



Introduction

Docker permet d'installer des solutions logicielles GNU/Linux ou MS Windows de son choix, dans les versions de son choix, quelle que soit notre système d'exploitation. Pour cela l'orchestrateur **isole** les logiciels qu'on souhaite utiliser les uns des autres avec leurs dépendances dans des "**conteneurs**" (images système). Mais il permet aussi d'éviter les inconvénients de la **virtualisation** (fichiers lourds, ressources machines divisées, lenteurs d'exécutions, etc.).

C'est donc une solution extrêmement pertinente lorsqu'il s'agit de déployer une plateforme de développement (afin de reproduire n'importe quel environnement de production), ou d'une manière générale de déployer des applications ou des versions de logiciels non supportées par le système courant. De plus, cela confère une portabilité très aisée et une grande souplesse à sa configuration.

Note(1) : l'ensemble des **commandes** proposées dans ce cours **sont à tester** sur votre terminal.

Note(2) : Dans ce TP, Docker peut être installé sur Windows ou GNU/Linux, **mais pas les deux** !

1/ Installation sur MS Windows 10 ou MS Windows 11 (intel Chipset)

Ouvrir **PowerShell** en mode administrateur et lancer les deux commandes :

```
Enable-WindowsOptionalFeature -Online -FeatureName containers -All  
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Hyper-V -All
```

Après le redémarrage du système :

- Télécharger et installer **Docker Desktop** et les **dépendances WSL** (GNU/Linux) pour Windows
- <https://desktop.docker.com/win/main/amd64/178610/Docker%20Desktop%20Installer.exe>
- https://wslstorestorage.blob.core.windows.net/wslblob/wsl_update_x64.msi

Après l'installation (qq minutes !), le redémarrage du système, lancer **Docker Desktop** (sur le bureau) accepter les termes de la licence et lancer la commande sur **PowerShell** : **docker version**

Si une erreur intervient avec le « *daemon* », lancer la commande **sur le terminal Windows** :

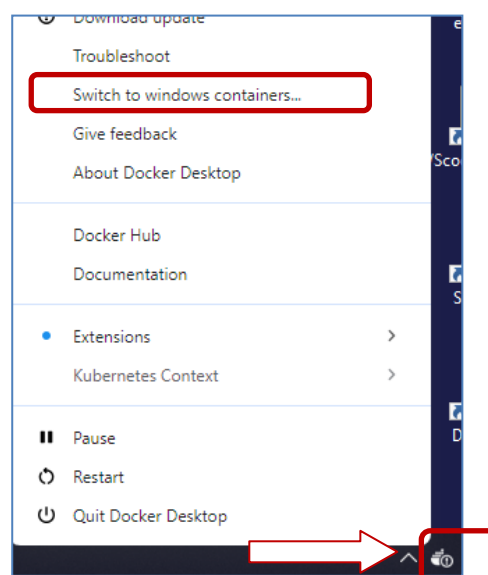
```
cd "C:\Program Files\Docker\Docker" && .\DockerCli.exe -SwitchDaemon
```

- Vérifier l'exécution des conteneurs **GNU/Linux** :



Si :
Switch to GNU/Linux containers...
s'affiche, cliquez sur l'option du menu.

Si :
Switch to windows containers...
s'affiche, laissez tel quel.



2/ Installation sur une distribution GNU/Linux de base Debian (Ubuntu, Mint, Mageia, Kali ...)

- Installer Docker en ligne de commande :

```
sudo su # mot de passe choisi lors de l'installation
apt update
apt -y install docker.io && apt -y install docker-compose
```

- Activation de Docker :

```
systemctl start docker
systemctl enable docker
systemctl status docker
docker-compose -h
```

```
lines 1-18/18 (END)
• docker.service - Docker Application Con
  Loaded: loaded (/lib/systemd/system/do
  Active: active (running) since Mon 202
    Docs: https://docs.docker.com
  Main PID: 6712 (dockerd)
    CGroup: /system.slice/docker.service
            └─6712 /usr/bin/dockerd -H fd:
```

CTRL +C pour sortir

3/ Premier test d'une image dockerisée

Docker inclus une image « **hello world** » afin de tester l'environnement de conteneurisation :

```
docker run hello-world
```

```
base login: non payé @ 10:00:00 on 04/08/2022
→ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
b8dfde127a29: Pull complete
Digest:
sha256:f2266cbfbc127c960fd30e76b7c792dc23b588c0db76233517e1891a4e357d519
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/
```

Lancer la commande proposée afin d'analyser le principe de la conteneurisation :

```
docker run -it ubuntu bash
```

```
→ docker run -it ubuntu bash
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
345e3491a907: Pull complete
57671312ef6f: Pull complete
5e9250ddb7d0: Pull complete
Digest:
sha256:cf31af331f38d1d7158470e095b132acd126a7180a54f263d386da88eb681d93
Status: Downloaded newer image for ubuntu:latest
root@b38443f9b962:/#
```

Dès lors, on se retrouve comme utilisateur « **root** » sur la version d'une distribution GNU/Linux dans notre propre système ! Afin de vérifier ceci, nous allons lire la version courante :

```
cat /etc/issue
```

```
root@b38443f9b962:/# cat /etc/issue
Ubuntu 20.04.2 LTS
```

Pour quitter l'image dockerisée, on sort :

```
exit
```

Pour arrêter l'image dockerisée :

```
docker stop ID
```

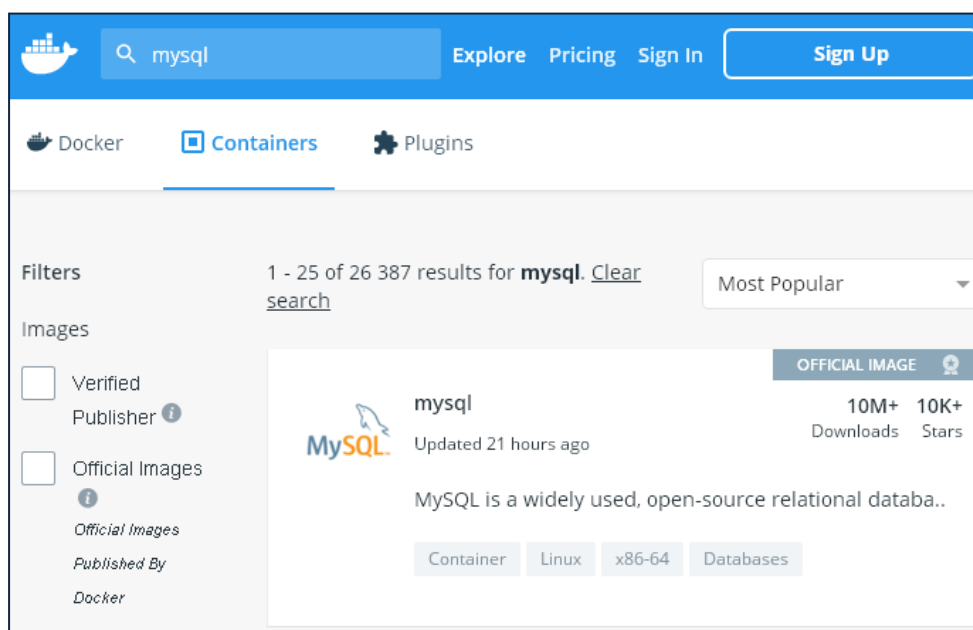
Pour supprimer l'ensemble des images téléchargées auparavant, et vider entièrement le stockage des données de Docker, on lance la commande :

```
docker system prune -a
```

Docker est alors remis à zéro.

4/ Test d'un environnement avec une image du SGBDR : MySQL

Docker Hub (hub.docker.com) propose de très nombreuses images pouvant être déployées. La plupart concernent des comptes comme le nôtre, et d'autres officielles, comme le SGBDR **MySQL** :



Afin de tester l'image conteneurisée de MySQL, dans un premier temps nous allons créer un nouveau paramètre réseau personnalisé sur le serveur DHCP virtualisé de Docker, celui-ci nommé « **local** » :

```
docker network create local
```

```
docker network ls
```

Par défaut une connexion par pont (Bridge) a été créée:

```
→ docker network create local
04d413155f1738646d2756159434dd7f46d6e6f1abf587e2c3532983d94bc1f9
→ docker network ls
NETWORK ID        NAME        DRIVER        SCOPE
359846a539ca     bridge     bridge        local
e065fbb4a75a     host       host          local
04d413155f17     local      bridge        local
869a6b4e69ab     none      null          local
```

Afin de connaître l'ensemble des commandes proposées par Docker, le paramètre « **help** » est indispensable: `docker -help`

Exécution de l'image MySQL :

```
docker run --name database --network local -e MYSQL_ROOT_PASSWORD=secret -d mysql:latest
```

Cette commande se compose de plusieurs paramètres à comprendre :

- 1/ **run** : lance la conteneurisation.
- 2/ **--name** : le nom digest (associé à un ID) de l'image.
- 3/ **--network** : le nom de notre réseau créé précédemment.
- 4/ **-e** : variable d'environnement incluse dans la documentation de l'image MySQL. Celle-ci ajoute un mot de passe à l'utilisateur « root ».
- 5/ **-d** : Detach, ici l'image sera lancée en arrière-plan sur notre OS (détachée de la commande interactive).

Docker va maintenant déployer l'image officielle de MySQL, et « *puller* » (télécharger et installer) l'ensemble des dépendances associées à l'image :

```
Unable to find image 'mysql:latest' locally
latest: Pulling from library/mysql
f7ec5a41d630: Pull complete
9444bb562699: Pull complete
6a4207b96940: Pull complete
181cefd361ce: Pull complete
8a2090759d8a: Pull complete
15f235e0d7ee: Pull complete
d870539cd9db: Pull complete
493aaa84617a: Pull complete
bfc0e534fc78: Pull complete
fae20d253f9d: Pull complete
9350664305b3: Pull complete
e47da95a5aab: Pull complete
Digest: sha256:04ee7141256e83797ea4a84a4d31b1f1bc10111c8d1b
```

Lancement d'un client MySQL avec la même image dockerisée

L'avantage de Docker est de pouvoir lancer une autre instance de notre image, sans forcément l'attacher en arrière plan à notre système – c'est-à-dire non incluse avec la commande « **-d** ».

Pour cela on lance la commande :

```
docker run -it --network local --rm mysql:latest mysql -hdatabase -uroot -p
```

Enter password : **secret**

Ici on se retrouve sur le client MySQL qui s'est connecté à l'image précédemment installée en arrière-plan (le serveur) . Afin de tester les contenus du SGBDR on lance la commande **SQL** :

SHOW DATABASES;

```
→ docker run -it --network local --rm mysql:latest mysql -hdatabase -uroot -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 9
Server version: 8.0.24 MySQL Community Server - GPL

Copyright (c) 2000, 2021, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show database;
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax to use near
'database' at line 1
mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| performance_schema |
| sys |
+-----+
```

On quitte le client MySQL : **exit ;**

Explications sur la commande du client MySQL :

```
docker run -it --network local --rm mysql:latest mysql -hdatabase -uroot -p
```

- 1/ **-it** : interactive + tty (l'image se lance en exécutant directement « l'intérieur » du conteneur, non en arrière-plan qui est l'option **-d** pour « detach »).
- 2/ **--rm** : on ne souhaite pas conserver l'image du Docker après son exécution (rm = retirée).
- 3/ **mysql** : nom de l'image à exécuter.
- 4/ **-h** : nom de l'hôte (créé précédemment) sans espace, *Container host name*
- 5/ **-u -p** : Commandes courantes afin de lancer un client MySQL (nom de l'utilisateur, et l'invite du mot de passe). Celles-ci sont passées comme variables d'environnement.

5/ Gestion des conteneurs Docker

La commande « **docker ps** » est indispensable pour savoir quelle (s) image(s) est actuellement lancée(s), activée(s) (ou non) par Docker en arrière-plan : `docker ps`

```
→ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED
23163307413b   mysql:latest  "docker-entrypoint.s..."  10 seconds ago
STATUS        PORTS        NAMES
Up 5 seconds   3306/tcp, 33060/tcp  database
```

=> CONTAINER ID

Un conteneur sera toujours accessible avec son identifiant unique, **ID**.

Si le conteneur possède un nom « => **NAMES** », l'id peut se substituer par ce nom.

Par exemple, la commande : `docker logs ID` nous permettra d'accéder à la journalisation du conteneur (logs).

Pour « rentrer » dans son image détachée qui est exécutée en arrière plan - le cas ici pour l'image « **mysql :latest** », la commande : `docker exec -u 0 -it ID bash` nous permettra d'accéder directement au système conteneurisé.

=> STATUS

- **Stopper une image :**

La commande « **docker ps** » affiche le statut de l'image depuis sa première exécution.

Pour arrêter cette exécution la commande : `docker stop ID` ou `docker stop database` arrêtera la *stack* (processus empilés) de l'image.

Ici, le fait de relancer la commande `docker ps` n'affichera rien, car aucune image Docker n'est exécutée.

Dans ce cas, l'option « **-a** » (*active*) : `docker ps -a` affiche un autre statut : `Exited (0)`

- **Relancer l'image :**

Pour redémarrer l'image, la simple commande `docker start database` relancera la stack du conteneur.

Tester à nouveau les commandes : `docker ps -a` et `docker ps`

=> PORTS

La correspondance des ports se fait entre l'hôte et le conteneur :

- Ici le port **3306** (classique pour MySQL), sera actif sur le système réel (source).
- Le port **33060** sera la porte d'entrée / la cible sur le conteneur (destination).

Ces deux ports communiqueront via le réseau créé précédemment, nommé « **local** ».

6/ Gestion d'une donnée permanente

Par définition un conteneur utilise la donnée de trois façons distinctes :

1/ Temporairement : une image se lance avec sa configuration d'origine, on exécute ce qu'on a besoin, puis, après le lancement, la donnée disparaît. C'est par exemple le mode *interactive* associé à *remove* (**-it --rm**).

2/ Gérée par Docker : une image est détachée (**-d**), mais reste active grâce au gestionnaire d'images de Docker. Si on reboot notre système hôte, la donnée sera **persistante** - tant que l'image sera détachée et active en arrière-plan par le gestionnaire Docker, visible avec **docker ps**.

3/ Permanente : Dans ce cas on utilise un répertoire de notre système hôte afin de conserver la donnée (**-v** pour volume). L'image va se baser sur notre système hôte pour conserver cette donnée, et donc pourra être réutilisée dans le cas où une image ne sera plus exécutée par le gestionnaire Docker.

- **Exemple avec le conteneur MySQL**

Si l'on relance notre client :

```
docker run -it --network local --rm mysql:latest mysql -hdatabase -uroot -psecret
```

Qu'on crée une nouvelle base donnée : **CREATE DATABASE test;**

Qu'on l'affiche : **SHOW DATABASES;**

On voit bien notre nouvelle base de données sur le conteneur MySQL :

```
mysql> create database test;
Query OK, 1 row affected (0.00 sec)

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| performance_schema |
| sys |
| test |
+-----+
5 rows in set (0.00 sec)

mysql> exit
Bye
```

Le fait de relancer à nouveau le client MySQL affichera à nouveau la base de données « **test** » :

```
docker run -it --network local --rm mysql:latest mysql -hdatabase -uroot -psecret
```

SHOW DATABASES;

```
mysql
performance_schema
sys
test
5 rows in set (0.00 sec)
```

Ici la donnée est **persistante**, et gérée par Docker.

- **Suppression du conteneur MySQL**

Auparavant nous avons vu la possibilité d'arrêter un conteneur avec « **stop** », le lancer avec « **start** », mais le fait de l'arrêter ne le retire pas intégralement l'instance du conteneur sur Docker. Pour cela la commande « **rm** » (*remove*) doit être utilisée :

```
docker stop database && docker rm database
```

Ici notre conteneur n'est donc plus actif : `docker ps -a` , ni présent : `docker ps`

```
→ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS        NAMES
```

- **Vérification de la donnée, et création d'un nouveau conteneur MySQL**

Afin de vérifier l'état de la donnée, on récré à nouveau notre conteneur MySQL :

```
docker run --name database --network local -e MYSQL_ROOT_PASSWORD=secret -d mysql:latest
```

On lance le client :

```
docker run -it --network local --rm mysql:latest mysql -hdatabase -uroot -psecret
```

L'état de la donnée indique qu'il n'y a plus la base de données nommée « **test** » :

```
SHOW DATABASES;
```

```
mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| performance_schema |
| sys |
+-----+
4 rows in set (0.01 sec)

mysql> exit
Bye
```

Ici la donnée (**test**) a été intégralement effacée, car l'option « **rm** » a supprimé le conteneur actif depuis Docker.

Nous souhaitons maintenant conserver la donnée en permanence afin de la restituer dans n'importe quel conteneur MySQL.

- Configuration d'une donnée permanente

Dans un premier temps, on retire à nouveau l'image de MySQL :

```
docker stop database && docker rm database
```

Afin de conserver la donnée, nous allons créer un dossier à la racine de notre dossier personnel :

```
cd
pwd
```

Ici le chemin absolu de notre répertoire personnel s'affiche

Création du dossier de stockage : `mkdir db-data`

On affiche son contenu : `ls -al db-data`

```
→ ls -al db-data
drwxr-xr-x@ 28 user  staff      896  3 mars 14:20 .
drwxr-xr-x+ 22 user  staff      704  3 mars 14:18 ..
```

Ici il n'y a rien dans ce dossier. C'est normal, il vient d'être créé...

Création d'une nouvelle instance du conteneur MySQL avec l'option « **-v** » associé à notre répertoire personnel (attention commande sur une seule ligne) Note : sur Windows remplacer `$(pwd)/` par `.\`

```
docker run --name database -v $(pwd)/db-data:/var/lib/mysql --network local -e
MYSQL_ROOT_PASSWORD=secret -d mysql:latest
```

```
docker ps
```

```
ls -al db-data
```

```
→ ls -al db-data
total 396264
-rw-r-----  1 user  staff      196608  3 mars 14:21 #ib_16384_0.dblwr
-rw-r-----  1 user  staff     8585216  3 mars 14:20 #ib_16384_1.dblwr
drwxr-xr-x-- 12 user  staff        384  3 mars 14:20 #innodb_temp
drwxr-xr-x@ 28 user  staff        896  3 mars 14:20 .
drwxr-xr-x+ 22 user  staff        704  3 mars 14:18 ..
-rw-r-----  1 user  staff         56  3 mars 14:20 auto.cnf
-rw-r-----  1 user  staff    3118706  3 mars 14:20 binlog.000001
-rw-r-----  1 user  staff        156  3 mars 14:20 binlog.000002
-rw-r-----  1 user  staff         32  3 mars 14:20 binlog.index
-rw-r-----  1 user  staff       1680  3 mars 14:20 ca-key.pem
-rw-r--r--  1 user  staff       1112  3 mars 14:20 ca.pem
-rw-r--r--  1 user  staff       1112  3 mars 14:20 client-cert.pem
```

Le dossier « **db-data** » est maintenant associé à celui du répertoire présent dans le conteneur à l'emplacement : « **/var/lib/mysql** ». Un lien symbolique a été monté (ciblé) entre ces deux répertoires (source et destination). La donnée est donc stockée dans notre répertoire du système hôte, et restera si le conteneur MySQL sera effacé.

La commande bash : `$(pwd)` permettra dynamiquement de retourner le chemin absolu de notre répertoire personnel.

- Test de la configuration d'une donnée permanente

Le conteneur MySQL est maintenant configuré afin de stocker les bases de données au niveau de notre système hôte (réel). On relance notre client MySQL, et l'on crée à nouveau une base de données nommée « **test** » :

```
docker run -it --network local --rm mysql:latest mysql -hdatabase -uroot -psecret
```

```
CREATE DATABASE test;
```

```
SHOW DATABASES;
```

```
exit;
```

On supprime (encore une dernière fois !) notre conteneur MySQL :

```
docker stop database && docker rm database
```

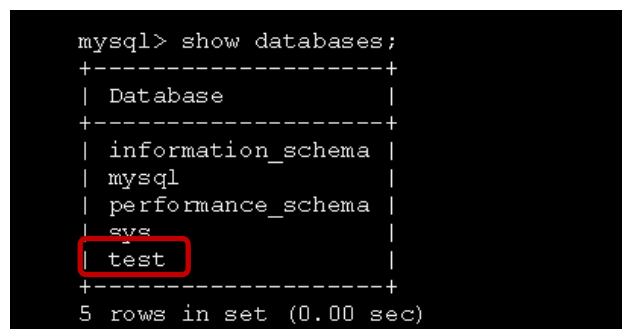
On crée (à nouveau) notre conteneur MySQL avec l'attachement du volume local:

```
docker run --name database -v $(pwd)/db-data:/var/lib/mysql --network local -e  
MYSQL_ROOT_PASSWORD=secret -d mysql:latest
```

On lance le client :

```
docker run -it --network local --rm mysql:latest mysql -hdatabase -uroot -psecret
```

```
SHOW DATABASES;
```



```
mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| performance_schema |
| sys |
| test |
+-----+
5 rows in set (0.00 sec)
```

Ici la base de données "test" a été conservée !

7/ Portainer : l'interface graphique de Docker

Portainer permet de gérer via une interface web pour les conteneurs locaux ou distants. On trouve l'application sous forme d'image docker.

- Site du projet : <https://portainer.io/>
- Dépôt Git du projet : <https://github.com/portainer>

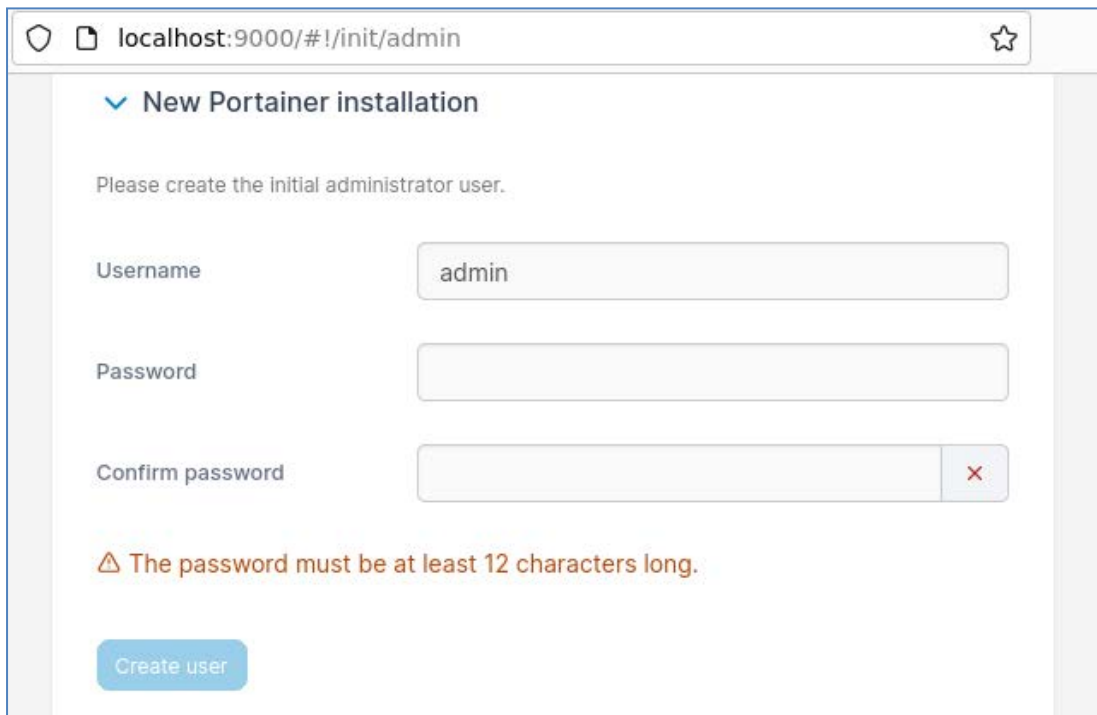
Pourquoi utiliser Portainer ?

- L'interface web de Portainer offre une représentation visuelle de l'infrastructure Docker, la rendant plus facile à comprendre et à gérer.
- Contrôle d'accès basé sur les rôles (RBAC) : Portainer permet de définir des rôles et des autorisations pour les utilisateurs, améliorant la sécurité et le contrôle d'accès.
- Surveillance des ressources comme l'utilisation du CPU, la mémoire et le réseau des conteneurs et des services.
- Portainer fournit une bibliothèque de modèles d'applications préconfigurés, simplifiant la création de conteneurs.

Installation de l'image Docker de Portainer (attention commande sur une seule ligne) :

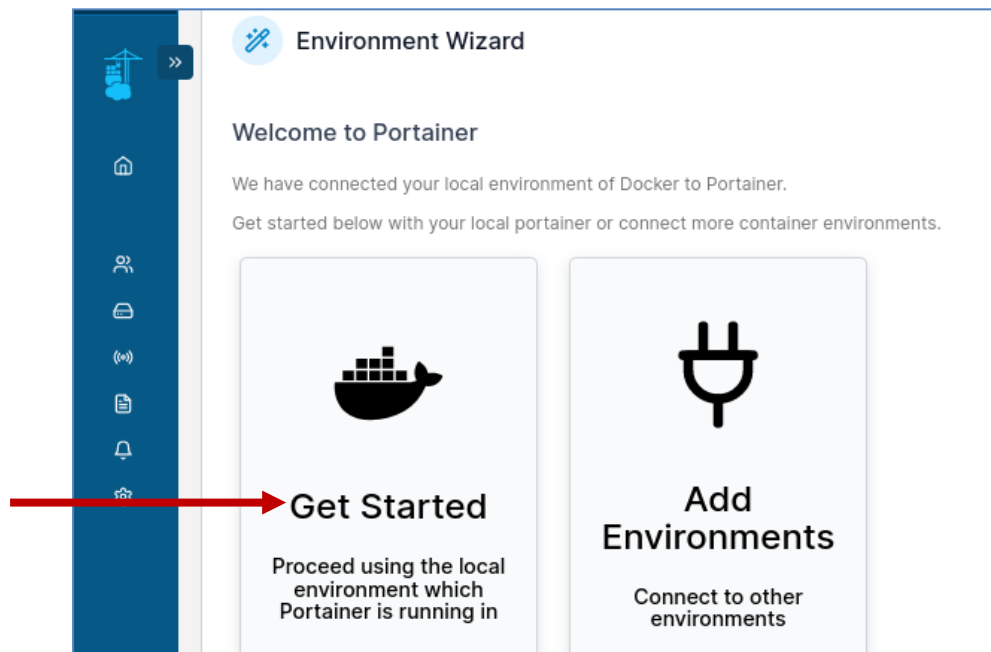
```
docker run -d -p 9000:9000 --name portainer --restart always -v /var/run/docker.sock:/var/run/docker.sock -v portainer_data:/data portainer/portainer-ce
```

Une fois Portainer opérationnel, vous pouvez accéder à son interface web en naviguant vers <http://localhost:9000>

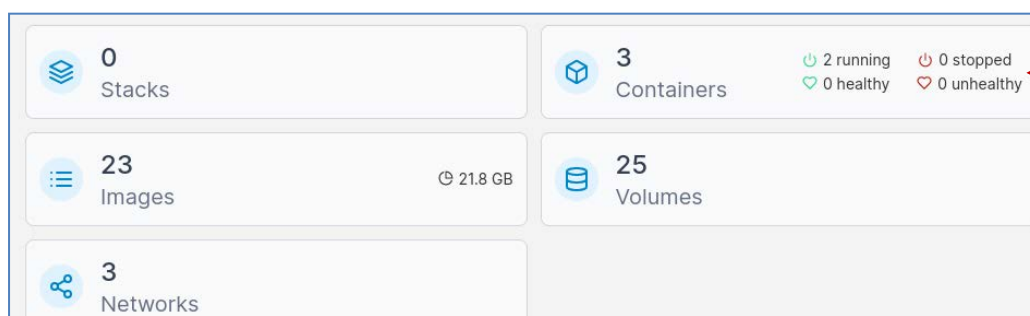
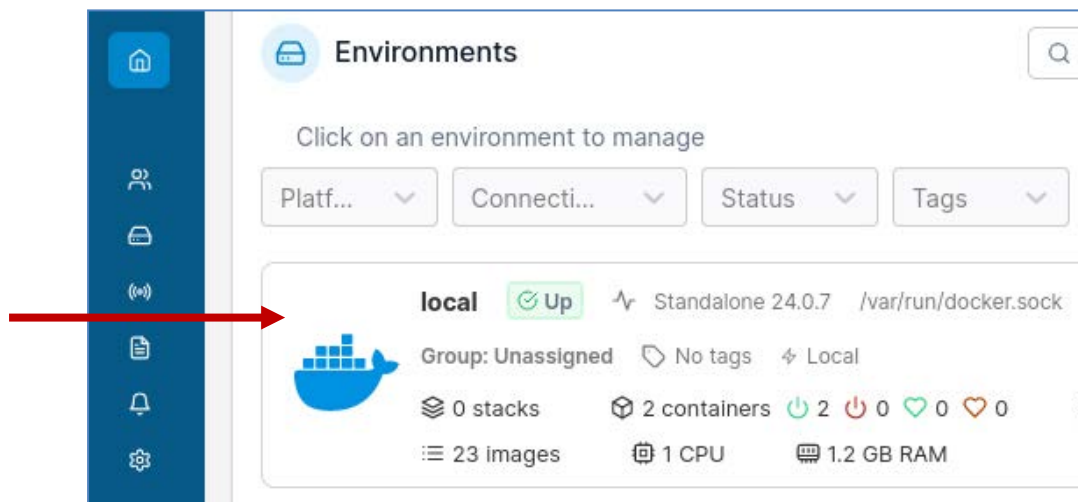


La capture suivante propose de gérer une instance locale ou distante de Docker.

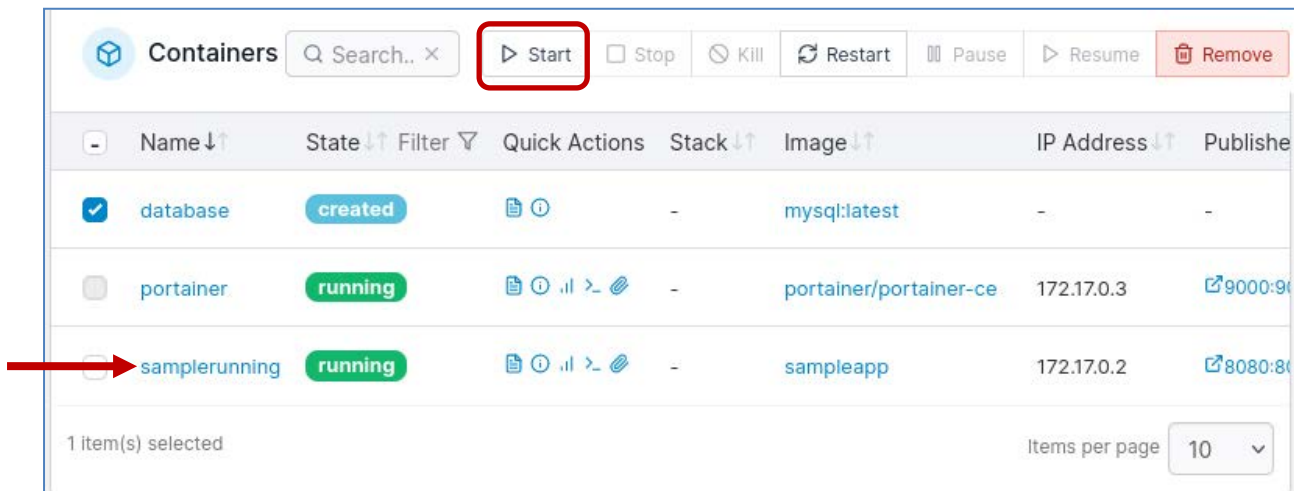
- Sélectionner : « **Get Started (...)** » pour gérer une instance locale :



L'interface, très intuitive, permet d'avoir un aperçu global, de gérer les éléments existants (conteneurs, stacks, images, volumes, réseaux), de créer d'autres conteneurs via notamment des templates et de personnaliser l'installation d'une nouvelle image. La partie administration permet de gérer les utilisateurs et les registres.

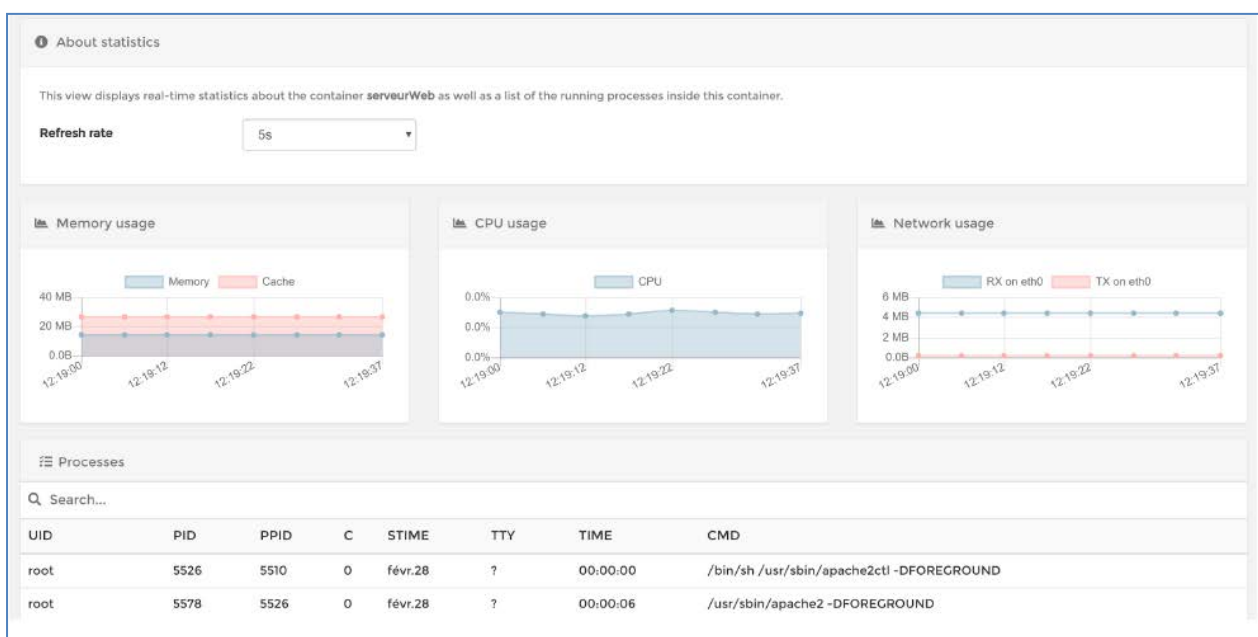


Un clic sur « Containers » affiche une liste détaillée des containers avec leur état ainsi que des informations utiles (nom, image, IP, ports, ...). Il est possible d'effectuer certaines actions sur ces containers (le démarrer, le stopper, consulter les stats, les logs, accéder à une console, etc.) :



Pour chaque conteneur, on retrouve des informations utiles ainsi qu'un accès aux statistiques détaillées, aux logs ainsi qu'à la console (via « Quick actions » ou en cliquant sur le conteneur voulu).

Les statistiques sont en temps réel et permettent de suivre l'état du CPU, de la mémoire, l'utilisation du réseau ainsi que les processus en cours d'exécution :



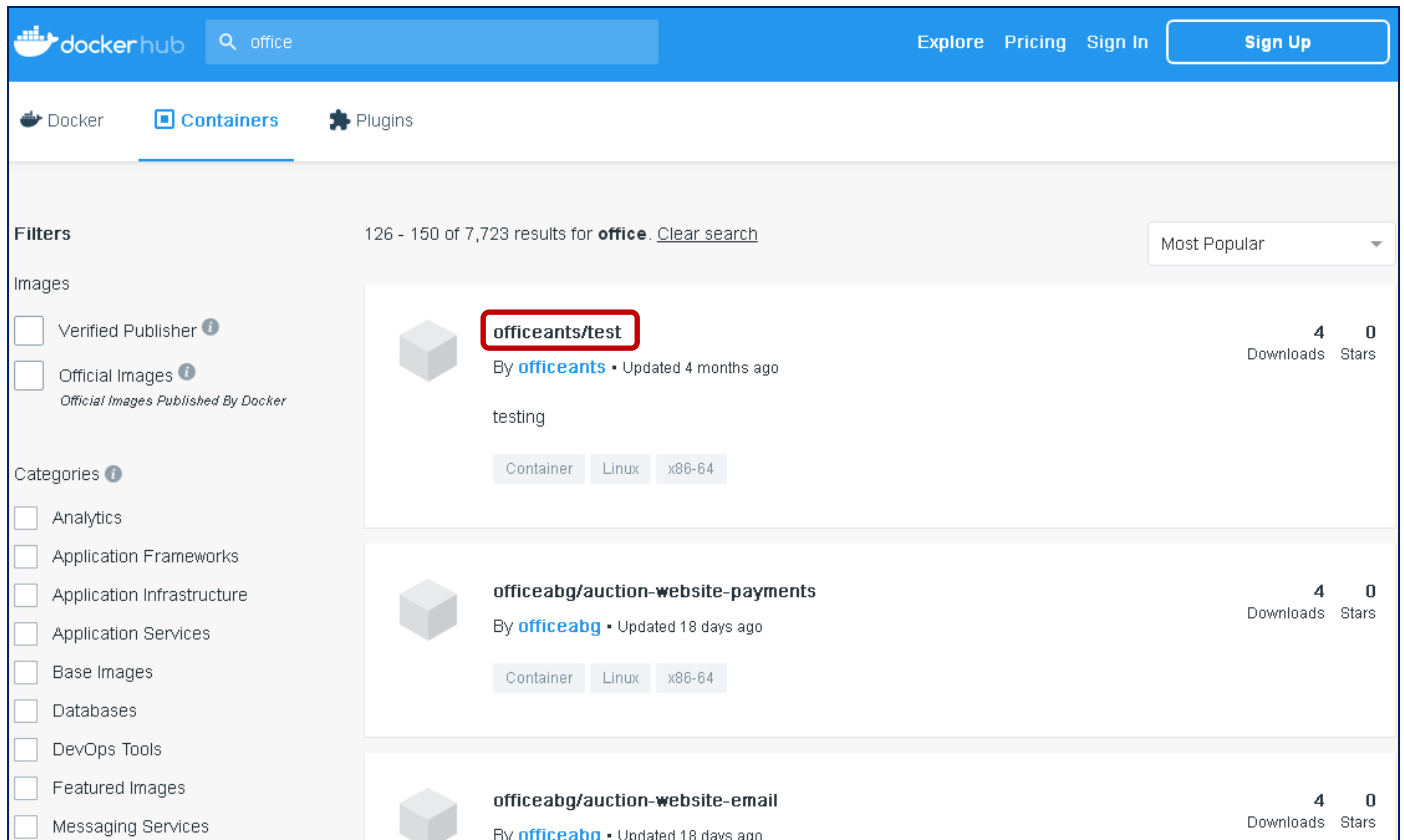
Toutes les autres possibilités et fonctionnalités sont intéressantes, libre à vous de les explorer et de les tester ...



Sachant que **DockerDesktop** affiche partiellement les mêmes fonctionnalités que **Portainer**, pourquoi déployer **Portainer** sur un serveur ?

8/ Compilation et distribution d'un conteneur personnalisé

Par nature, [Docker Hub](#) (registre public d'images Docker) permet le stockage et la publication des conteneurs créés avec Docker. Pour les identifier, on associe l'identifiant du développeur avec le nom du conteneur (**id/nom**) :



Nous allons partager un conteneur à la communauté - et donc à notre équipe, à partir d'un projet « Node » factice. Ce projet pouvant être aussi développé sous un autre langage, comme PHP, JAVA, PYTHON ...

Dans un premier temps, on crée un dossier « node » dans lequel on va également créer un simple fichier « **index.js** ». Ce fichier sera ultérieurement exécuté par JavaScript avec un serveur conteneurisé incluant [node.js](#) :

```
mkdir node && cd node
```

```
echo "console.log('Hello world !')" > index.js
```

```
ls
```

```
→ pwd
/home/user
→ mkdir node
→ cd node
→ node % echo "console.log('hello world !')" > index.js
→ node % ls
index.js
```

Afin de compiler notre premier conteneur, un fichier « **Dockerfile** » doit être créé :

nano Dockerfile

```
FROM node:latest
WORKDIR /app
ADD ./index.js /app/index.js
CMD node /app/index.js
```

GNU nano 2.5.3 Fichier : Dockerfile Modifié

```
FROM node:latest  ← Ici Docker va inclure le conteneur officiel de Node.js !
WORKDIR /app      ← Répertoire de travail (ou d'exécution) de Node dans le conteneur
ADD ./index.js /app/index.js  ← On ajoute la source vers la destination
CMD node /app/index.js  ← On exécute Node dès le lancement du conteneur
```

^G Aide ^O Écrire ^W Chercher ^K Couper ^J Justifier ^C Pos. cur.
^X Quitter ^R Lire fich. ^\ Remplacer ^U Coller ^T Orthograp. ^_ Aller lig.

Sélectionner **CTRL+O** , et ensuite la touche **ENTREE** pour enregistrer.
CTRL+X pour quitter.

Lancement de la compilation du conteneur avec l'option « **-t** » pour TAG (le nom de l'image qui sera exécutée à la prochaine étape) :

docker build . -t node-test

```
→ node % docker build . -t node-test
[+] Building 0.2s (8/8) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 36B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/node:latest
=> [internal] load build context
=> => transferring context: 29B
=> [1/3] FROM docker.io/library/node:latest
=> CACHED [2/3] WORKDIR /app
=> CACHED [3/3] ADD ./index.js /app/index.js
=> exporting to image
=> => exporting layers
=> => writing image sha256:65379965daac6c43a638f044d1bb608628e2444d213fa
=> => naming to docker.io/library/test
0.0s
```

Au final, lancement de notre conteneur (*sans le conserver en arrière-plan après son exécution*) :

docker run --rm node-test

```
=> => naming to docker.io/library/test
0.0s
→ node % docker run --rm node-test
Hello world !
```

- **Changer et attribuer des TAGS à notre conteneur**

Afin de distribuer la bonne version de notre conteneur, l'option « **tag** » permet de modifier ses paramètres. Exemple :

Ici on considère la dernière version portant le numéro "**1.0.0**" :

```
docker tag node-test:latest node-test:1.0.0
```

```
docker run --rm node-test:1.0.0
```

Le conteneur sera lancé avec son tag !

Ou avec simplement avec son nom :

```
docker run --rm node-test
```

Pour renommer ce tag en « **node-dev** » :

```
docker tag node-test:1.0.0 node-test:dev
```

```
docker run --rm node-test:dev
```

- **Publication et partage du conteneur sur Docker Hub**

Après la création de son compte sur hub.docker.com , on associe l'identifiant de son compte à l'image conteneurisée :

```
docker tag node-test:latest moncompte/node-test:dev
```

```
docker login Mon Id / mot de passe → LOGIN SUCCESSFUL
```

```
docker push moncompte/node-test:latest
```

Désormais notre conteneur est disponible pour notre équipe, ou soit pour un autre environnement de développement...

- **Tester le conteneur publié sur le registre**

Dans un premier temps, on supprime l'image créée précédemment :

```
docker image rm moncompte/node-test:latest
```

Puis on le pull depuis Docker Hub :

```
docker run --rm moncompte/node-test:latest
```

```
→ node % docker run --rm my-docker-id/node-test:latest  
hello world !
```

Pour terminer - si l'on souhaite, on vide les images téléchargées afin de libérer de l'espace sur notre système (**après avoir stoppés et retirés tous les conteneurs !**) :

- Seulement les images concernées :

```
docker images docker rmi image_id
```

- Conteneurs et images qui sont inutilisés :

```
docker volume prune docker image prune
```

- Tout ce qui a été déployés !

```
docker system prune -a
```


Index du cours

- [Introduction](#) (p1)
- 1/ [Installation sur MS Windows 10, Windows 11](#) (p1)
- 2/ [Installation sur une distribution GNU/Linux](#) (p2)
- 3/ [Premier test d'une image dockerisée](#) (p2)
- 4/ [Test d'un environnement avec une image du SGBDR : MySQL](#) (p3)
- 5/ [Gestion des conteneurs Docker](#) (p6)
- 6/ [Gestion d'une donnée permanente](#) (p7)
- 7/ [Portainer : l'interface graphique de Docker](#) (p11)
- 8/ [Compilation et distribution d'un conteneur personnalisé](#) (p14)