جامعة الحسن الأول
UNIVERSITÉ HASSAN 1er

كلية العلوم و التقنيات سطات
FACULTÉ DES SCIENCES ET TECHNIQUES SETTAT

RAPPORT DOCKER

docker

Sous la supervision de : Mme MIHI_SOUKAINA

Réalisé par : El kasimi Basma

El Odmi Salma

Anba Hanan

Yakine Chouaib

TABLE DES MATIERES

I-Présentation générale du Docker

Docker permet de créer des environnements (appelés conteneurs, ou *containers* en anglais) de manière à isoler des applications. Docker repose sur le kernel Linuxet sur une fonctionnalité : les *containers*, que vous connaissez peut-être déjà sous le nom de LXC. L'idée est d'exécuter un processus (ou plusieurs) dans un environnement isolé. Le conteneur qui accueille votre application aura un système de fichier qui lui est propre (pouvant contenir uniquement les fichiers nécessaires à l'application) et aura accès à certaines ressources du système hôte (mémoire, CPU, réseau). Avec Docker, on va pouvoir très simplement gérer nos différents conteneurs.

II-Historique rapide de Docker

Docker a été lancé en 2013 par la société DotCloud, qui a ensuite changé de nom pour devenir Docker Inc.. À l'origine, Docker s'appuyait sur des technologies existantes de virtualisation légère du noyau Linux, notamment LXC (Linux Containers). L'objectif était de simplifier le processus de création, de déploiement et d'exécution d'applications en les encapsulant dans des conteneurs légers, portables et reproductibles.
En 2014, Docker a connu une adoption rapide dans la communauté des développeurs grâce à sa simplicité et son efficacité. Il a ensuite remplacé LXC par sa propre bibliothèque d'exécution de conteneurs, appelée libcontainer (aujourd'hui runc), pour avoir un meilleur contrôle sur le fonctionnement des conteneurs.
Depuis, Docker s'est imposé comme un outil de référence dans le monde du DevOps et de l'intégration/déploiement continu (CI/CD). Il est aujourd'hui utilisé dans une grande variété de contextes, allant du développement local jusqu'au déploiement en production dans des environnements cloud et multi-serveurs.

III-À quoi sert Docker en pratique ?

Dans un contexte de développement logiciel, il est courant que chaque développeur travaille sur une machine avec une configuration différente (système, bibliothèques, dépendances, etc.). L'application en cours de développement peut donc réagir différemment d'un poste à un autre. De plus, les environnements de test et de production utilisés par l'entreprise sont souvent standardisés et présentent d'autres différences par rapport à l'environnement local des développeurs.

Cela peut entraîner des problèmes d'incompatibilité entre les différents environnements, rendant le passage du développement à la production complexe, voire risqué.

C'est ici que Docker intervient. Grâce à la conteneurisation, il est possible d'encapsuler l'application avec toutes ses dépendances, fichiers et configurations nécessaires dans un conteneur unique. Ce conteneur peut ensuite être exécuté de manière identique, peu importe l'environnement (local, test, production), sans avoir à modifier quoi que ce soit.

L'utilisation de Docker permet donc :

- **→ d'éviter les "bugs liés à l'environnement",**
- **→ de réduire les écarts entre les différentes phases du cycle de vie logiciel,**
- **→ d'assurer une meilleure continuité entre développement, test et déploiement.**

En résumé, Docker simplifie le travail des équipes, garantit une plus grande fiabilité des applications et réduit les coûts liés à la gestion des environnements.

IV-Principes de Docker :

Docker repose sur trois grands principes qui font sa force : la reproductibilité, l'isolation et la portabilité.

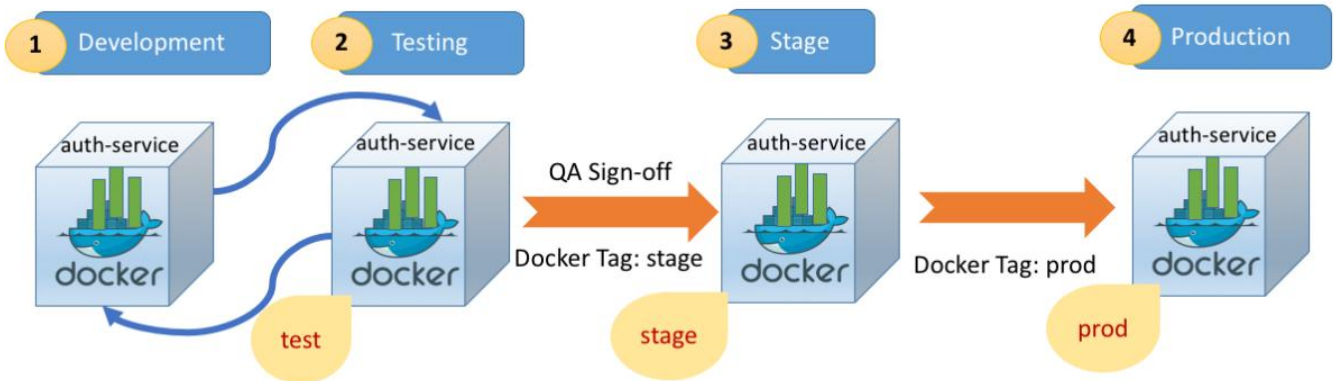
→ **Reproductibilité : Docker permet de créer un environnement identique entre le développement, les tests et la production. Cela élimine les problèmes classiques du type "ça fonctionne sur ma machine" en garantissant que l'application se comportera de la même façon quel que soit l'environnement.**

→ **Isolation : Chaque conteneur fonctionne de manière indépendante, avec ses propres processus, bibliothèques et fichiers. Cela signifie qu'il est possible d'exécuter plusieurs applications ou services sur la même machine sans qu'ils n'interfèrent les uns avec les autres.**

→ **Portabilité : Les conteneurs Docker peuvent être exécutés de la même manière sur différentes machines ou plateformes (Linux, Windows, serveurs cloud, etc.), indépendamment de leur configuration. Cela facilite considérablement le déploiement sur divers environnements physiques ou virtuels.**

V-Pourquoi utiliser Docker ?

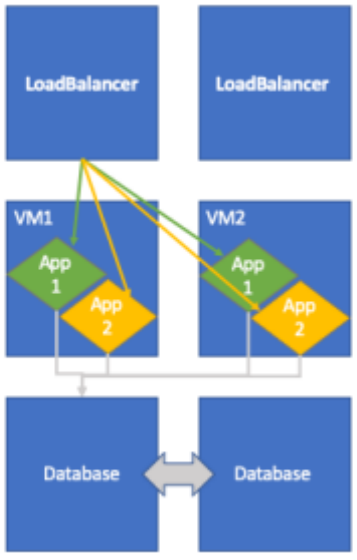
Pour les développeurs :



Dans le processus de développement, il est fréquent que les développeurs travaillent sur des environnements locaux qui diffèrent de ceux utilisés en production. Par exemple, un développeur utilisant PHP sur macOS peut rencontrer des différences de comportement lorsqu'il déploie son code sur un serveur Linux, même si les versions de PHP sont identiques. Cela s'explique par des écarts dans la configuration, la gestion des permissions ou d'autres spécificités du système d'exploitation.
Dans ces situations, plusieurs questions se posent :
Faut-il modifier l'environnement de production pour qu'il corresponde à celui du développeur ? Ou bien contraindre les développeurs à travailler dans un environnement strictement identique à la production ?
En théorie, l'idéal serait d'avoir une cohérence totale entre les différents environnements, de la machine du développeur aux serveurs de production. Mais dans la pratique, cela est difficile à maintenir, surtout lorsqu'on travaille en équipe, avec des dizaines ou des centaines de développeurs ayant chacun leurs préférences et habitudes.
Docker apporte une réponse concrète à cette problématique en permettant de créer des environnements de développement reproductibles. Chaque application peut être encapsulée dans un conteneur contenant toutes ses dépendances, ce qui garantit un comportement identique à travers toutes les étapes du cycle de vie : développement, test, intégration et production.

Pour les administrateurs système

Les administrateurs système doivent souvent gérer plusieurs services répartis sur différentes machines virtuelles (VM) : base de données, serveurs web, load balancer, etc. Imaginons que deux versions d'un même logiciel soient nécessaires pour deux modules distincts d'une application. Avec une architecture classique, ces versions peuvent entrer en conflit, car elles ne peuvent pas cohabiter facilement sur le même système.



Plusieurs solutions classiques sont envisageables :

→ Ajouter de nouveaux serveurs : coûteux et parfois impossible.

→ Réutiliser d'autres serveurs existants : cela complexifie l'architecture et augmente les risques en cas de panne.

VI-Différences entre conteneurs et VM :

1-Conteneurs :

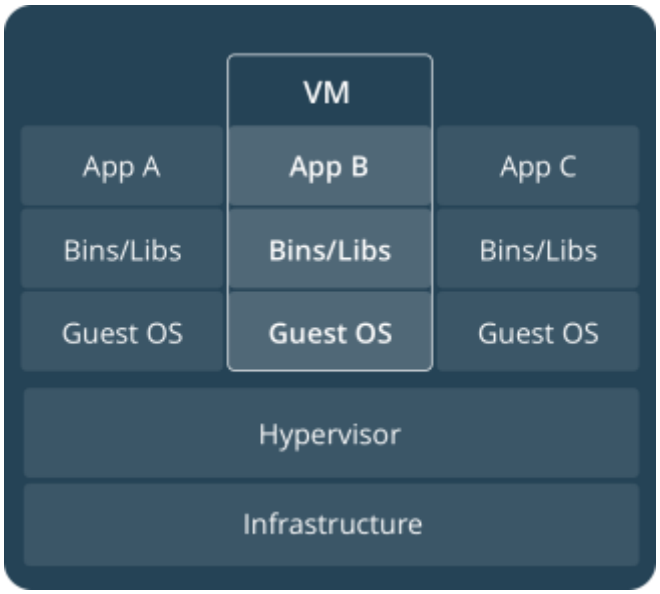
Les applications modernes nécessitent de nombreuses dépendances et doivent fonctionner sur des environnements variés. Pour gérer cette complexité, on utilise souvent des machines virtuelles qui embarquent l'application, ses dépendances et un système d'exploitation complet. Mais cette approche est lourde : les VM sont volumineuses (plusieurs Go) et consomment beaucoup de ressources.

Les conteneurs, à l'inverse, permettent d'exécuter les applications de manière isolée, mais sans embarquer un système d'exploitation complet. Ils partagent le noyau du système hôte, ce qui les rend plus légers, plus rapides et plus efficaces. Docker repose sur cette technologie, et nécessite un noyau Linux en version 3.10 ou plus.

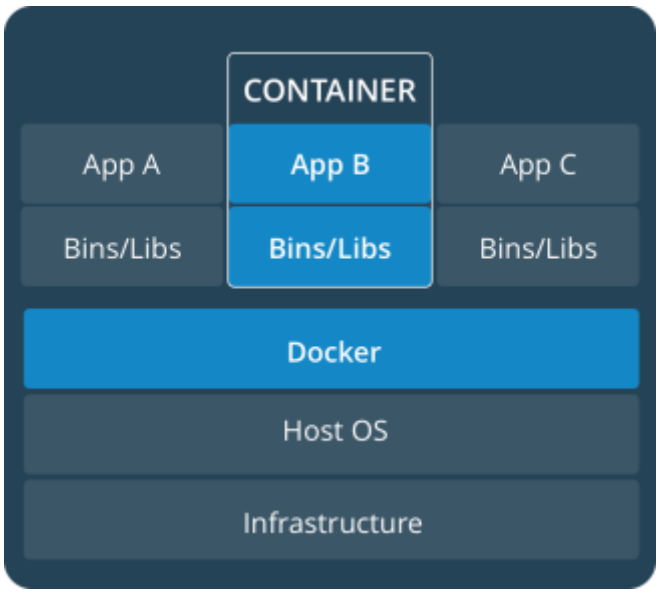
2-Docker isole un environnement comme une VM ?

Docker n'isole pas les applications comme une machine virtuelle (VM). Contrairement à une VM qui simule un serveur complet avec son propre système d'exploitation, Docker utilise directement les ressources de l'hôte (CPU, RAM, réseau, etc.) pour exécuter des conteneurs. Ces derniers offrent un environnement isolé, mais sans émuler un système d'exploitation complet. Docker est donc plus léger qu'une VM, et chaque conteneur est souvent dédié à une fonctionnalité spécifique (ex : base de données, service web).

Comme on peut le voir sur ce schéma, dans le cadre d'une machine virtuelle classique, chaque VM possède son propre système d'exploitation (Guest OS) contenant lui-même des bibliothèques (bin/lib) qui sont répliquées pour chacune des machines virtuelles.
De fait, les machines virtuelles sont lourdes ; ajoutons à cela le fait qu'elles doivent simuler tous les composants physiques d'une machine. Toute cette architecture de virtualisation consomme beaucoup de ressources pour fonctionner.



Ce schéma nous donne une bonne idée du principal avantage de Docker. Il n'est pas nécessaire de disposer d'un système d'exploitation complet chaque fois que nous devons créer un nouveau conteneur, ce qui réduit la taille globale des conteneurs.
Docker utilise le noyau Linux du système d'exploitation hôte (car presque toutes les versions de Linux utilisent les modèles de noyau standard) pour le système d'exploitation sur lequel il a été créé, telle que Debian, Ubuntu ou CentOS. Pour cette raison, vous pouvez utiliser presque n'importe quel système d'exploitation Linux en tant que système d'exploitation. Il est ainsi possible d'avoir un conteneur CentOS sur une machine Debian.



.

VII - Manipulation des conteneurs

1-Terminologie et concepts fondamentaux :

Deux concepts centraux :

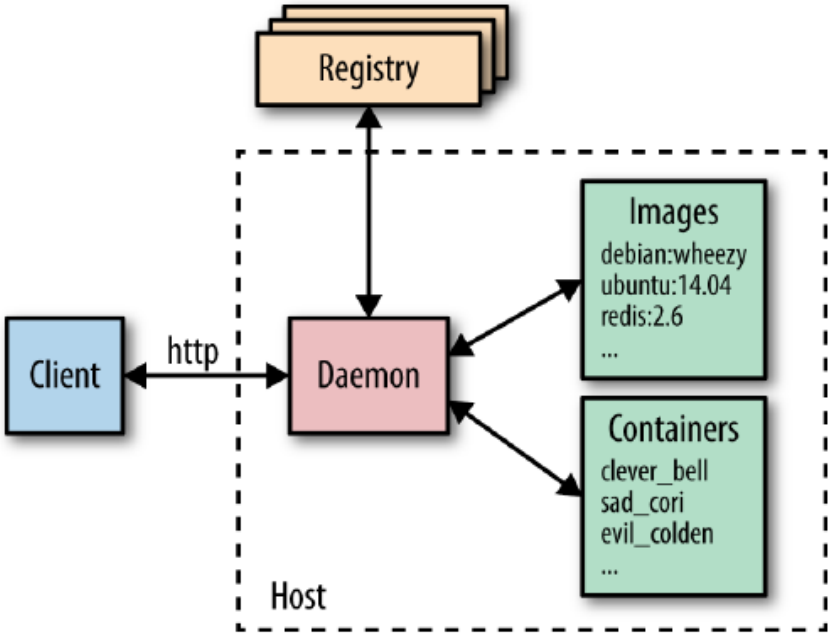
- **Une image : un modèle pour créer un conteneur**
- **Un conteneur : l'instance qui tourne sur la machine.**

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

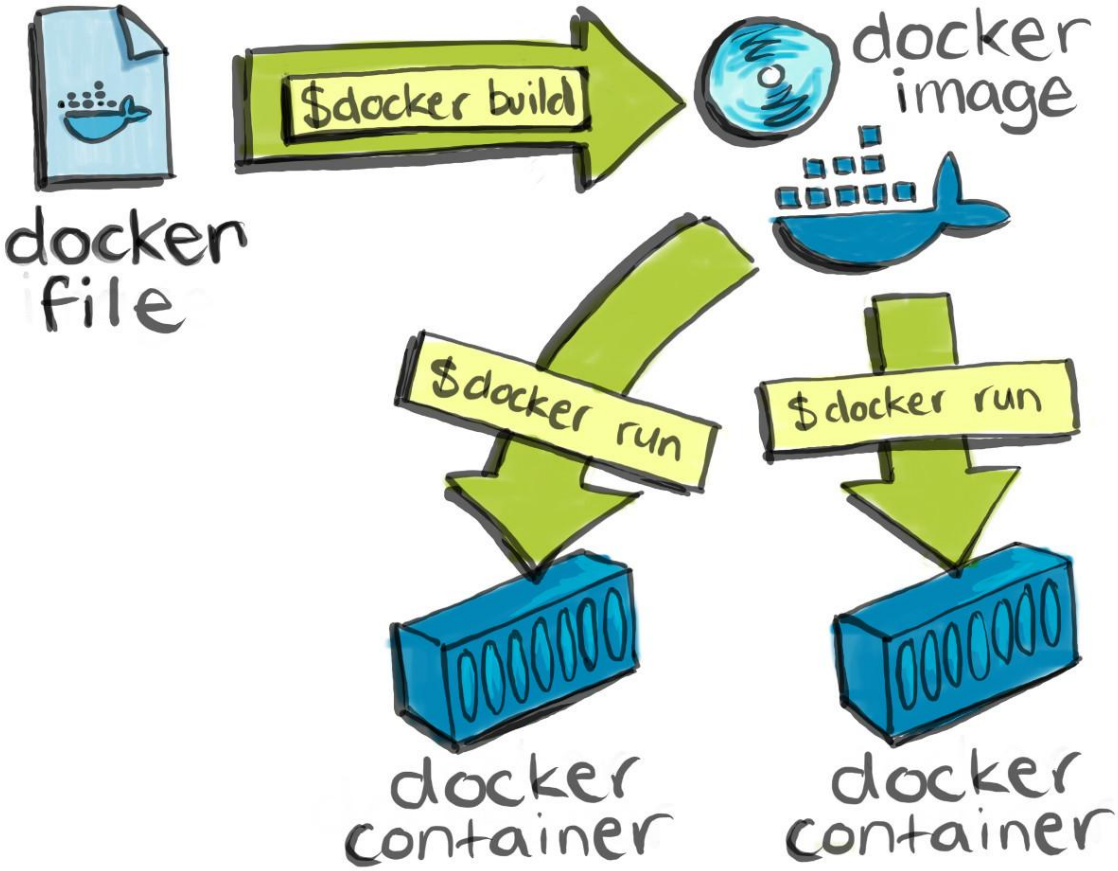


2-L'écosystème Docker :

- 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)**

VIII-Les images et conteneurs

1-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 :

```
salma-elodmi@LENOVO:~$ docker images
```

```
salma-elodmi@LENOVO:~$ docker images ls
```

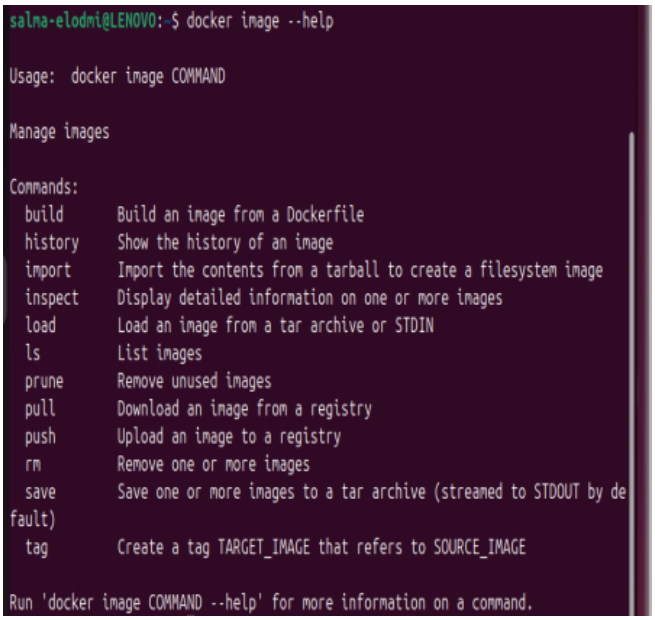
2-Les conteneurs :

- **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)**

3-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



IX-Pour vérifier l'état de Docker

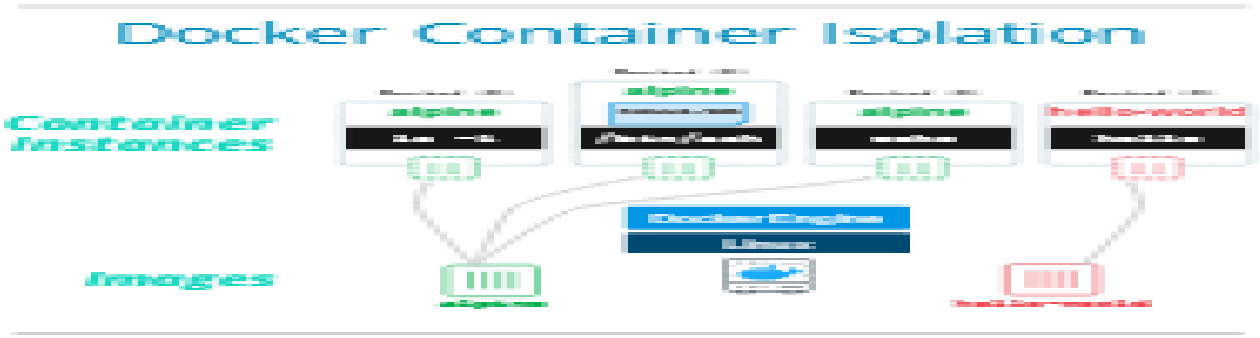
- **Les commandes de base pour connaître l'état de Docker sont :**

```
salma-elodmi@LENOVO:~$ docker info
    # affiche plein d'information sur l'engine avec lequel vous êtes en contact
```

```
salma-elodmi@LENOVO:~$ docker ps
# affiche les conteneurs en train de tourner
```

```
salma-elodmi@LENOVO:~$ docker ps -a
# affiche  également les conteneurs arrêtés
```

X-Créer et lancer un conteneur :



- **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>
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**.**

XI-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

```
salma-elodmi@LENOVO:~$ docker run debian

# s'arrête tout de suite
```

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

```
salma-elodmi@LENOVO:~$ docker run debian echo 'attendre 10s' && sleep 10

# s'arrête après 10s
```

XII-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
```

XIII-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

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

- 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**.

--- Installer Docker et jouer avec---

1-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 `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.**

2-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 mkdir /etc/bash_completion.d/
sudo curl -L https://raw.githubusercontent.com/docker/docker-
ce/master/components/cli/contrib/completion/bash/docker -o
/etc/bash_completion.d/docker.sh
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

3-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
```

- **Lancez simplement un conteneur Debian en mode *attached*. Que se passe-t-il ?**

Résultat :

- **Lancez un conteneur Debian (`docker run` puis les arguments nécessaires, cf. l'aide `--help`) en mode détaché avec la commande `echo "Debian container"`. 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.**

Résultat :

- **Affichez la liste des conteneurs en cours d'exécution**

Solution :

- **Affichez la liste des conteneurs en cours d'exécution et arrêtés.**

Solution :

- **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).**

Solution :

- **Tentez de lancer deux conteneurs avec le nom `debian_container`**

Solution :

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.**

4-Faire du ménage

- 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)

Solution :

- 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` ?

5-Décortiquer un conteneur

- **En utilisant la commande `docker export votre_conteneur -o conteneur.tar`, puis `tar -C conteneur_decompresse -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.**

XIV-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**.

XV-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.

>>>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 élements 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 fichier 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 `ENTRYPOINT`

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

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

`CMD` et `ENTRYPOINT`

- **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 `ENTRYPOINT`)**
- `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'`ENTRYPOINT` en combination 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.

XVI-Lancer la construction de l'image :

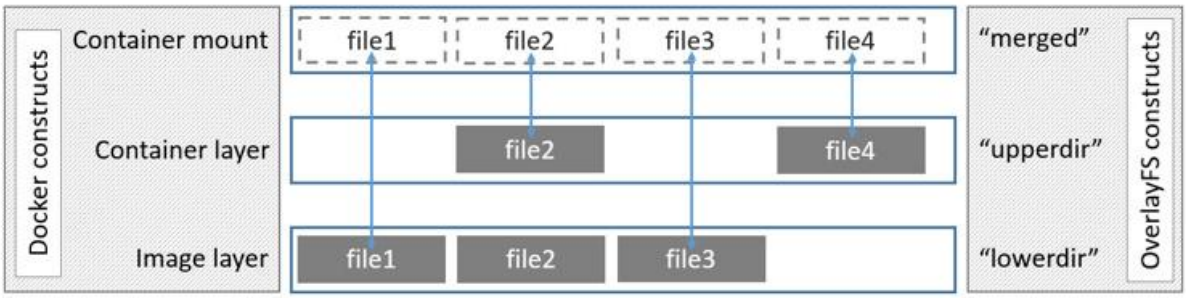
- **La commande pour lancer la construction d'une image est :**

```
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**

XVII-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 coeur 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.

1-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**.

2-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).**

XVIII-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`.

XIX-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.**

Transfert d'Images Docker entre Ubuntu et CentOS

1. Introduction

Ce rapport documente le processus de création, d'exportation, de transfert et d'importation d'une image Docker personnalisée entre deux systèmes Linux différents (Ubuntu et CentOS). L'objectif était de démontrer la portabilité des conteneurs Docker entre différentes distributions Linux.

2. Méthodologie

2.1. Environnement de test

- Machine source : **Ubuntu 22.04 LTS (IP: 192.168.1.100)**
- Machine destination : **CentOS 8 (IP: 192.168.1.192)**
- Docker version : **20.10.7 sur les deux systèmes**

2.2. Étapes réalisées

Sur Ubuntu (Machine source)

1. **Création d'un conteneur éphémère** :

   ```
   docker run fedora /bin/bash -c "echo 'copy from ubuntu to centos' > /tmp/copyfile"
   ```

2. **Commit en image personnalisée** :

   ```
   docker commit <CONTAINER_ID> copy
   ```

3. **Exportation de l'image** :

   ```
   docker save -o /tmp/copy.tar copy
   ```

4. **Transfert vers CentOS** :

   ```
   scp /tmp/copy.tar root@192.168.1.192:/tmp
   ```

Sur CentOS (Machine destination)

3. Résultats

1. **Importation de l'image** :

   ```
   docker load -i /tmp/copy.tar
   ```

2. **Vérification** :

   ```
   docker run -it copy /bin/bash -c "cat /tmp/copyfile"
   ```

- **L'image Docker a été créée avec succès sur Ubuntu (taille: 187MB)**
- **Le transfert via SCP a pris 12 secondes pour 187MB sur un réseau local**
- **L'importation sur CentOS a confirmé l'intégrité des données:**

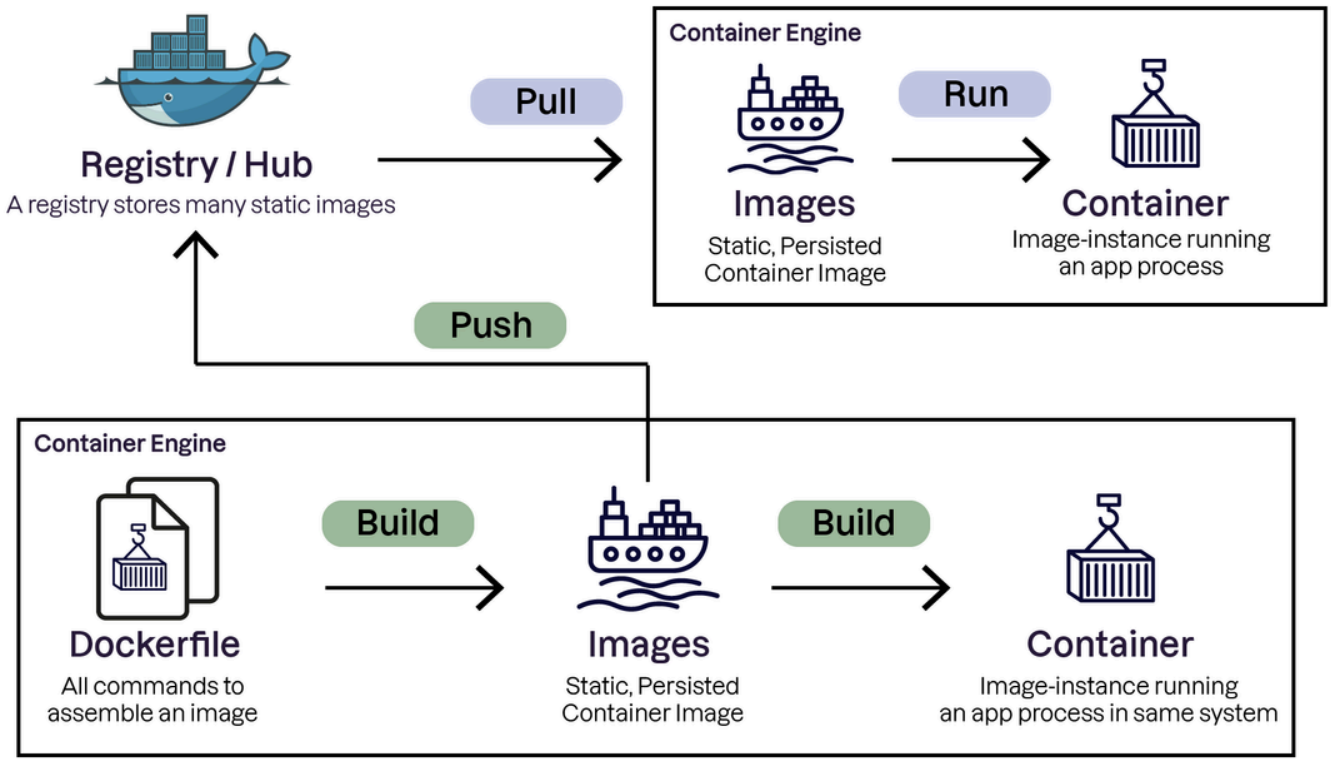
  ```
  copy from ubuntu to centos
  ```

Découvrez les registres

Les registres sont des dépôts où les images de conteneurs sont stockées et partagées. La spécification OCI définit un ensemble de standards pour la distribution d'images de conteneurs, garantissant ainsi l'interopérabilité entre différents outils et plateformes.

Concrètement, un registre OCI est défini par une API et permet à un client de réaliser les opérations suivantes :

- **Push** : Envoyer une image au registre.
- **Pull** : Télécharger une image du registre.
- **Tag** : Associer des métadonnées supplémentaires à une image pour la versionner.
- **Delete** : Supprimer des images obsolètes ou non nécessaires.



Publiez une image sur le registre Docker Hub

Docker Hub est le registre public par défaut utilisé par Docker. Il permet de stocker et de partager des images de conteneurs avec la communauté ou au sein de votre organisation. Ils offrent les services suivants :

- **Comptes personnels :** Vous pouvez créer un compte gratuit pour publier des images publiques ou souscrire à des plans payants pour des images privées.
- **Repositories :** Les images sont organisées par dépôts, et chaque dépôt peut contenir plusieurs tags d'images.

S'identifier sur le Docker Hub

Pour publier une image sur Docker Hub, vous devez tout d'abord vous authentifier avec votre client. Pour ce faire, tapez la commande :

`docker login`

Entrez vos identifiants Docker lorsque vous y êtes invité :

```
docker login
Log in with your Docker ID or email address to push and pull images from Docker Hub. If you don't have a Docker ID, head over to https://hub.docker.com/ to cr
eate one.
You can log in with your password or a Personal Access Token (PAT). Using a limited-scope PAT grants better security and is required for organizations using S
SO. Learn more at https://docs.docker.com/go/access-tokens/

Username: bornholm
Password:
WARNING! Your password will be stored unencrypted in /home/wpetit/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credential-stores

Login Succeeded
                                                                                                                                                    25s
```

Renommer l'image pour cibler votre dépôt Docker Hub

Supposons que vous ayez une image locale nommée libra:latest. Vous devez la taguer avec votre nom d'utilisateur Docker Hub et le nom du dépôt cible afin de la publier.

`docker tag libra:latest yourusername/libra:latest`

Pousser l'image sur le Docker Hub

Maintenant que l'image est renommée, vous pouvez la publier sur le Docker Hub avec la commande suivante :

`docker push yourusername/libra:latest`

```
docker push bornholm/libra:latest
The push refers to repository [docker.io/bornholm/libra]
ac28800ec8bb: Pushed
latest: digest: sha256:d37ada95d47ad12224c205a938129df7a3e52345828b4fa27b03a98825d1e2e7 size: 524
                                                                                                                                                    10s
```

L'image sera alors disponible à quiconque souhaitera la récupérer en exécutant sur sa machine :

`docker pull yourusername/libra:latest`

```
docker pull bornholm/libra:latest
latest: Pulling from bornholm/libra
Digest: sha256:d37ada95d47ad12224c205a938129df7a3e52345828b4fa27b03a98825d1e2e7
Status: Downloaded newer image for bornholm/libra:latest
docker.io/bornholm/libra:latest
```

Play with Docker

Play with Docker est une plateforme en ligne gratuite qui permet d'expérimenter Docker sans installation locale. Elle offre un environnement temporaire, accessible via un simple navigateur, pour créer et gérer des conteneurs. PWD est particulièrement utile pour les débutants, les

formateurs et les développeurs souhaitant tester rapidement des commandes Docker ou simuler des environnements multi-nœuds grâce au support de Docker Swarm.

Les sessions sont limitées dans le temps (environ 4 heures), mais suffisantes pour des tests, des démonstrations ou de la formation. L'interface permet de lancer plusieurs instances (nœuds) et de travailler sur des scénarios distribués.

Avantages :

- Aucun besoin d'installation
- Environnement prêt à l'emploi
- Idéal pour l'apprentissage ou la démonstration