



CESAB
CENTRE DE SYNTHÈSE ET D'ANALYSE
SUR LA BIODIVERSITÉ



Introduction à Docker

the hardest part 😕



Nicolas CASAJUS

{ Data scientist FRB-CESAB }

Mercredi 4 novembre 2020

C'est parti !



Il était une fois...

... un étudiant qui a passé plusieurs mois à analyser ses données. Comme il a tout compris à la vie, il travaille sous GNU/Linux. Son code est complexe et repose sur de nombreux logiciels et librairies système. Arrive le temps de partager son code avec ses collaborateurs. Mais, un de ses collaborateurs le contacte en lui disant que son code ne fonctionne pas. Ce dernier est sous Windows (*no comment*). Après plusieurs tentatives, il apparaît que cela n'a rien à voir avec le code : c'est un problème d'environnement de travail.

Que faire ?



Option 1

👉 Ecrire des tutoriels pour chaque OS (et version d'OS) pour installer toutes les dépendances système requises pour que le code fonctionne à peu près n'importe où

Mouais, bof...

J'ai autre chose à faire. En plus, j'ai pas accès à tous les OS...

Et quid des futurs OS ?

Option 2 - Virtualisation

- 👉 Fixer le type d'OS (et sa version) et utiliser une machine virtuelle dans lequel il s'exécutera



C'est une solution qui marche et qui est encore utilisée.

Le collaborateur devra alors :

1. Installer un logiciel de virtualisation (par ex. Oracle Virtual Box)
2. Télécharger l'ISO de l'OS (par ex. Ubuntu Focal Fossa 20.04 LTS)
3. Installer l'OS dans la machine virtuelle
4. Configurer la machine virtuelle (création de volumes de persistence)
5. Télécharger et configurer toutes les dépendances
6. Lancer le projet dans l'OS virtualisé

N.B. Certaines étapes peuvent être automatisées (par ex. avec **Vagrant**)

Option 2 - Virtualisation

👉 Problèmes :

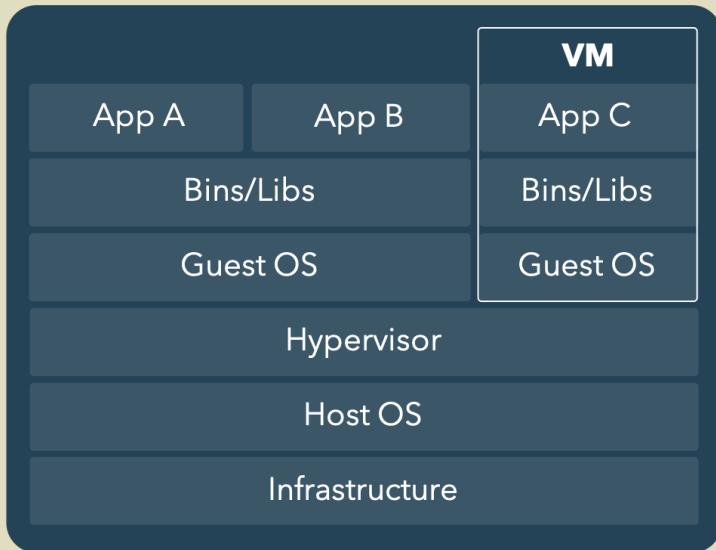
1. c'est compliqué à mettre en place,
2. c'est lourd (plusieurs Go juste pour l'OS et les dépendances),
3. c'est gourmand en ressources (RAM, CPU).

Option 3 - Conteneurisation

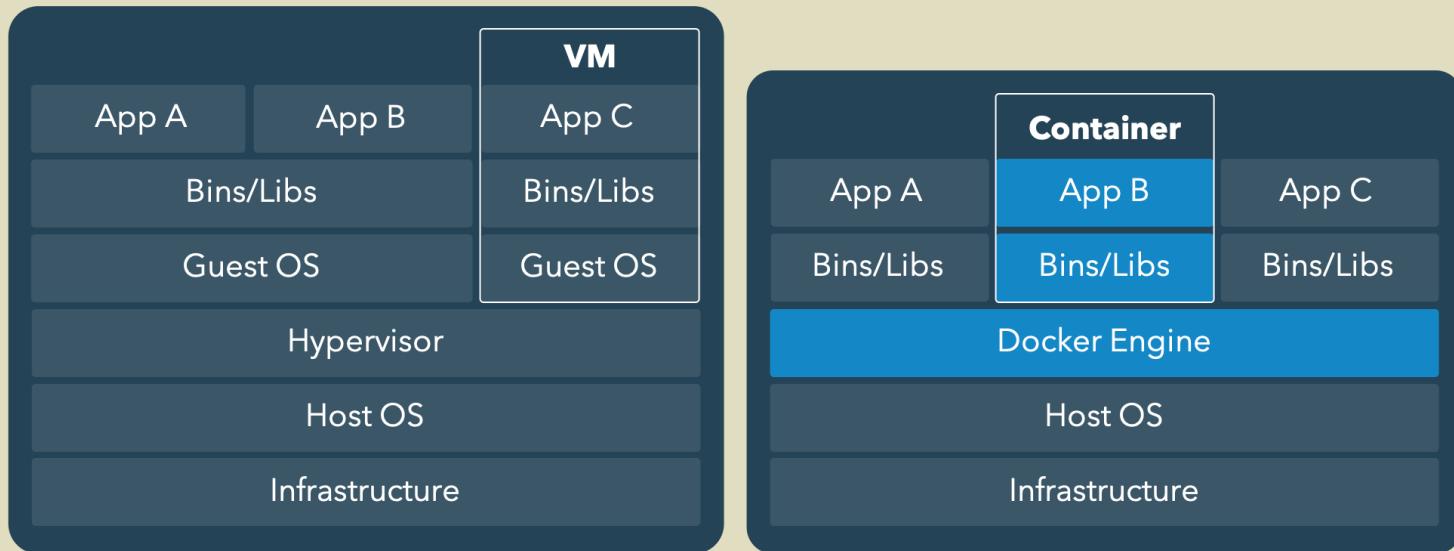
👉 **Solution :** s'orienter vers des solutions de **conteneurisation** telles que **Docker** 🚤 🚤 🚤



Virtualisation



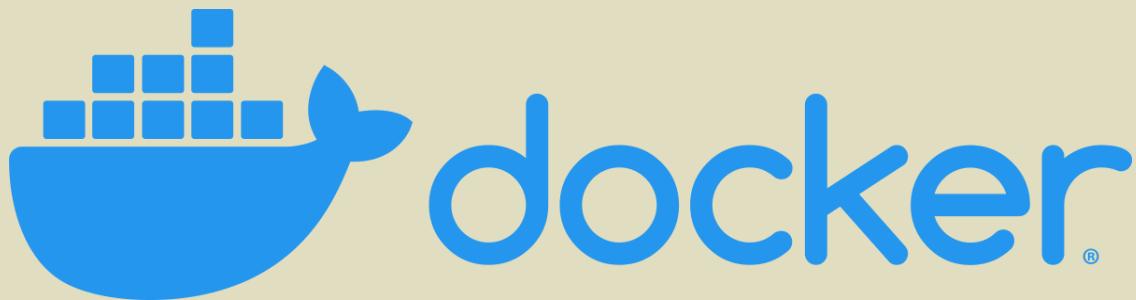
Virtualisation vs. Conteneurisation



👍 Avantages de la conteneurisation :

- Extrêmement léger (exploitation du noyau Linux)
- Utilisation réduite de ressources (RAM, CPU)
- Partage des ressources entre les instances
- Déploiement simple et instantané

Docker en bref



👉 Plateforme open source permettant de :

- empaqueter une application avec toutes ses dépendances système
- partager facilement un environnement de travail complet
- déployer rapidement une application en production

👉 Un conteneur est *isolé* du système hôte : Docker permet donc de :

- tester des choses sans crainte d'endommager son système hôte
- garder son système hôte propre, en installant tout sur Docker
- utiliser différentes versions d'une librairie, d'un serveur, etc.

⚠ Idéal pour créer un environnement de développement (basé sur Linux)

La notion d'Image

👉 Ca a l'air super, mais concrètement, comment je crée un conteneur ?

Tout conteneur est basé sur une image que l'on crée soi-même ou qu'on récupère depuis un site d'archivage (comme le CRAN pour R)

👉 Une **image** est une sorte de conteneur figé : c'est un template **fixe** (une recette de cuisine) à partir duquel on créera un ou des conteneurs. Une image est **immuable**.

👉 Un **conteneur** est donc une instance (exécution) d'une image qui pourra être utilisée/modifiée une fois créée

La notion d'Image

Prenons un exemple tiré par les cheveux.

Quand je lance RStudio (équivalent à une image  dans notre exemple), j'ouvre une instance RStudio (équivalent, dans notre exemple, à un conteneur). Si je clique une seconde fois sur l'exécutable de RStudio, j'ouvre une seconde instance (un second conteneur) indépendante de la première.

Tout ce que je fais dans la première instance n'est pas répercuté dans la seconde. Et vice versa.

Plus important encore, ces modifications ne se répercutent pas dans l'exécutable de RStudio.

C'est la même logique avec les images et conteneurs .

Docker Hub

👉 Où trouver des images 🚤 ? Sur un site d'archivage officiel **Docker Hub**. C'est une sorte de CRAN (publique) pour 🚤.

The screenshot shows the Docker Hub homepage. At the top, there's a blue header with the Docker Hub logo, a search bar containing "Search for great content (e.g., mysql)", and navigation links for Explore, Repositories, Organizations, Get Help, and a user account. Below the header, there are three tabs: Docker, Containers (which is selected), and Plugins. On the left, there are several filter categories with checkboxes: Docker Certified, Images, Categories, and Base Images. The "Containers" section displays 1 - 25 of 4 182 986 available images. The first card is for the "couchbase" image, which is marked as an official image with over 10M downloads and 639 stars. It's described as a NoSQL document database with a distributed architecture. The second card is for "postgres", also an official image with over 10M downloads and 8.5K stars, described as a PostgreSQL object-relational database system providing reliability and data integrity. The third card is for "traefik", an official image with over 10M downloads and 1.7K stars, described as a Cloud Native Edge Router.

Filters

1 - 25 of 4 182 986 available images.

Most Popular

Docker Certified

Docker Certified

Images

Verified Publisher Docker Certified And Verified Publisher Content

Official Images Official Images Published By Docker

Categories

Analytics

Application Frameworks

Application Infrastructure

Application Services

Base Images

Databases

DevOps Tools

Featured Images

Messaging Services

Monitoring

couchbase

Updated 16 hours ago

Couchbase Server is a NoSQL document database with a distributed architecture.

Container Linux x86-64 Storage Application Frameworks

OFFICIAL IMAGE 10M+ 639 Downloads Stars

postgres

Updated 16 hours ago

The PostgreSQL object-relational database system provides reliability and data integrity.

Container Linux ARM 64 PowerPC 64 LE x86-64 mips64le ARM IBM Z 386 Databases

OFFICIAL IMAGE 10M+ 8.5K Downloads Stars

traefik

Updated 16 hours ago

Traefik, The Cloud Native Edge Router

Container Windows Linux x86-64 ARM 64 ARM Application Infrastructure

OFFICIAL IMAGE 10M+ 1.7K Downloads Stars

Installation de Docker

Installation

- 👉 Docker s'installe sur GNU/Linux, et depuis peu, sur macOS et Windows 10 (installation différente selon l'édition, e.g. Home vs. Pro). Rendez-vous sur cette **page** et suivez les instructions.
- 👉 Une autre façon d'accéder à Docker sans l'installer en local est d'utiliser la plateforme **Play with Docker** (identifiant Docker requis)



C'est une plateforme qui donne accès à un serveur Alpine Linux sur lequel Docker et d'autres utilitaires (git, serveur ssh) sont préinstallés. La session dure 4h et on peut démarrer plusieurs instances.

👉 Durant l'exercice, on verra comment récupérer les données depuis ce serveur (ssh vs. git).

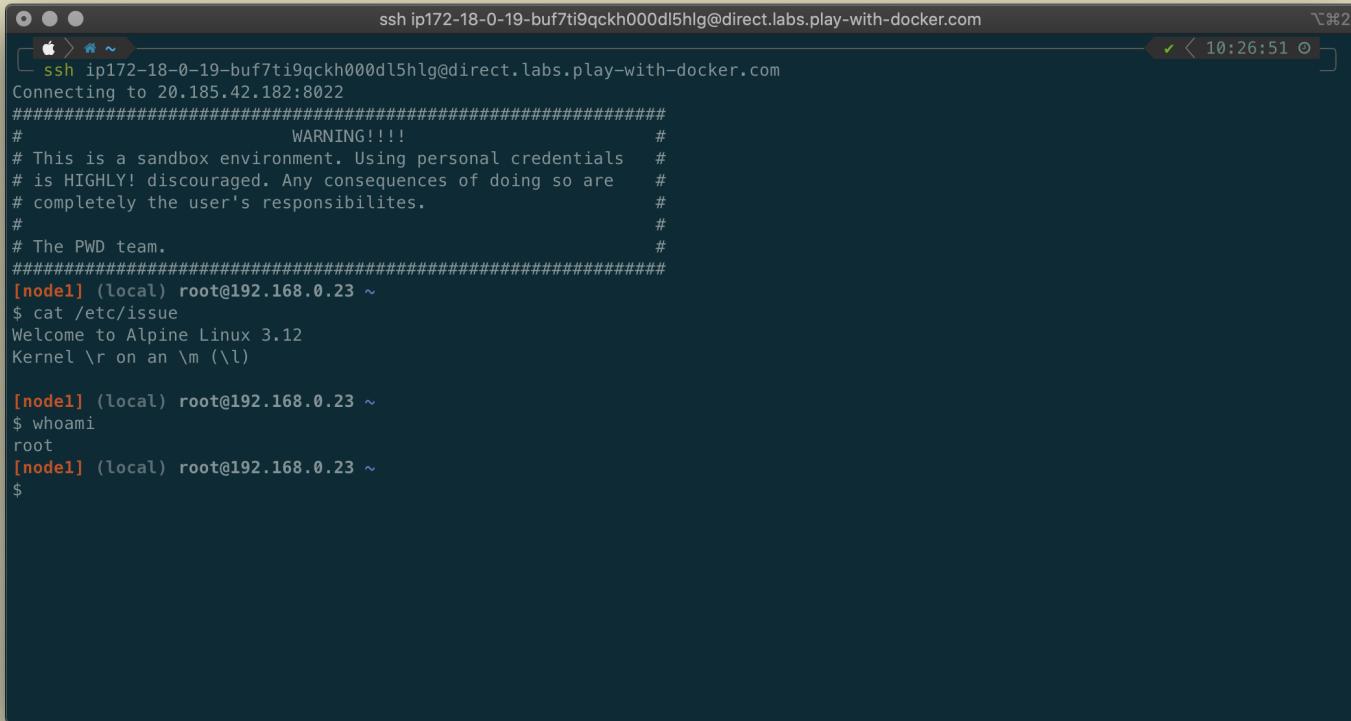
Play with Docker - Navigateur

The screenshot shows the 'Navigateur' interface for a Docker session. At the top, there's a blue header bar with the time '03:59:08'. Below it is an orange button labeled 'CLOSE SESSION'. The main area has a white background. At the top left, it says 'buf7ti9q_buf7tjpqckh000dl5hmg'. Below that, there are two buttons: 'IP' (set to 192.168.0.23) and 'OPEN PORT'. To the right of the IP button are 'Memory' (0.94% / 3.906GiB) and 'CPU' (2.50%). Further down, there's an 'SSH' field containing the command 'ssh ip172-18-0-19-buf7ti9qckh000dl5hlg@direct.labs.play-with-d...'. Below this is a 'DELETE' button and an 'EDITOR' button. On the left side, there's a sidebar with 'Instances' and a '+ ADD NEW INSTANCE' button. Under 'Instances', it shows '192.168.0.23' and 'node1'. The bottom half of the screen is a terminal window displaying a root shell on 'node1'. The terminal output includes a warning about using personal credentials, the Alpine Linux welcome message, and a series of commands like 'cat /etc/issue', 'docker --version', 'docker-compose --version', and 'git --version'.

```
#####
#          WARNING!!!!
# This is a sandbox environment. Using personal credentials #
# is HIGHLY! discouraged. Any consequences of doing so are #
# completely the user's responsibilites.                   #
#
# The PWD team.                                         #
#####
[node1] (local) root@192.168.0.23 ~
$ cat /etc/issue
Welcome to Alpine Linux 3.12
Kernel \r on an \m (\l)

[node1] (local) root@192.168.0.23 ~
$ docker --version
Docker version 19.03.11, build 42e35e61f3
[node1] (local) root@192.168.0.23 ~
$ docker-compose --version
docker-compose version 1.26.0, build unknown
[node1] (local) root@192.168.0.23 ~
$ git --version
git version 2.26.2
[node1] (local) root@192.168.0.23 ~
$ 
```

Play with Docker - Accès SSH



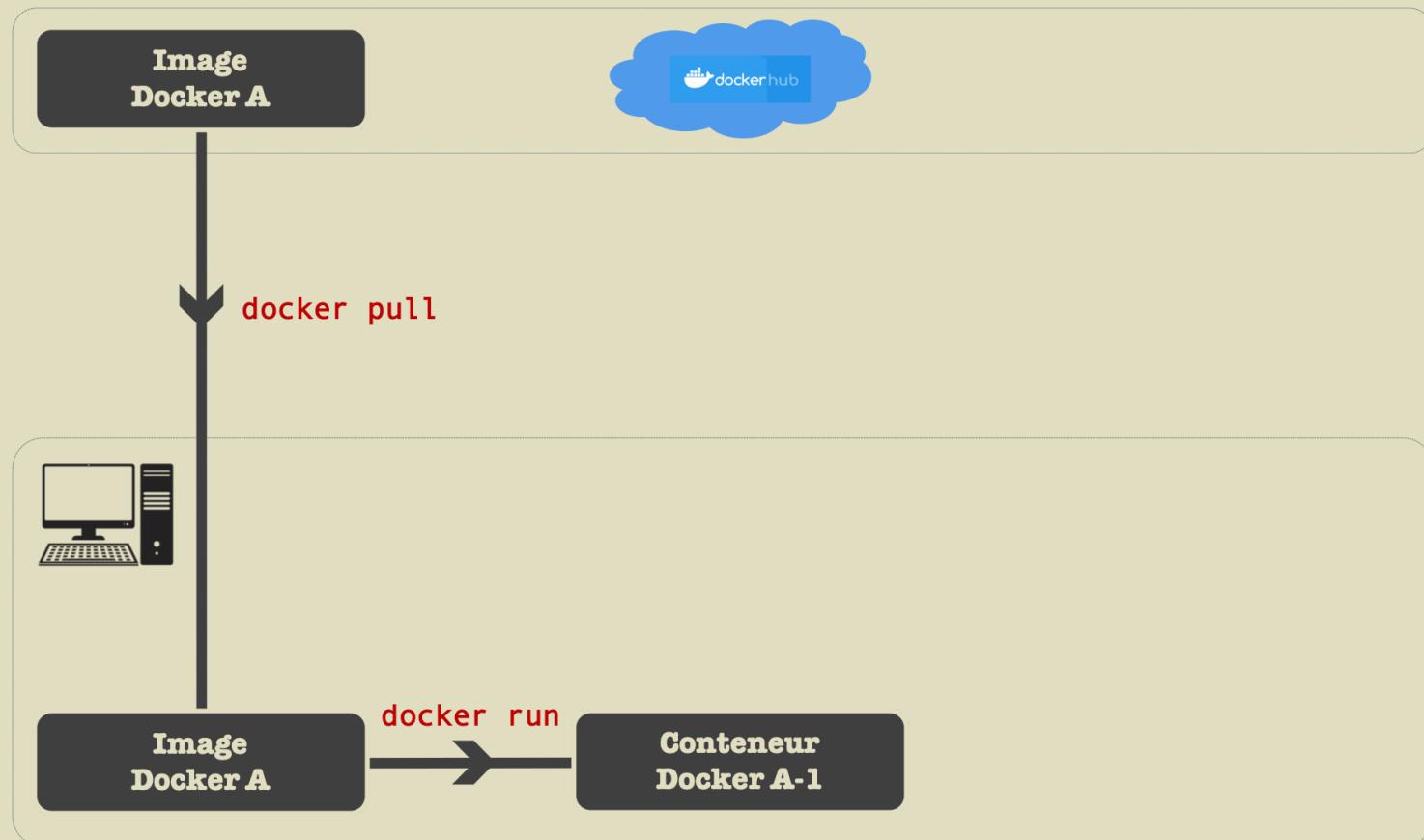
A screenshot of a terminal window titled "ssh ip172-18-0-19-buf7ti9qckh000dl5hlg@direct.labs.play-with-docker.com". The window shows an SSH session to a container. The session starts with a warning about using personal credentials in a sandbox environment. It then connects to a local host at 192.168.0.23 and runs the command "cat /etc/issue", which outputs the Alpine Linux 3.12 welcome message. Finally, it runs "whoami" and shows the user is root.

```
ssh ip172-18-0-19-buf7ti9qckh000dl5hlg@direct.labs.play-with-docker.com
ssh ip172-18-0-19-buf7ti9qckh000dl5hlg@direct.labs.play-with-docker.com
Connecting to 20.185.42.182:8022
#####
#           WARNING!!!!
# This is a sandbox environment. Using personal credentials
# is HIGHLY! discouraged. Any consequences of doing so are
# completely the user's responsibilites.
#
# The PWD team.
#####
[node1] (local) root@192.168.0.23 ~
$ cat /etc/issue
Welcome to Alpine Linux 3.12
Kernel \r on an \m (\l)

[node1] (local) root@192.168.0.23 ~
$ whoami
root
[node1] (local) root@192.168.0.23 ~
$
```

Création d'un conteneur

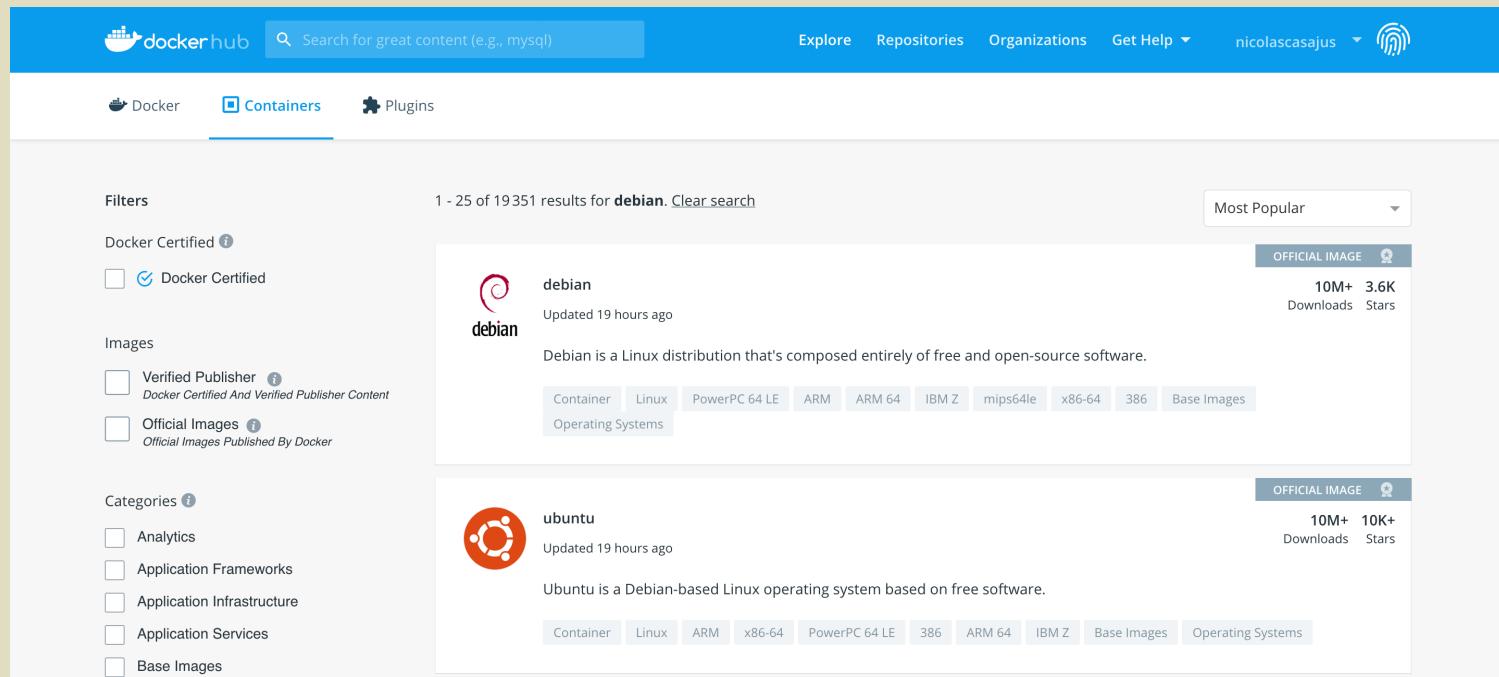
Création d'un conteneur



Mon premier conteneur

Je ne connais pas du tout GNU/Linux, mais j'aimerais bien l'essayer !

👉 Qu'à cela ne tienne : nous allons télécharger une image 🚀 de Debian. Allons faire un tour sur **Docker Hub**.



The screenshot shows the Docker Hub interface with a search query for "debian". The results page displays two official images: "debian" and "ubuntu".

Filters:

- Docker Certified
- Docker Certified
- Images

 - Verified Publisher Docker Certified And Verified Publisher Content
 - Official Images Official Images Published By Docker

- Categories

 - Analytics
 - Application Frameworks
 - Application Infrastructure
 - Application Services
 - Base Images

Search Results:

1 - 25 of 19351 results for **debian**. [Clear search](#)

debian OFFICIAL IMAGE 
Updated 19 hours ago
10M+ Downloads 3.6K Stars
Debian is a Linux distribution that's composed entirely of free and open-source software.
Container Linux PowerPC 64 LE ARM ARM 64 IBM Z mips64le x86-64 386 Base Images Operating Systems

ubuntu OFFICIAL IMAGE 
Updated 19 hours ago
10M+ Downloads 10K+ Stars
Ubuntu is a Debian-based Linux operating system based on free software.
Container Linux ARM x86-64 PowerPC 64 LE 386 ARM 64 IBM Z Base Images Operating Systems

Mon premier conteneur

- 👉 On peut également chercher une image 🛠 depuis le terminal

```
docker search debian
```

| NAME | DESCRIPTION | STARS | OFFICIAL |
|--------|---|-------|----------|
| ubuntu | Ubuntu is a Debian-based Linux operating sys... | 11465 | [OK] |
| debian | Debian is a Linux distribution that's compos... | 3635 | [OK] |
| ... | ... | ... | ... |

- 👉 Pour télécharger l'image 🛠, on peut lancer la commande suivante :

```
docker pull debian
```

⚠ Cependant, en procédant ainsi nous téléchargerons la dernière version de Debian. Afin d'être reproductible, nous fixerons une version particulière (appelée **tag** sous 🛠).

- 👉 Téléchargeons l'image 🛠 de Debian 10 (appelée Buster).

```
docker pull debian:buster          # ou, docker pull debian:10
```

Mon premier conteneur

👉 Listons les images 🚤 présentes en local :

```
docker images # ou, docker image ls
```

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|------------|--------|--------------|-------------|-------|
| debian | buster | 1510e8501783 | 2 weeks ago | 114MB |

Deux choses importantes sont à noter :

1. Chaque image 🚤 dispose d'un identifiant unique (**IMAGE ID**)
2. La taille de l'image 🚤 !!!

👉 Pour supprimer une image :

```
docker rmi 1510e8501783
```

Mon premier conteneur

👉 Créons notre conteneur, c'est-à-dire une instance de notre image 🚤.

```
docker run -it debian:buster          # ou, docker run -it 1510e8501783
```

Notre prompt vient de changer :

```
root@ea8d63136eef:/#
```

Nous sommes dans notre conteneur 🚤 ! Le prompt nous dit que nous sommes connectés en tant qu'utilisateur **root** sur le conteneur dont l'identifiant est **ea8d63136eef** (différent de l'ID de l'image !)

Vous ne me croyez pas ?

```
cat /etc/issue
```

```
Debian GNU/Linux 10
```

Mon premier conteneur

Revenons sur la commande `docker run -it debian:buster`

1. L'instruction `docker run` exécute une instance de l'image  **debian** portant le tag **buster**. Il faut noter que si cette image est absente de notre machine, alors  essaiera de la télécharger depuis Docker Hub. Donc, cette commande fait également un `docker pull` si besoin est.
2. Le double drapeau `-it` indique que l'on souhaite interagir avec le conteneur. En général, les images  ne contenant qu'un OS (pas d'application) retournent toujours un shell (invite de commandes), mais pour y accéder, il faut demander l'accès (avec le drapeau `-it`).

Pour être certain d'interagir avec le shell, nous aurions pu indiquer la commande devant être exécutée au lancement du conteneur (ici, **bash**, le shell par défaut sous Debian)

```
docker run -it debian:buster bash
```

Mon premier conteneur

👉 Si jamais nous avions oublié le drapeau `-it`, ce n'est pas grave. Nous avons toujours la possibilité de rentrer dans le conteneur avec :

```
docker exec -it ea8d63136eef bash
```

⚠️ Lorsqu'on quitte le conteneur avec la commande `exit`, celui-ci est stoppé si on est rentré dans le conteneur avec la commande `docker run`. Mais il n'est pas stoppé si on a accédé au shell avec `docker exec bash`.

Par défaut, Docker donne un nom aléatoire à un nouveau conteneur (notre conteneur a été nommé `magical_lehmann`). Mais, on peut lui en attribuer un de notre choix avec le drapeau `--name` :

```
docker run -it --name "debbie" debian:buster
```

Gestion des conteneurs

Gestion des conteneurs

👉 Pour lister les conteneurs (processus) actifs :

```
docker ps
```

| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | PORTS | NAMES |
|--------------|---------------|---------|-----------|----------|-------|-----------------|
| ea8d63136eef | debian:buster | "bash" | 2 min ago | Up 2 min | | magical_lehmann |

👉 Pour arrêter un conteneur :

```
docker stop ea8d63136eef # ou, docker stop magical_lehmann
```

👉 Pour démarrer un conteneur :

```
docker start ea8d63136eef # ou, docker start magical_lehmann
```

👉 Pour supprimer un conteneur :

```
docker rm ea8d63136eef # après un docker stop  
docker rm -f ea8d63136eef # si le conteneur est actif
```

Gestion des conteneurs

👉 Pour lister les conteneurs (processus) actifs et arrêtés :

```
docker ps -a
```

| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS | NAMES |
|--------------|---------------|---------|-------------|-------------------|-----------------|
| ea8d63136eef | debian:buster | "bash" | 2 min ago | Up 2 min | magical_lehmann |
| 178b8649b677 | debian:buster | "bash" | 9 hours ago | Exited (0) 2s ago | debbie |

👉 Enfin, pour connaître les ressources consommées par les conteneurs actifs :

```
docker stats
```

| CONTAINER ID | NAME | CPU % | MEM USAGE / LIMIT | MEM % |
|--------------|-----------------|-------|-------------------|-------|
| ea8d63136eef | magical_lehmann | 0.00% | 788KiB / 1.944GiB | 0.04% |

⚠️ Après un reboot de la machine, les conteneurs actifs et stoppés existent toujours. Mais, ceux qui ont été supprimés avec `docker rm`, eux, n'existent plus.

Persistance

Persistance des conteneurs

👉 Mais si un conteneur est isolé, comment je récupère mes données qui sont à l'intérieur ? Et, comment je donne accès aux données stockées sur ma machine à mon conteneur ?

En effet, un conteneur n'est pas persistant : tout ce qui est ajouté à l'intérieur (fichiers, logiciels, etc.) n'est pas accessible depuis l'extérieur. Et lorsque celui-ci est détruit, tout est perdu ! Ce qui est un avantage pour bidouiller le contenu, mais un inconvénient lorsqu'on veut travailler sérieusement.

👉 Pour les données, on créera des **volumes**

👉 Pour les logiciels, on créera une nouvelle **image** 🚤

Volumes de persistance

👉 Un volume* est un dossier. L'idée est de créer un pont entre un dossier du conteneur et un dossier de notre machine. On parle de **volumes mapping**

Ainsi, un fichier/dossier créé dans le volume de la machine sera accessible par le conteneur via le volume lié. Et vice-versa.

Le volume local (celui sur la machine) sera persistant même après la suppression du conteneur (et de son volume persistant).

👉 C'est cela qu'on appelle la **persistance**

* Ce n'est pas tout à fait vrai, et un volume peut avoir un autre sens avec Docker, mais faisons simple...

Volumes de persistance

Soit le dossier `projet/` présent dans mon dossier personnel (noté `~` sous Unix) de ma machine (macOS). Celui-ci contient le dossier `data/`, un fichier `.csv` à l'intérieur et le fichier `README.md` :

```
projet
└── README.md
└── data
    └── data-1.csv
```

Je souhaite mapper ce volume à un dossier de mon `nouveau` conteneur, par ex. le dossier personnel de l'utilisateur `root` : `/root/`

👉 Pour cela, lorsque je créerai un nouveau conteneur basé sur l'image `debian:buster` avec la commande `docker run`, je rajouterais le drapeau `--volume` (ou `-v`) de la manière suivante :

```
-v chemin_absolu_vers_dossier_local:chemin_absolu_vers_dossier_conteneur
```

⚠️ Les volumes mentionnés doivent exister tant sur la machine locale et dans le conteneur et il faut mentionner le chemin absolu vers ces dossiers.

Volumes de persistance

👉 Ce qui nous donne :

```
docker run -it --name "debbie" -v ~/projet:/root debian:buster
```

Maintenant, si j'affiche le contenu du dossier `/root/` de mon conteneur

```
ls /root
```

`README.md` `data`

```
ls /root/data/
```

`data-1.csv`

Créons un second fichier `README-2.md` dans mon conteneur

```
echo "Cree dans le conteneur" > /root/README-2.md
```

Volumes de persistance

Vérifions qu'il a bien été créé depuis le conteneur :

```
ls /root
```

```
README-2.md README.md data
```

Affichons son contenu

```
cat /root/README-2.md
```

```
Cree dans le conteneur
```

👉 Qu'en est-il sur ma machine ?

```
tree ~/projet
```

```
projet
├── README-2.md
├── README.md
└── data
    └── data-1.csv
```

Volumes de persistance

👉 Et si je supprime le conteneur, que deviendra mon dossier sur ma machine ?

```
docker stop debbie  
docker rm debbie
```

```
tree ~/projet
```

```
projet  
├── README-2.md  
├── README.md  
└── data  
    └── data-1.csv
```

👉 Les données ont été persistées !!!

Création d'une image

Ajout d'outils

👉 Que se passe-t-il si mon conteneur ne contient pas tous les outils (bibliothèques, logiciels, etc.) dont j'ai besoin ?

👉 **Option 1** : la méthode **sale**

L'idée est de rentrer dans un conteneur, d'installer les outils manquants (**apt-get install**) et d'enregistrer le nouvel état du conteneur vers une nouvelle image (avec **docker commit**). A titre informatif (uniquement) :

```
docker commit id_image nom_nouvelle_image
```

C'est **très très sale** : les outils sont installés à la volée (depuis le conteneur) et on ne garde aucune trace des changements apportés. Niveau sécurité, c'est pas top... Et en plus, on n'est pas à l'abri de faire des mauvaises manip' et de compromettre la future image 🚤

👉 **Option 2** : le **Dockerfile**

Création d'image - Dockerfile

👉 Le **Dockerfile** est la recette qui va créer une nouvelle image 🚤

C'est un simple fichier texte (**sans extension**) qui fournit une suite d'instructions amenant à la construction de la future image. Chaque instruction va créer une couche dans l'image (système de Lego) : plus il y aura de couches, plus l'image sera lourde.

De plus, il est rare de construire une image *from scratch* : on se base très souvent sur une (seule) image existante.

👉 Ainsi, le processus de création d'une image est **incrémental** et peut être comparé à un **git pull**

Lors de la construction de l'image, si l'image de référence est présente sur la machine, pas besoin de la télécharger (à l'instar, lors d'un **git pull**, on ne télécharge pas l'ensemble de l'historique, juste les dernières modifications).

Création d'image - Dockerfile

👉 **Objectif**: nous allons améliorer l'image `debian:buster` en lui ajoutant quelques utilitaires manquants :

- `htop` : outil de monitoring du système
- `nano` : éditeur de texte simple en ligne de commandes
- `tree` : outil d'arborescence
- `wget` : programme de téléchargement de fichiers

Sur notre machine locale, nous allons créer un dossier, par ex. `mydebbie/` dans lequel nous allons créer un fichier qu'on va nommer `Dockerfile` et dans lequel nous écrire ces lignes.

```
FROM debian:buster
MAINTAINER Nicolas Casajus <nicolas.casajus@fondationbiodiversite.fr>
```

- La clause `FROM` permet de définir l'image de base (et sa version) sur laquelle on va bâtir la nouvelle image.
- La clause `MAINTAINER` indique le créateur de l'image ainsi que le moyen de le contacter.

Création d'image - Dockerfile

👉 Tel quel, ce **Dockerfile** construira une image identique à l'image **debian:buster**

Rajoutons les instructions qui permettront d'installer les utilitaires requis

```
FROM debian:buster

MAINTAINER Nicolas Casajus <nicolas.casajus@fondationbiodiversite.fr>

RUN apt-get update -yq
RUN apt-get install -yq --no-install-recommends htop nano tree wget
```

- La clause **RUN** permet de passer des commandes Linux classiques. Elle ajoute une couche supplémentaire à notre image, ce qui l'alourdi. Il est donc recommandé de limiter ces clauses et de les fusionner.

```
FROM debian:buster

MAINTAINER Nicolas Casajus <nicolas.casajus@fondationbiodiversite.fr>

RUN apt-get update -yq \
    && apt-get install -yq --no-install-recommends htop nano tree wget
```

Création d'image - Dockerfile

- 👉 On peut réduire encore la taille de la future image en supprimant le cache du gestionnaire **apt** qui contient les archives des utilitaires téléchargés.

```
FROM debian:buster

MAINTAINER Nicolas Casajus <nicolas.casajus@fondationbiodiversite.fr>

RUN apt-get update -yq \
    && apt-get install -yq --no-install-recommends htop nano tree wget \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/
```

Création d'image - Dockerfile

- 👉 Nous pouvons aussi créer un répertoire avec la commande Unix `mkdir`

```
FROM debian:buster

MAINTAINER Nicolas Casajus <nicolas.casajus@fondationbiodiversite.fr>

RUN apt-get update -yq \
    && apt-get install -yq --no-install-recommends htop nano tree wget \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/ \
    && mkdir -p /root/data
```

Création d'image - Dockerfile

👉 Améliorons encore notre recette en définissant le répertoire par défaut du conteneur, par ex. `/root/data` (celui dans lequel on se retrouvera lorsqu'on rentrera dans le conteneur) avec la clause `WORKDIR`

```
FROM debian:buster

MAINTAINER Nicolas Casajus <nicolas.casajus@fondationbiodiversite.fr>

RUN apt-get update -yq \
    && apt-get install -yq --no-install-recommends htop nano tree wget \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/ \
    && mkdir -p /root/data

WORKDIR /root/data
```

Création d'image - Dockerfile

👉 J'ai du écrire deux fois `/root/data` : en programmation, c'est jamais bon... Heureusement, Docker offre la possibilité de manipuler des variables d'environnement avec la clause `ENV`. Optimisons donc le code.

```
FROM debian:buster

MAINTAINER Nicolas Casajus <nicolas.casajus@fondationbiodiversite.fr>

ENV FOLDER="/root/data"

RUN apt-get update -yq \
    && apt-get install -yq --no-install-recommends htop nano tree wget \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/ \
    && mkdir -p "$FOLDER"

WORKDIR "$FOLDER"
```

Création d'image - Dockerfile

👉 Je voudrais copier un fichier que j'ai en local (et dans le même dossier dans l'exemple) dans l'image, afin qu'il soit présent dans mes futurs conteneurs. Pour cela, j'ai la clause **COPY**

```
FROM debian:buster

MAINTAINER Nicolas Casajus <nicolas.casajus@fondationbiodiversite.fr>

ENV FOLDER="/root/data"

RUN apt-get update -yq \
    && apt-get install -yq --no-install-recommends htop nano tree wget \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/ \
    && mkdir -p "$FOLDER"

COPY README.md "$FOLDER"

WORKDIR "$FOLDER"
```

Création d'image - Dockerfile

👉 Finalement, je peux aussi définir une commande par défaut qui sera exécutée lors du lancement d'un conteneur. Ici, ouvrons un shell **BASH**

```
FROM debian:buster

MAINTAINER Nicolas Casajus <nicolas.casajus@fondationbiodiversite.fr>

ENV FOLDER="/root/data"

RUN apt-get update -yq \
    && apt-get install -yq --no-install-recommends htop nano tree wget \
    && apt-get clean \
    && rm -rf /var/lib/apt/lists/ \
    && mkdir -p "$FOLDER"

COPY README.md "$FOLDER"

WORKDIR "$FOLDER"

CMD ["bash"]
```

La clause **CMD** s'écrit toujours sous la forme d'un array (une sorte de liste) : **CMD** `["executable", "param1", "param2", "..."]`

Le .dockerignore

👉 Tout comme **git** et  est capable d'ignorer certains fichiers/dossiers lors de la construction d'une image grâce à un **.dockerignore**

Cela est particulièrement important lorsque la clause **COPY** copie des dossiers entiers.

Etant sous macOS, je vais ignorer les fameux fichiers **.DS_Store** et pourquoi pas les fichiers et dossiers cachés de **git** (ça grossirait l'image pour rien)

```
echo ".DS_Store" > .dockerignore
echo ".git" >> .dockerignore
echo ".gitignore" >> .dockerignore
```

Build de l'image

- 👉 Nous sommes prêt pour construire (builder) notre image à partir du **Dockerfile**. Pour cela, je vais utiliser la commande **docker build** avec le drapeau **-t** pour donner un nom à mon image. Je dois aussi spécifier le répertoire contenant ma recette. Ici, c'est le dossier dans lequel je vais lancer la commande, donc je peux écrire .

```
docker build -t buster .
```

Après quelques minutes, l'image est construite sur ma machine et a rejoint ma collection.

```
docker images
```

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|------------|--------|--------------|----------------|-------|
| buster | latest | dd48c15a9389 | 46 minutes ago | 121MB |
| debian | buster | 1510e8501783 | 2 weeks ago | 114MB |

- 👉 Ma nouvelle image est légèrement plus grosse que celle d'origine : nous n'avons pas abusé des clauses **RUN**.

- 👉 Par défaut, la tag **latest** a été ajouté à la nouvelle image.

Utilisation de l'image

👉 Nous revoilà revenu au point de départ. Mais, avec une toute nouvelle image, faite maison qui plus est ! Créons-nous un conteneur pour l'fun

```
docker run -it --name "buster-1" buster
```

Et tapons quelques commandes :

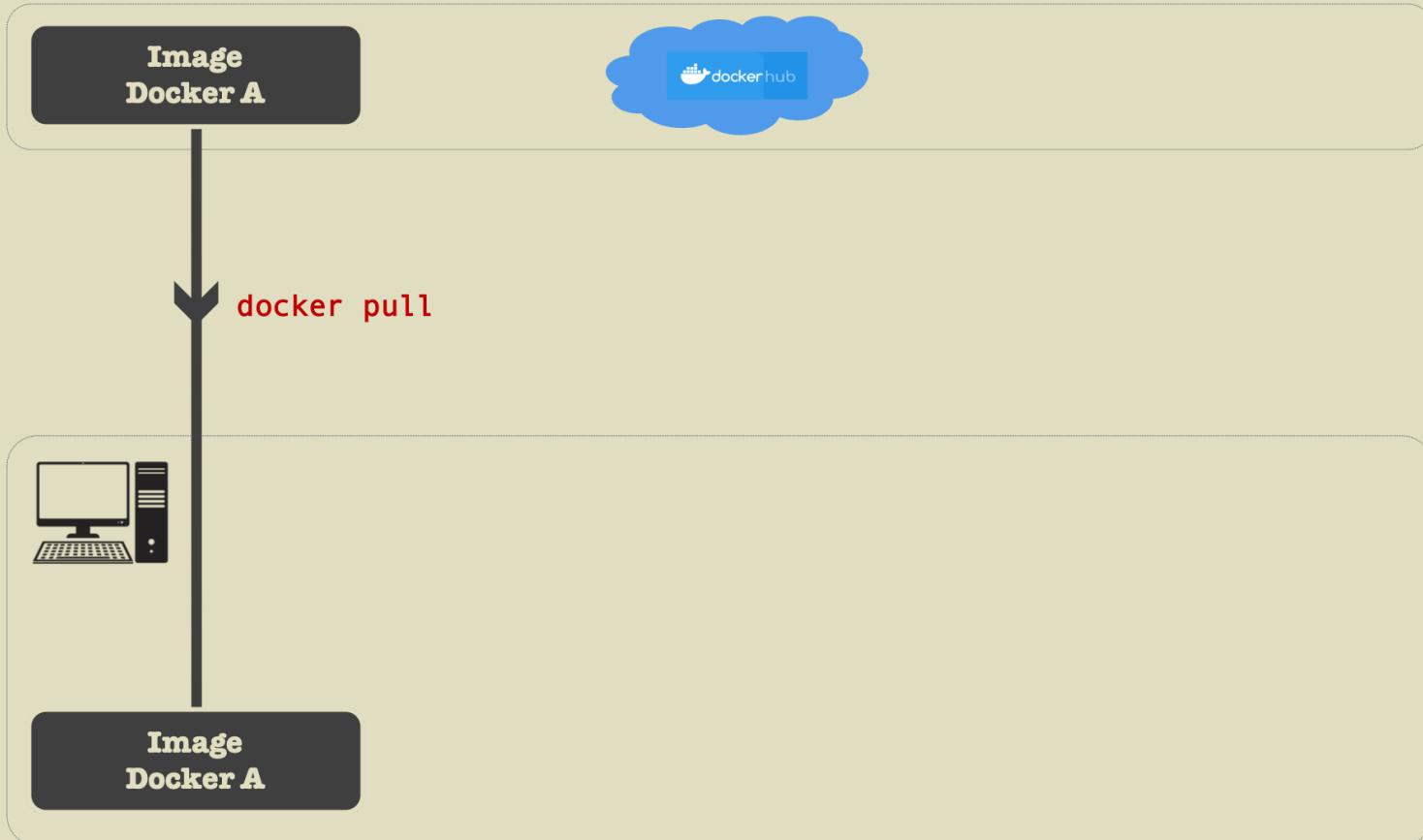
```
pwd # Répertoire de travail
```

```
/root/data
```

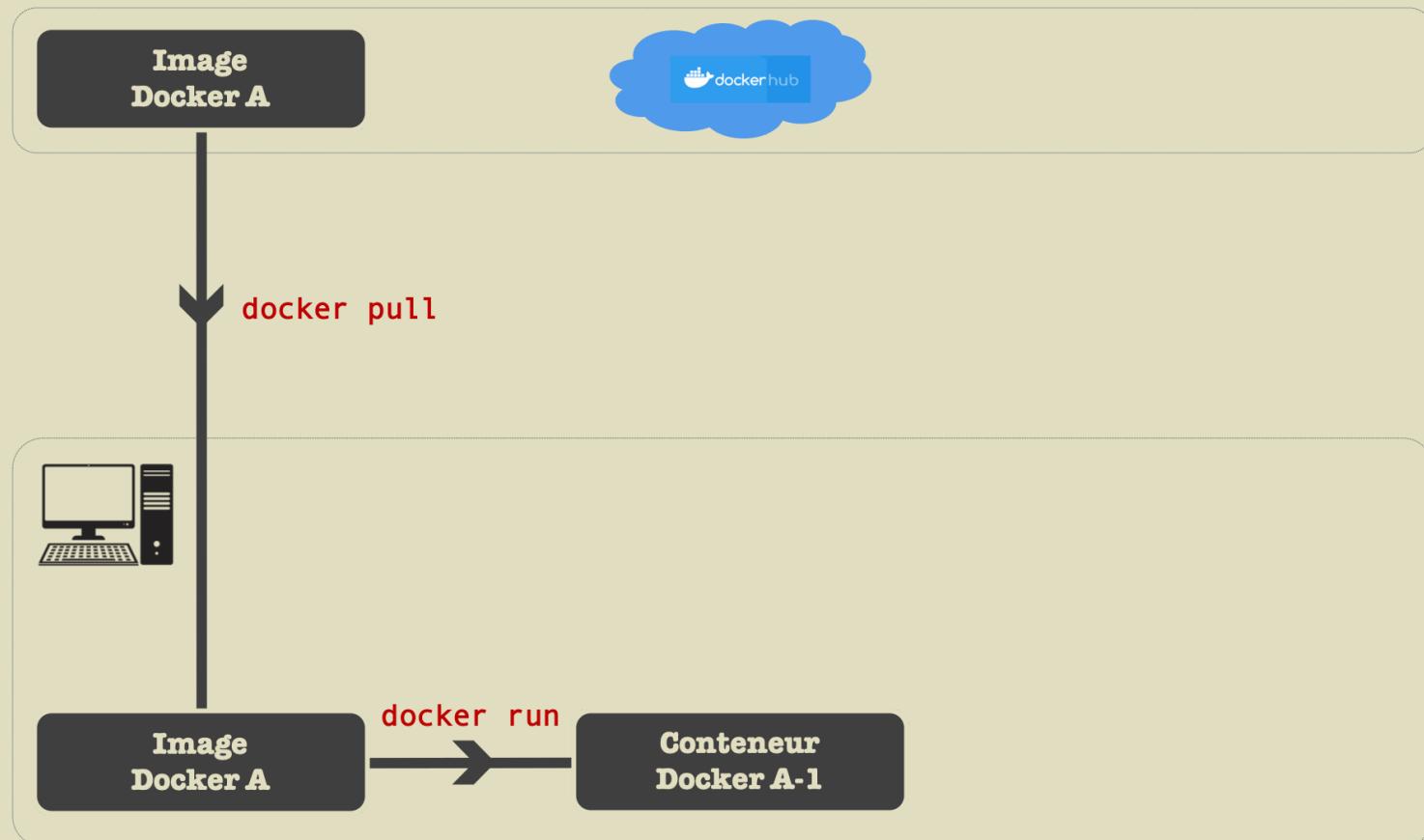
```
tree ../ # Utilisation du nouvel utilitaire
```

```
.
└── data
    └── README.md
```

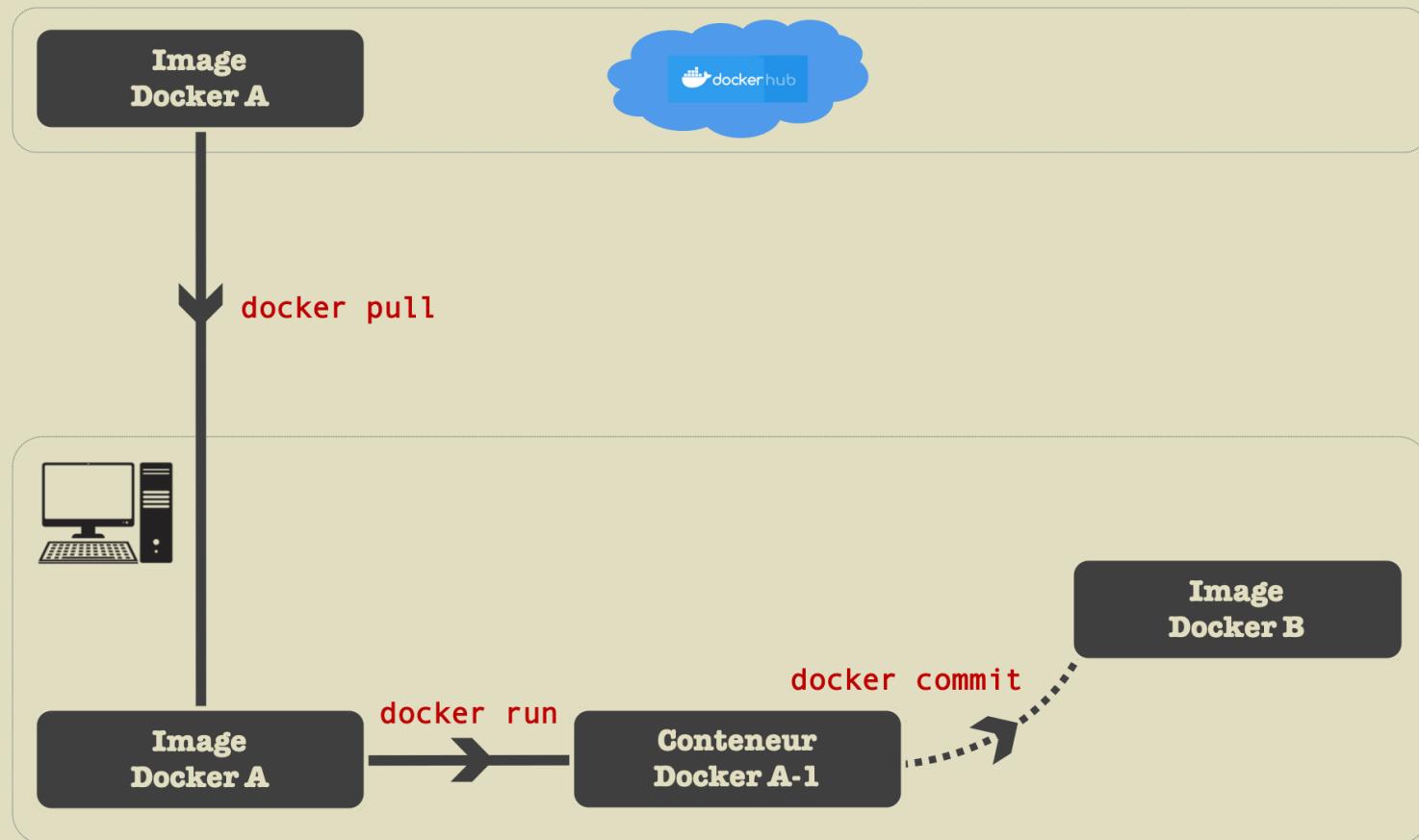
Résumé...



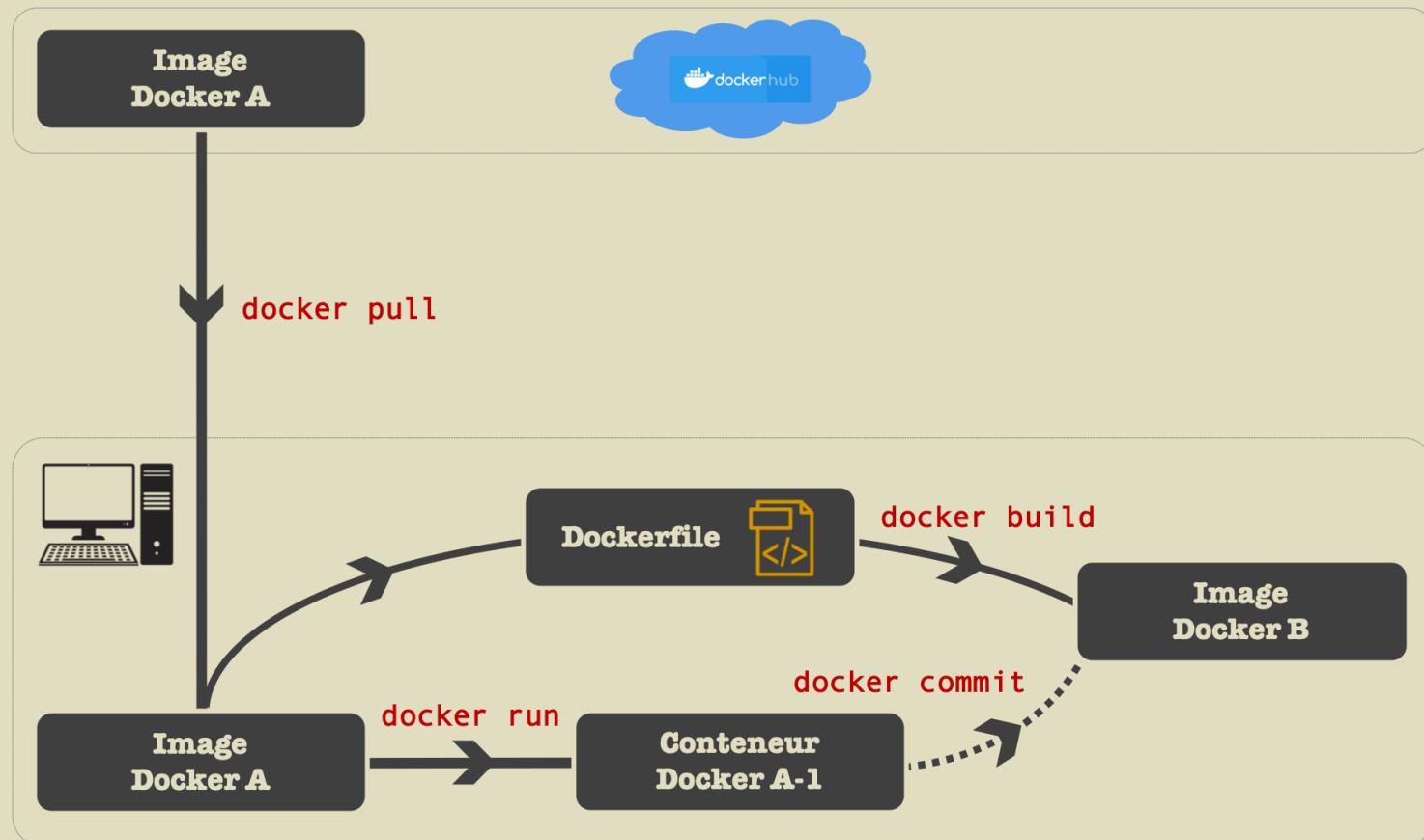
Résumé...



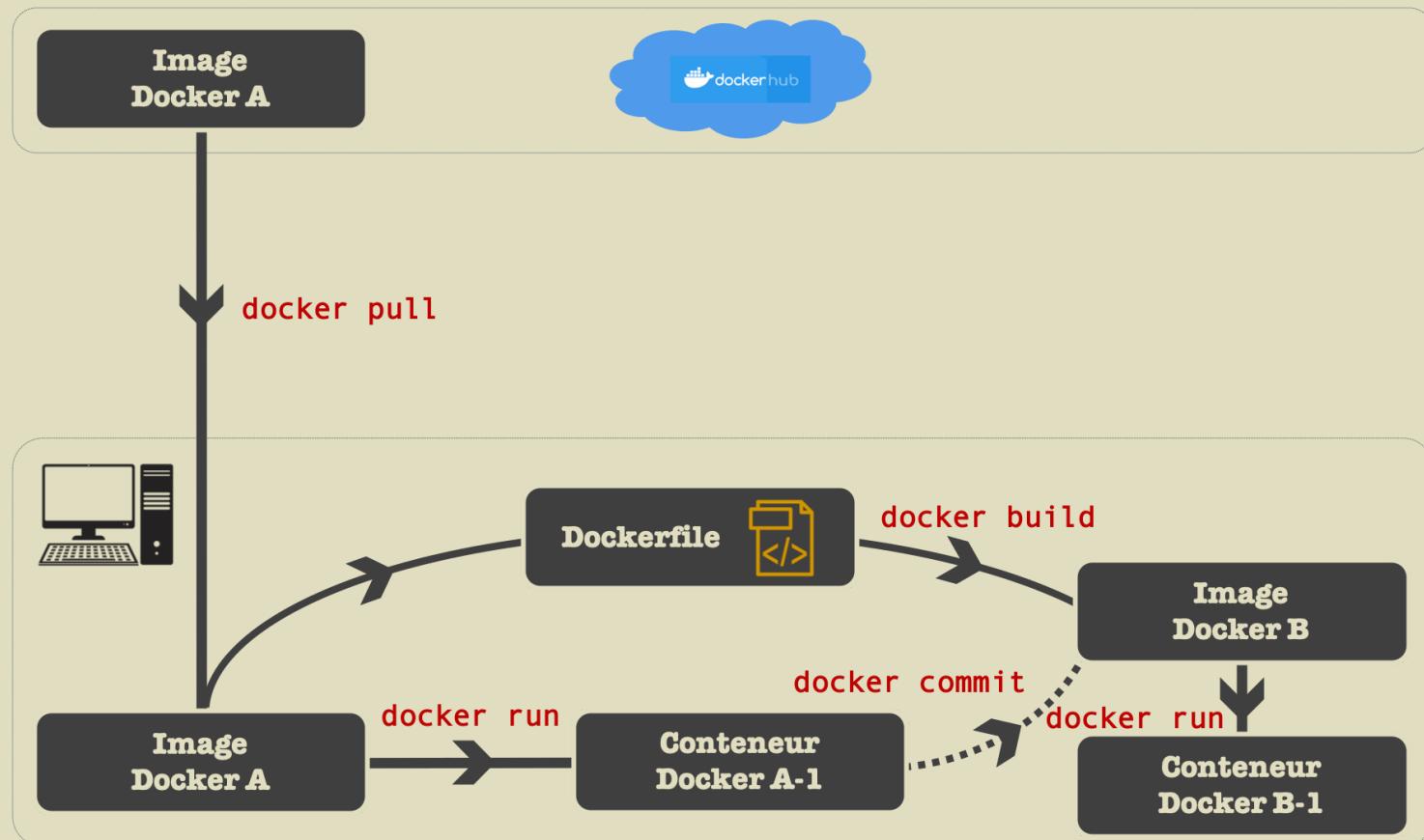
Résumé...



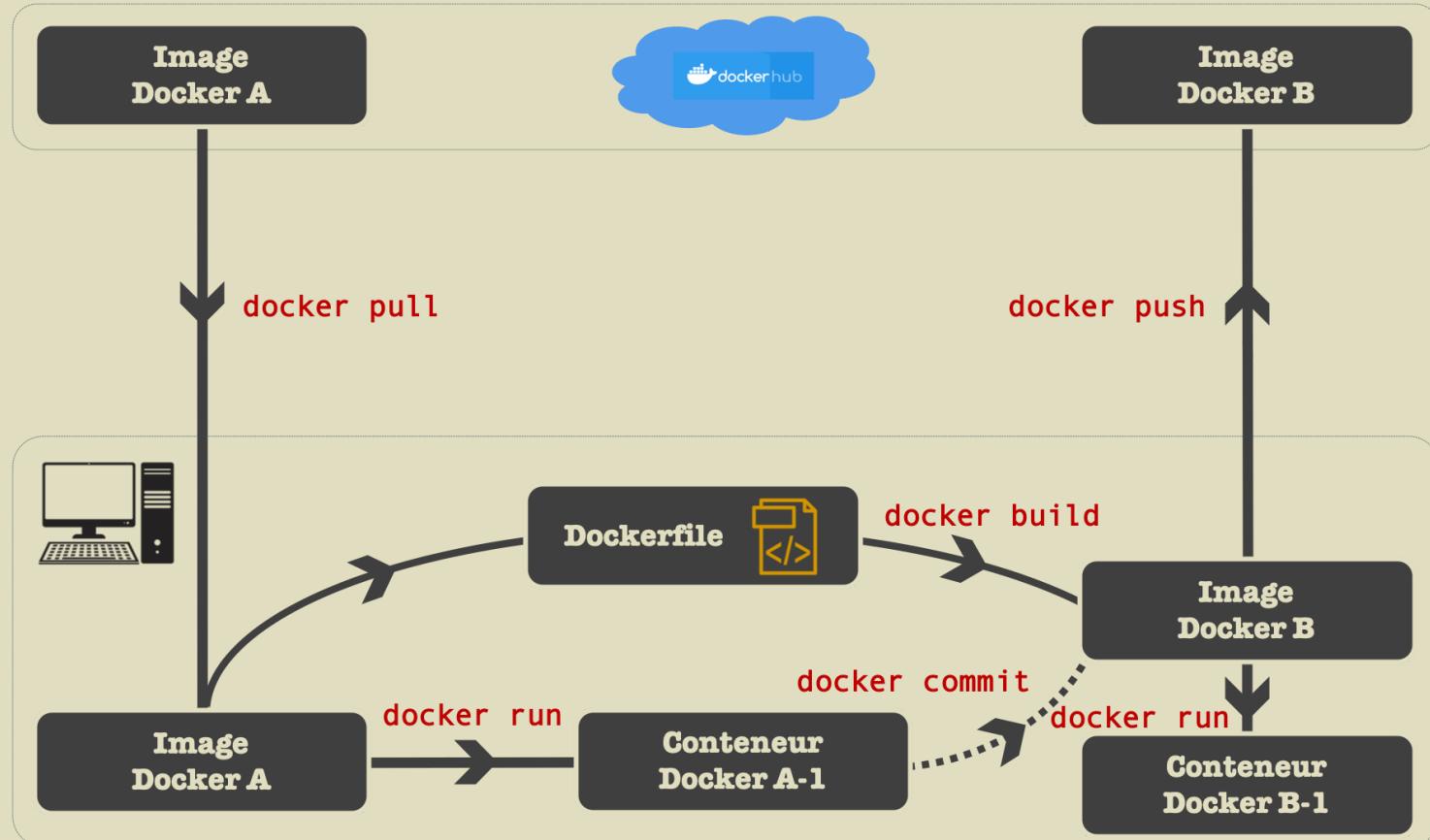
Résumé...



Résumé...



Résumé...



Partage d'une image

Publier sur le Docker Hub

👉 Nous allons voir qu'il est facile de publier une image 🚤 sur le **Docker Hub**. Pour cela, il nous faut tout d'abord nous créer un identifiant Docker en se rendant sur cette **page**. Une fois créé, nous devons stocker notre Docker ID sur notre machine local grâce à la commande suivante (nous devrons saisir l'ID et son mot de passe)

```
docker login
```

👉 Nous allons déposer notre image sur Docker Hub dans notre repository personnel (publique). Cependant, nous avons un problème, car pour ce faire, le nom de notre image doit contenir notre identifiant Docker : **docker_id/image**.

Pas de soucis : nous n'avons qu'à créer une nouvelle image à partir de notre **Dockerfile**. Comme elle sera identique, cela prendra 1s.

```
docker build -t nicolascasajus/buster .
```

Publier sur le Docker Hub

Vérifions :

```
docker images
```

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|-----------------------|--------|--------------|-------------|-------|
| buster | latest | b389eba1f1c9 | 2 hours ago | 121MB |
| nicolascasajus/buster | latest | b389eba1f1c9 | 2 hours ago | 121MB |
| debian | buster | 1510e8501783 | 2 weeks ago | 114MB |

👉 Docker lui a attribué la version `latest`. Pour publier sur le Hub, il est préférable de lui attribuer un autre tag, plus parlant. Allons-y

```
docker tag nicolascasajus/buster nicolascasajus/buster:1.0
```

Vérifions :

```
docker images
```

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|-----------------------|-----|--------------|-------------|-------|
| nicolascasajus/buster | 1.0 | b389eba1f1c9 | 2 hours ago | 121MB |

Publier sur le Docker Hub

👉 Il est temps de pusher !

```
docker push nicolascasajus/buster:1.0
```

Publier sur le Docker Hub

The screenshot shows the Docker Hub user profile for `nicolascasajus`. The profile page includes a blue header bar with the Docker Hub logo, a search bar, and navigation links for Explore, Repositories, Organizations, Get Help, and the user's name. Below the header is a profile card with a blue fingerprint icon, the user's name, a 'Edit profile' link, a 'Community User' badge, and a 'Joined October 28, 2020' timestamp. A navigation bar below the profile card has tabs for 'Repositories' (which is selected), 'Starred', and 'Contributed'. The main content area displays a single repository: `nicolascasajus/buster`, created by `nicolascasajus` and updated a few seconds ago. It is categorized as a 'Container'. To the right of the repository card are download and star counts (1 Download, 0 Stars).

Displaying 1 of 1 repository

| nicolascasajus/buster | 1 | 0 |
|---|----------|-------|
| By nicolascasajus • Updated a few seconds ago | Download | Stars |
| Container | | |

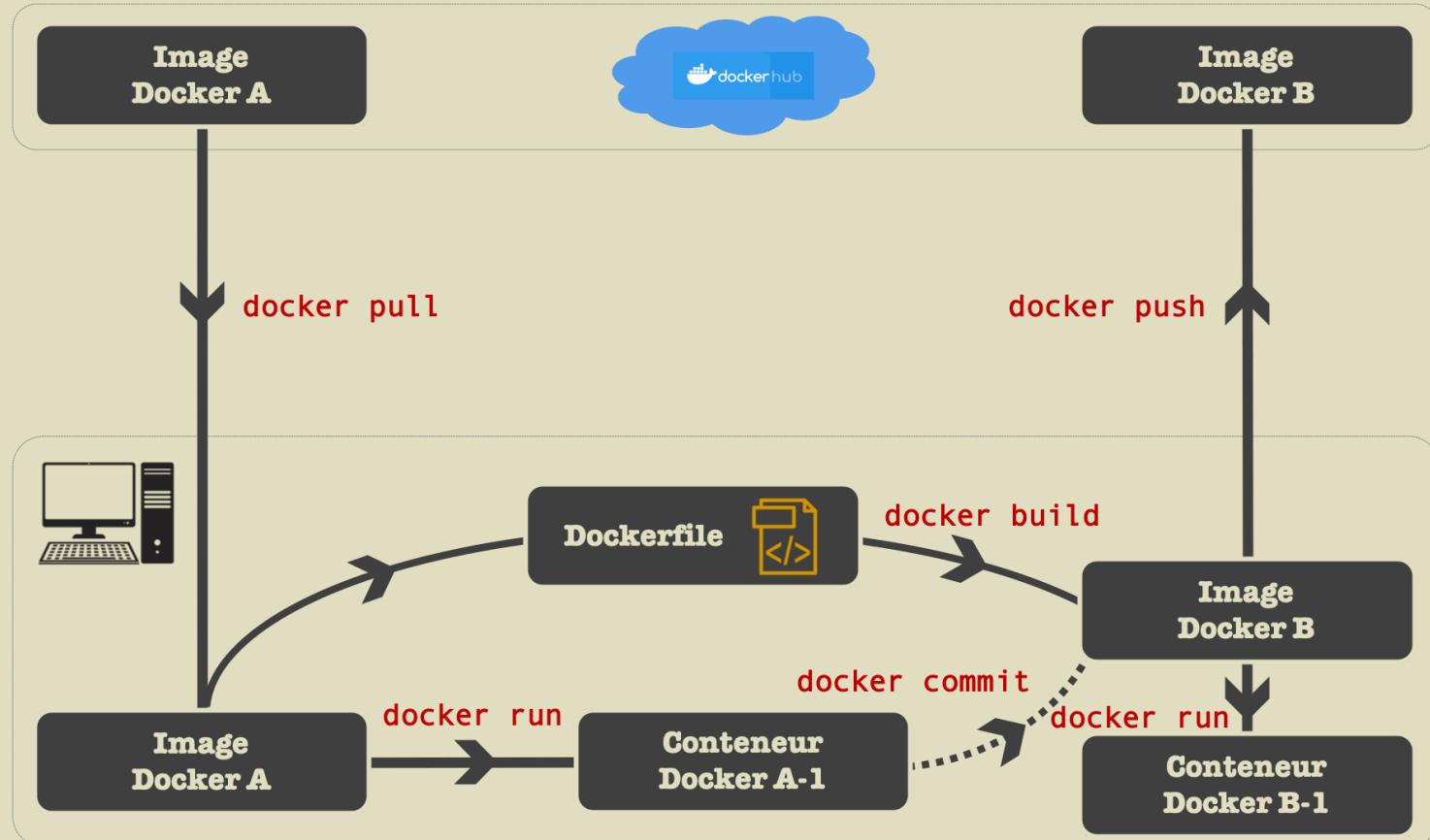
Publier sur le Docker Hub

The screenshot shows the Docker Hub interface for a repository named 'nicolascasajus/buster'. At the top, there's a blue header bar with the Docker Hub logo, a search bar, and navigation links for 'Explore', 'Repositories', 'Organizations', 'Get Help', and a user account dropdown. Below the header, the repository path 'Explore > nicolascasajus/buster' is shown, along with a message about private repositories and a 'Get more' link. The main content area features a large blue hexagonal icon representing the repository. The repository name 'nicolascasajus/buster' is displayed with a star icon, followed by the text 'By [nicolascasajus](#) • Updated a few seconds ago'. A 'Container' badge is present. On the right, there's a 'Manage Repository' button and a 'Pulls 1' badge. Below the icon, there are two tabs: 'Overview' (which is active) and 'Tags'. The 'Overview' section contains a placeholder message 'No overview available' and a note that the repository doesn't have an overview. To the right, there are two boxes: one for the 'Docker Pull Command' containing the text 'docker pull nicolascasajus/buster' and a copy icon, and another for the 'Owner' containing a profile picture and the name 'nicolascasajus'.

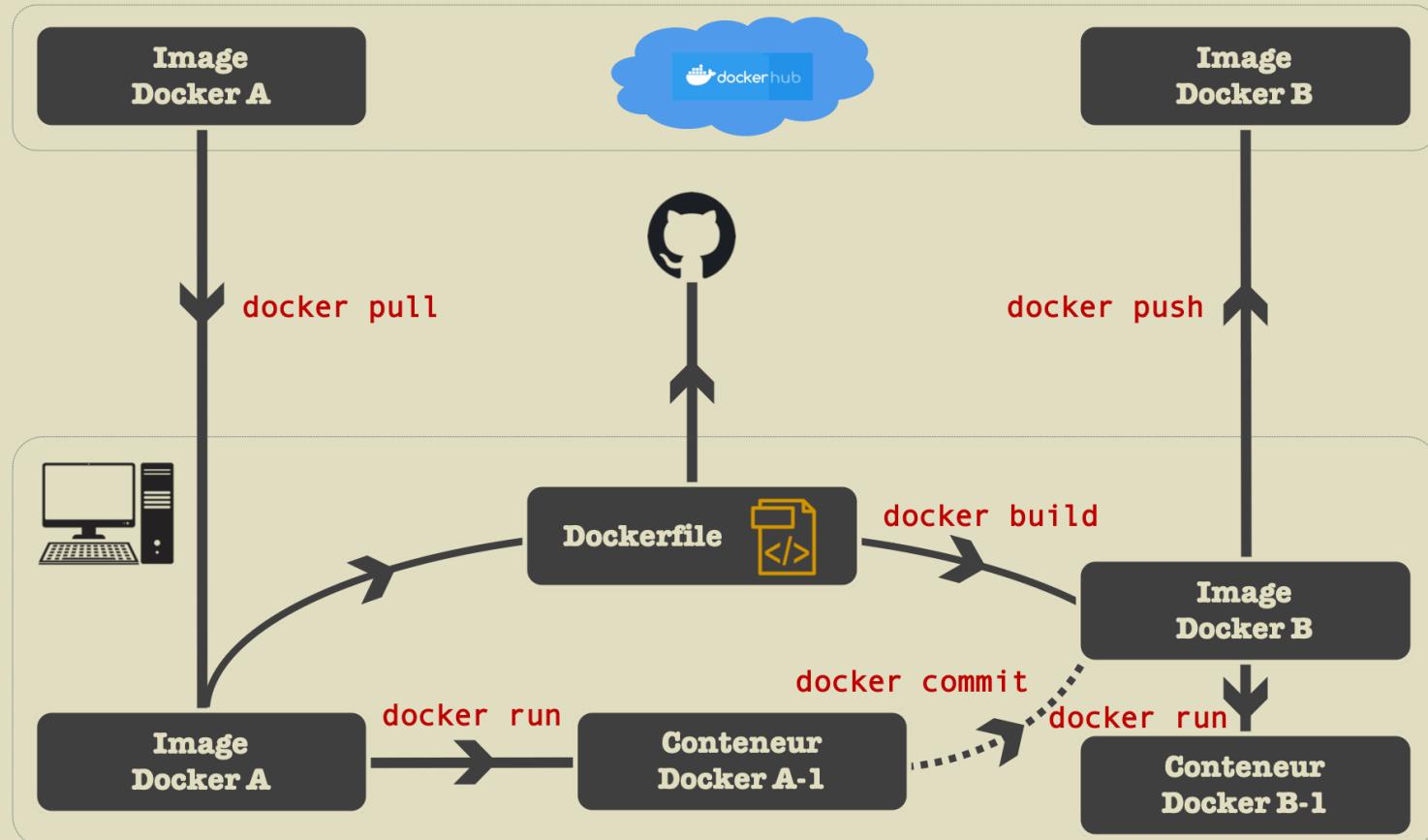
Publier sur le Docker Hub

The screenshot shows the Docker Hub interface for a repository named 'nicolascasajus/buster'. At the top, there's a search bar and navigation links for 'Explore', 'Repositories', 'Organizations', 'Get Help', and a user account section. Below the header, it displays 'Using 0 of 1 private repositories. [Get more](#)'. The main content area features a blue hexagonal icon, the repository name 'nicolascasajus/buster' with a star icon, and a 'Manage Repository' button. It shows '2 Pulls' and indicates the repository is a 'Container'. Below this, there are tabs for 'Overview' and 'Tags', with 'Tags' being the active tab. A 'Filter Tags' input field and a 'Sort by' dropdown set to 'Latest' are visible. Under the 'Tags' section, the '1.0' tag is listed, showing it was last pushed 3 minutes ago by 'nicolascasajus'. It includes a 'docker pull' command and a copy icon, a digest hash '7031cbf1595a', OS/ARCH information 'linux/amd64', and a compressed size of '51.03 MB'.

Publier sur le Docker Hub



Partager le Dockerfile





Initiative Rocker

Si on se rappelle la définition d'un conteneur, celui-ci contient un environnement de travail mais aussi, et surtout, une application. Or, il nous manque l'application. Et idéalement, une application développée sous ...

👉 L'initiative Rocker



Initiative Rocker



| image | description | size | pull |
|-------------|---|-------------------|-------------------|
| r-ver | Version-stable base R & src build tools | 303.6MB 16 layers | docker pulls 422k |
| rstudio | Adds rstudio | 355.1MB 21 layers | docker pulls 5.3M |
| tidyverse | Adds tidyverse & devtools | 712.7MB 26 layers | docker pulls 1.8M |
| verse | Adds tex & publishing-related packages | 1.2GB 30 layers | docker pulls 672k |
| geospatial | Adds geospatial packages on top of 'verse' | 1.7GB 32 layers | docker pulls 231k |
| shiny | Adds shiny server on top of 'r-ver' | 0B 13 layers | docker pulls 771k |
| shiny-verse | Adds tidyverse packages on top of 'shiny' | 0B 14 layers | docker pulls 144k |
| binder | Adds requirements to 'geospatial' to run repositories on mybinder.org | 0B 45 layers | docker pulls 73k |
| ml | Adds python and Tensorflow to 'tidyverse' | 0B 47 layers | docker pulls 20k |
| ml-verse | Adds python and Tensorflow to 'verse' | 0B 47 layers | docker pulls 20k |

👉 Ressources: [GitHub](#) | [Docker](#)

Demo

{sfrocker}

Pour aller plus loin...

👉 Docker-compose - L'orchestrateur ultime

```
version: "3.8"
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
      - logvolume01:/var/log
    links:
      - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```