Afficher tous les conteneurs
- **`docker ps -l`**
  Afficher le dernier conteneur créé
- **`docker ps -q`**
  Afficher que l'ID des conteneurs
- **`docker ps -a --filter name=`*`PartieNomConteneur`*
  --filter filtre la sortie d'une commande
  ici, filter ne garde que les noms des conteneurs contenant " PartieNomConteneur"
- **`docker ps -a --filter exited=`*`137`*
  Ne garde ques les conteneurs ayant le code de sortie 137
- **`docker ps -a --format "{{.Names}} : {{.Status}}"`**
  Modifie l'affichage pour n'avoir que le nom et l'état des conteneurs

Il est possible de lier les commandes :

- `docker rm -f $(docker ps -aq --filter status=exited)`
  Nous forçons la suppression de tous les conteneurs dont le status est "exited"

Les images



Adrien Vossough

Les conteneurs contiennent :

- **Une image est un template en lecture seule** qui contient l'OS souhaité sans le kernel(noyau) et un ensemble d'applications
- L'application est l'élément du conteneur qui contient les modifications

Une image :

- est une collection de fichiers stockés dans des systèmes de fichiers : fichiers binaires, fichiers de configuration et librairies.
- est composée de plusieurs couches, toutes en lecture. Chaque couche représente une brique logicielle qui va se rajouter à la couche inférieure.

Le kernel est la partie d'un OS qui gère le matériel. Les images Docker n'en ont pas, ce sont des "OS light".

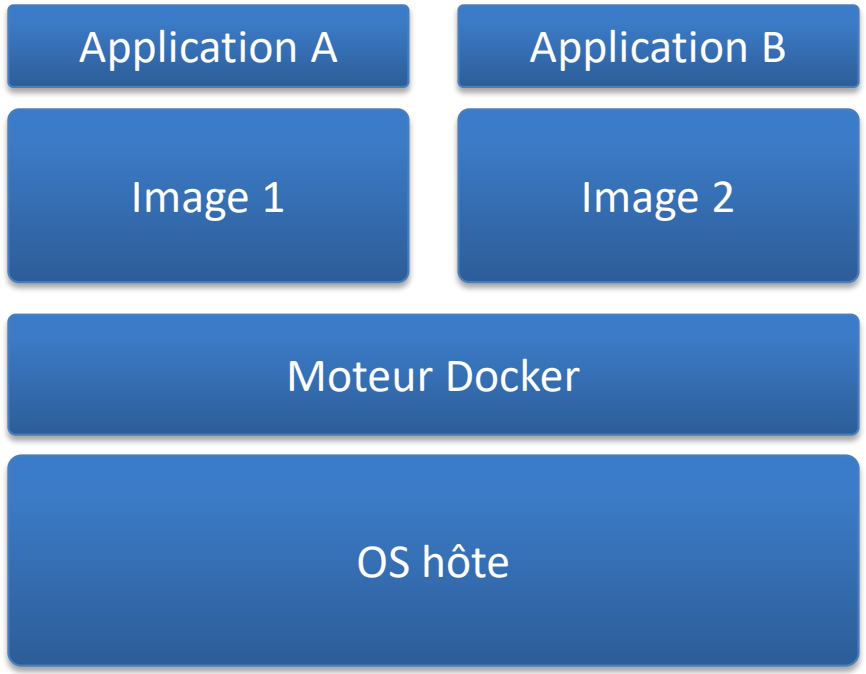
Application en lecture/écriture qui contient :
Toutes les modifications d'une image

image en lecture seule qui contient :
un OS light, des binaires (applications) et bibliothèques

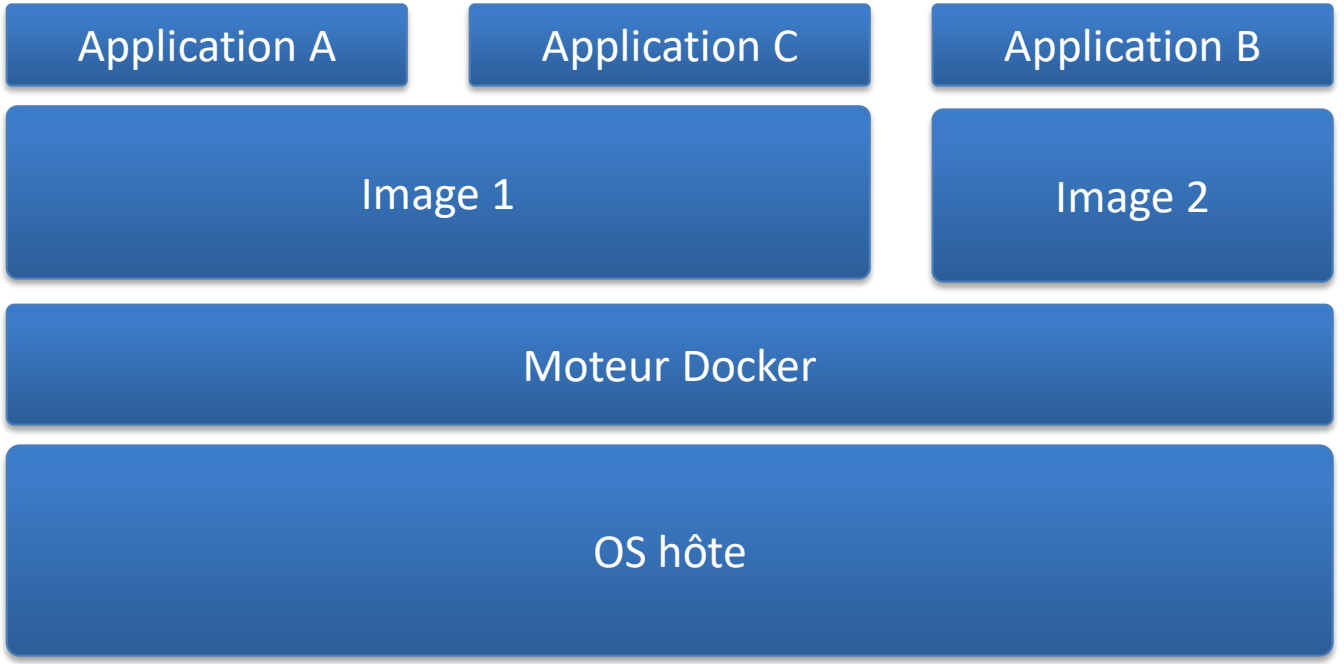
Moteur Docker *qui gère les conteneurs : Création, suppression, téléchargement*

Système d'exploitation hôte : *OS où docker est installé. Docker utilise son noyau pour communiquer avec le matériel*

Le kernel est la partie d'un OS qui gère le matériel. Les images Docker n'en ont pas, se sont des OS "light".



Applications avec des images différentes



Si plusieurs application utilise la même image

Avec la virtualisation :
une machine virtuelle = une copie de l'image complète
Et chaque image comprend un OS complet avec l'ensemble des applications

Une image est composée de couches :

Ces couches ne peuvent pas être modifié dans un conteneur.

Lors de la création d'un conteneur une dernière couche est appliquée, seule celle-ci pourra être modifiée

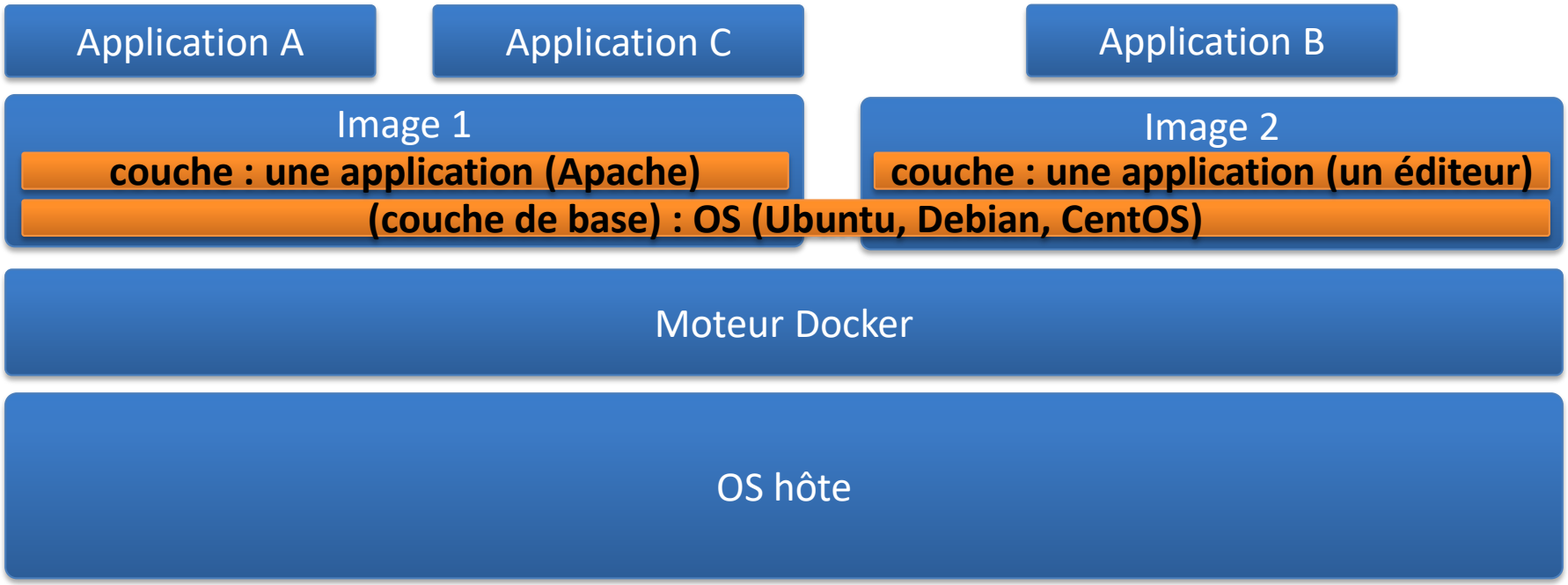
.

Si le conteneur veut modifier un fichier qui se trouve dans une couche basse (partie image), le fichier est d'abord copié sur la couche haute, en écriture, ensuite il sera possible de le modifier à ce niveau-là.

Si on accède à un fichier, dans la partie image, uniquement en lecture dans la part, il ne sera pas copié en écriture.

.

Plusieurs images peuvent se partager des couches :



Cela rend les conteneurs encore plus léger.

Pour créer sa propre image, plusieurs solutions :

- Créer un conteneur, apporter nos modifications, arrêter le conteneur et utiliser la commande **Docker commit**.
  La couche en lecture/écriture est ajouté à l'image est devient une couche en lecture.
  Cette technique est dite dynamique.

- Créer un fichier de description **Dockerfile** qui détaille la construction de l'image.
  Cette technique est dite statique, et permet de gérer, par exemple les versions des images pour chaque version de Dockerfile.
  Permet de garder une trace des modifications apportées à l'image.

- A partir d'une **archive** qui contient les fichiers que l'on souhaite intégrer dans l'image.

Espace de nommage(namespace) des images :

- Le premier est le namespace Root qui est reservé à Docker et aux organisations officielles (Apache, Ubuntu, etc.) et n'aura pas de préfixe.

- Pour les autres organisations/utilisateurs, les images sont préfixées par le nom du compte Docker Hub

Il est possible de créer son propre serveur "registry" (dépôt d'image) et avoir son propre espace de nommage.

Création d'une image à partir d'un conteneur

- **`docker run -ti --name=mon-conteneur ubuntu`**
  Fabrication d'un conteneur à partir d'une image. Ce conteneur est nommé

Arrivé dans la shell du conteneur, il est possible d'installer les applications avec apt-get

- **`apt-get update && apt-get upgrade && apt-get dist-upgrade -y`**
  met à jour les sources de l'outil de gestion de paquets, les paquets et vérifie leur intégrité

- Installer les applications manquantes (exemple : php7, apache, etc.)

- Taper "**`exit`**" pour sortir du conteneur

- **`docker ps -a`**
  Affiche bien que le conteneur est arrêté

- **`docker diff IDConteneur`**
  Donne les différences entre le conteneur et l'image d'origine
  Le premier caractère donne l'opération : A pour add/ajout, C pour change/modification, D pour delete/suppression

- **`docker commit IDConteneur ma-societe/mon-image:1.0`**
  Va créer une image depuis un conteneur.
  Le nom de l'image est préfixé puisque non-officiel.
  La version est 1.0

- **`docker images`**
  liste les images présentent

- **`docker run ma-societe/mon-image:1.0`**
  Démarre l'image nouvelle créée, vérifier la présence des applications installée

Création d'une image à partir d'un Dockerfile

Un Dockerfile est un fichier de configuration qui contient les instructions pour construire une image.

Il est conseillé de faire un dossier par image.

- Créer un dossier
- Se placer dedans et créer un fichier Dockerfile

Dockerfile

```
1  # indique la premier couche de l'image
2  FROM ubuntu
3  # indique une commande à exécuter dans le conteneur
4  # pendant la création de l'image
5  # permet l'installation/configuration
6  RUN apt-get update
```

- Sauvegarder le fichier
- Placer une console dans le répertoire et taper :
  `docker build -t adrien/mon-image:1.0 .`
  créé l'image depuis le fichier Dockerfile
  -t pour le tag (libellé)
  le point indique le répertoire courant (possible de pointer un autre répertoire)

La commande docker build a construit l'image en lançant les commandes dans un conteneur temporaire qui est transformé en image.

- **`docker images`**
  Vérifier que l'image est présente

- Ajouter une nouvelle ligne **RUN**

Dockerfile

```
1  # indique la premier couche de l'image
2  FROM ubuntu
3  # indique une commande à exécuter dans le conteneur
4  # pendant la création de l'image
5  # permet l'installation/configuration
6  RUN apt-get update
7  # installe apache
8  RUN apt-get install -y apache2
```

- Relancer la commande build en ajoutant une version :
  `docker build -t adrien/mon-image:1.1 .`

Console

```
$ docker build -t adrien/mon-image:1.1 .
Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM ubuntu
 ---> 113a43faa138
Step 2/3 : RUN apt-get update
 ---> Using cache
 ---> 5b401ac421ac
Step 3/3 : RUN apt-get install -y apache2
 ---> Running in 4349114538b6
Reading package lists...
Building dependency tree...
Reading state information...
The following additional packages will be installed:
```

Part du FROM, image de base

L'étape 2 n'est pas répété car déjà faite, docker réutilise la précédente couche

exécution de la nouvelle étape

- **`docker images`**
  Vérifier que l'image est présente

- **`docker history adrien/mon-image:1.1`**
  Donne des informations sur les différentes couches d'une image. La ligne la plus basse est la première couche

Console

```
$ docker history adrien/mon-image:1.1
IMAGE               CREATED             CREATED BY                                      SIZE
COMMENT
ef51221f1a21        8 minutes ago       /bin/sh -c apt-get install -y apache2           101MB
5b401ac421ac        12 minutes ago      /bin/sh -c apt-get update                       40.9MB
113a43faa138        4 weeks ago         /bin/sh -c #(nop)  CMD ["/bin/bash"]            0B
<missing>           4 weeks ago         /bin/sh -c mkdir -p /run/systemd && echo 'do…   7B
<missing>           4 weeks ago         /bin/sh -c sed -i 's/^#\s*\(deb.*universe\)$…   2.76kB
<missing>           4 weeks ago         /bin/sh -c rm -rf /var/lib/apt/lists/*          0B
<missing>           4 weeks ago         /bin/sh -c set -xe   && echo '#!/bin/sh' > /…   745B
<missing>           4 weeks ago         /bin/sh -c #(nop) ADD file:28c0771e44ff530db…   81.1MB
```

Les commandes en rouges sont des commandes Dockerfile

ADD permet d'ajouter une archive à une image

CMD place une commande par défaut au démarrage d'une image

Dockerfile

```
# Partir d'une image de base
FROM ubuntu
# donnes des informations sur l'image
LABEL version="1.0"
LABEL description="ma premiere image"
LABEL os="ubuntu"
#initialise des variables d'environnement
ENV JAVA_HOME /usr/bin/java
# exécute une commande dans le conteneur lors de la création de l'image
RUN apt-get update
# définit le dossier de travail dans le conteneur
# lorsqu'une commande est tapée, indique où la trouver
# par défaut, le répertoire "/"
WORKDIR /opt
# copie un fichier de la source vers le conteneur de l'image
# généralement des fichiers placés dans le répertoire de création de l'image
# Permet aussi de décompresser des fichiers vers le conteneur
# Peut aussi renommer des fichiers
ADD monfichier.txt /
# lance une commande une fois le conteneur créé
# n'existe qu'une fois dans un fichier Dockerfile
CMD [ "ping", "127.0.0.1" ]
```