TP: Docker

1 Introduction

L'objectif de ce TP est de vous donner une initiation à l'utilisation de Docker et de vous familiariser avec ses principales commandes. Il vous est demandé de répondre aux questions (de façon lisible), ce qui vous permettra de mémoriser les commandes.

Le TP se fera via une MV Debian 11 ("bullseye", version stable depuis aout 2021), essentiellement dans un shell. Il s'appuie sur vos compétences dans un shell Linux, et suppose que vous maitrisez les commandes de base d'un environnement type Debian.

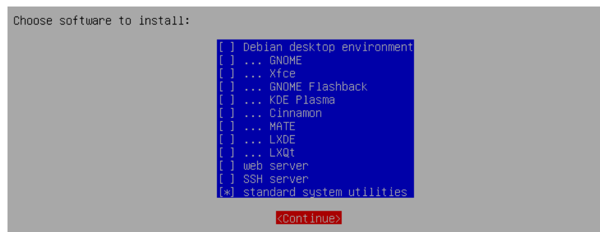
2 Installation de la MV et de Docker

Q2.1 - Créer une nouvelle MV dans VirtualBox avec 4096 MB de RAM et un disque de 32 GB.

Q2.2 - Télécharger l'iso :
`https://cdimage.debian.org/debian-cd/current/amd64/iso-cd/debian-11.2.0-amd64-netinst.iso`

Q2.3 - Démarrer la MV et installer l'OS en suivant ces consignes :

- choisir l'install non-graphique
- laisser la langue en anglais mais bien indiquer le clavier français.
- donner "root" comme mdp admin et créer un utilisateur **`user`** avec le mdp : user.
- laisser les options par défaut pour le partitionnement du disque
- lors de la sélection du miroir pour les paquets, bien choisir la France
- lors de la sélection des logiciels à installer, **décocher** les bureaux mais laisser les utilitaires :



- lors de l'install de **`grub`**, ne pas laisser le choix par défaut mais sélectionner l'install sur le disque.

Q2.4 - Une fois l'OS installé, se logger en "root", et après un **`apt update/apt upgrade`**, installer le bureau **`xfce4`**

Q2.5 - Ajouter **`user`** au groupe des sudoers avec : **`$ /sbin/usermod -aG sudo user`**

Q2.6 - Redémarrer la machine avec **`$ shutdown -r now`**, puis une fois le bureau actif, se logger en "user". Via le menu "Application->Settings->Display", régler la définition d'écran sur 1280x960.

Q2.7 - Via le terminal "xterm", passer "root" avec **`$ su`** et installer les paquets **`mousepad`** (editeur texte graphique léger), **`firefox-esr`**, **`sudo`**, **`xfce4-terminal`** (beaucoup plus convivial que "xterm", le terminal par défaut), et **`unzip`** :

```
$ apt install firefox-esr sudo xfce4-terminal mousepad unzip
```

Q2.8 - Se connecter depuis la MV sur `https://docs.docker.com/engine/install/debian/` et suivre la procédure recommandée pour installer Docker (en restant connecté en root).

Q2.9 - Quitter le mode "root" avec **`$ exit`**, vérifier que vous etes bien **`user`** avec **`$ whoami`**, et donner la version installée avec **`$ sudo docker version`** : [ ]

Post-install En l'état, l'utilisation du client implique les droits "admin" et donc l'usage de **`sudo`** pour chaque commande, ce qui est un peu ennuyeux. Effectuer la manip décrite ici :
`https://docs.docker.com/engine/install/linux-postinstall/`

Se relogger en "user" et vérifier que vous pouvez utiliser le client Docker sans "sudo".

3 Exercice 1 : premier conteneur

Q3.1 - Dans un shell, taper la commande suivante, et vérifier le fonctionnement :
`$ docker run hello-world`

Q3.2 - Afficher la liste des conteneurs avec **`$ docker ps -a`** (-a pour "all")
Combien de conteneurs avez-vous ?

Q3.3 - Relancer une deuxième fois **`$ docker run hello-world`**
puis reafficher la liste de tous les conteneurs. Combien en avez-vous ?
Pourquoi ?

Q3.4 - Lancer la commande **`$ docker info`** et observez la sortie. Que faut-il taper pour afficher uniquement le nombre de conteneurs locaux : (indice : grep)

4 Conteneur BusyBox

Q4.1 - On utilise ici une "mini-distribution" Linux, connue sous le nom de BusyBox et dont l'interêt est sa très petite taille. Elle est souvent utilisée pour construire des applications embarquées avec des ressources limitées (voir `https://en.wikipedia.org/wiki/BusyBox`).
Lancer : **`$ docker run busybox`**
Ceci produit-il un affichage ? Pourquoi ?

Q4.2 - Essayer la commande suivante et vérifier que vous avez bien le résultat attendu :
`$ docker run busybox echo "hello from busybox"`

Q4.3 - La commande précédente ne nous donne toujours pas la main. On peut ouvrir un Shell "sh" dans le conteneur avec les options -i ("*interactive*") et -t ("*terminal*").
Lancer un shell dans le conteneur avec : **`$ docker run -it busybox sh`**
Combien y-a-t-il de fichiers dans le dossier **`bin`** ?

Quelle est la commande que vous avez utilisée :

Q4.4 - Quitter le shell du conteneur avec **`$ exit`**

Q4.5 - Essayer de lancer le shell habituel (**`bash`**) dans le conteneur BusyBox. Que se passe-t-il ? Pourquoi ça ne fonctionne pas ?

Q4.6 - Quelle est la commande à utiliser pour afficher la liste des images disponibles :

Q4.7 - Quelle est la commande à utiliser pour supprimer une image **`xxx`** :

Q4.8 - Avec cette dernière commande, essayer de supprimer l'image **`busybox`** Cela fonctionne-t-il ? Pourquoi ?

Q4.9 - Que faut-il faire d'abord ?

Q4.10 - Quelle est la commande à utiliser pour cette dernière opération :

5 Conteneur Ubuntu

Q5.1 - Lancer un conteneur Ubuntu en y démarrant un Shell :

```
$ docker run -it ubuntu /bin/bash
```

Q5.2 - Quelle commande me donne la version du noyau :

Q5.3 - Quelle est la version du noyau :

Q5.4 - Arreter le conteneur avec **`$ exit`**, puis le relancer avec :

```
$ docker run -it ubuntu
```

Est ce que ceci change quelque chose ?

Q5.5 - Pourquoi ? (indice : taper **`echo $SHELL`**)

Q5.6 - Quelle est la commande pour afficher toutes les images disponibles localement :

Q5.7 - Noter la taille des images suivantes :

ubuntu :

busybox :

6 Utilisation de l'API REST

Docker fournit une API REST qui permet d'interagir avec le *daemon* de façon similaire au client mais via des requetes HTTP. Ceci permet une action à distance (si Docker est installé sur un serveur distant), soit via un client HTTP CLI, soit via une application dédiée ou un navigateur.

On va ici faire une petite démonstration de son utilisation.

Q6.1 - Installer deux outils dans votre MV :

```
$ sudo apt install curl jq
```

- **`jq`** est un "processeur JSON" : il fonctionne comme un filtre ("à-la" grep) et peut générer et/ou interpréter du texte en JSON
- **`curl`** est un client HTTP en ligne de commande

Q6.2 - Il est d'abord nécessaire d'éditer un fichier de config de Docker :

```
$ sudo nano /lib/systemd/system/docker.service
```

Modifier la ligne

`ExecStart=/usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock`

en

`ExecStart=/usr/bin/dockerd -H fd:// -H=tcp://0.0.0.0:5555`

puis lancer les commandes suivantes :

```
$␣sudo␣systemctl␣daemon-reload
$␣sudo␣service␣docker␣restart
```

Q6.3 - Une fois le service relancé, tester :

```
$ curl http://localhost:5555/images/json
```

Ceci doit renvoyer toutes les infos sur les images disponibles localement, sous forme de JSON "brut".

Q6.4 - Relancer la commande en la passant dans **jq**, et rediriger la sortie vers un fichier **dockimg**. Ouvrir le fichier avec `$ mousepad dockimg`. Vérifier que le JSON est bien formatté de façon plus lisible.

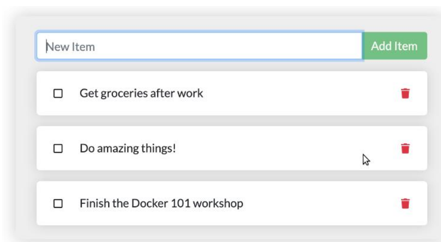
Q6.5 - Chercher sur https://docs.docker.com/engine/api/latest/ et donner la commande utilisant **curl** pour récupérer la liste des conteneurs en cours d'exécution :

Q6.6 - Donner la commande utilisant **curl** pour récupérer la liste de tous les conteneurs (équivalent à **docker ps -a**) :

7 Une application web

(D'après https://docs.docker.com/get-started/02_our_app/)

Dans cette partie, vous aller créer en local une application web permettant de créer des "todo" list.



Dans un premier temps, on va juste faire tourner l'application, sans persistance des données. Dans un second temps, on va lui connecter un "volume" docker. Ensuite, on va intégrer une BDD qui tournera dans un autre conteneur, et on verra comment connecter les deux services via un réseau privé Docker.

7.1 Mise en place de l'application web

Q7.1 - Dans la MV, télécharger le fichier **app.zip** sur Universitice et le copier dans **$HOME**. Décompresser dans ce dossier avec : `$ unzip app`

Q7.2 - `$ cd app`, puis créer dans ce dossier un fichier nommé **Dockerfile** et contenant ceci :
(copier/coller depuis la doc.)

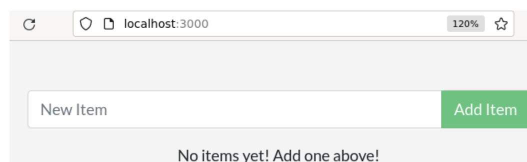
```
FROM node:12-alpine
# Adding build tools to make yarn install work on Apple silicon / arm64 machines
RUN apk add --no-cache python2 g++ make
WORKDIR /app
COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
```

Q7.3 - Construire l'image avec la commande : `$ docker build -t todolist1 .`

Q7.4 - Vérifier qu'elle est bien présente dans la liste des images. Quelle est sa taille ?

Q7.5 - Lancer le conteneur avec `$ docker run -p 3000:3000 todolist1`

Q7.6 - Avec le navigateur, se connecter sur le port 3000. Vous devez obtenir ceci :



Ajouter quelques items dans cette liste.

Q7.7 - Arretez le conteneur avec `$ docker stop <id-conteneur>`. Vérifier dans le navigateur avec F5 que la communication ne passe plus.

Q7.8 - Relancer le conteneur avec `$ docker start <id-conteneur>`, et vérifier dans le navigateur avec F5 que l'app est de nouveau fonctionnelle. Les élements ajoutés sont-ils toujours présents ?

7.2 Modification de l'appli On souhaite modifier le code : on veut remplacer le texte an anglais "No items yet! Add one above!" par "Pas encore d'items!". Il faut recréer une nouvelle image.

Q7.9 - Editer le fichier : `$ mousepad src/static/js/app.js` (lui ajouter préalablement les droits en écriture). Cherchez le texte et faire la modification.

Q7.10 - Créer la nouvelle image avec : `$ docker build -t todolist2 .`

Q7.11 - Lancer le nouveau conteneur :
`$ docker run -p 3000:3000 -d todolist2`
Si vous obtenez une erreur à ce stade, c'est parce que le port 3000 est déjà utilisé par le conteneur précédent. Il faut l'arreter avant de pouvoir démarrer le nouveau.

Q7.12 - A quoi sert le flag `-d` ?

Q7.13 - Vérifier que la nouvelle version a bien le texte en français. Les élements ajoutés sont-ils toujours présents ?

Pourquoi ?

7.3 Ajout d'un volume de stockage Les "volumes" Docker permettent de partager un point de montage du système hôte avec un conteneur.

Dans cette application, les données sont stockées dans une BDD "SqLite", qui stocke tout dans un fichier unique.

Q7.14 - Créer un volume avec : `$ docker volume create todo-db`
Vérifier qu'il a bien été créé avec la commande qui liste tous les volumes : `$ docker volume ls`

Q7.15 - Lancer le conteneur en indiquant l'usage de ce volume ainsi que sa localisation dans le conteneur :
`$ docker run -p 3000:3000 -v todo-db:/etc/todos todolist2`

Q7.16 - Ajouter quelques items dans la todolist, puis arreter le conteneur et le supprimer :
`$ docker stop <id-ou-nom>`
`$ docker rm <id-ou-nom>`
Relancer ensuite un conteneur avec la même commande que ci-dessus, mais ajoutant le flag "-d" pour garder la main. Les items sont-ils conservés ?

Q7.17 - Prendre la main dans le conteneur avec `$ docker exec -it <id-ou-nom> sh`
Vérifier avec "ls" la présence du fichier de BDD. Quelle est sa taille ?

Q7.18 - On va maintenant vérifier qu'on peut partager ce volume de stockage entre conteneurs. On va relancer un deuxième conteneur qui va utiliser le volume de stockage du 1er conteneur. Il faut d'abord noter le nom de ce premier conteneur :
Démarrer le 2e conteneur avec (qui écoutera sur le port 3001, pour éviter les conflits) :
`$ docker run -dp 3001:3000 --volumes-from <nom-conteneur-1> todolist2`
Sur `localhost:3001`, retrouvez-vous les élements ajoutés dans le 1er conteneur ?

Q7.19 - Le fichier contenant les données est évidemment physiquement sur la machine hôte. Sa localisation est donnée par `$ docker volume inspect <nom-volume>`

Donner le chemin où se trouve ce volume :
Sur la machine hôte, vérifier que vous retrouvez le fichier à l'endroit indiqué.
Donner sa taille :

7.4 Conteneur de BDD et connexions

Dans cette section, vous aller finaliser la création de l'application en la découpant en deux micro-services :

— D'un coté, l'application "todo list" tournant sur node.js dans un conteneur
— De l'autre coté, la base de donnée mysql, tournant dans un autre conteneur

Les deux parties seront connectées via un réseau virtuel Docker.

Q7.20 - Créer le réseau privé virtuel : `$ docker network create todo-app`

Q7.21 - Vérifier qu'il apparait bien avec la commande `$ docker network ls`

Q7.22 - Quelle est la commande pour obtenir les détails sur ce réseau :

Quel est l'adresse IPv4 et masque du réseau crée :

Q7.23 - Démarrer un conteneur MySQL et le connecter au réseau précédent :

```
docker run -d \
    --network todo-app --network-alias mysql \
    -v todo-mysql-data:/var/lib/mysql \
    -e MYSQL_ROOT_PASSWORD=secret \
    -e MYSQL_DATABASE=todos \
    mysql:5.7
```

On indique ici des variables d'environnement, nécessaires pour mySql

Q7.24 - A quoi sert l'option **-v** ?
(voir `https://docs.docker.com/engine/reference/commandline/run/`)

Q7.25 - A quoi sert l'option **-e** ? :

Q7.26 - Vérifiez que vous avez bien la connection avec la BDD et que vous pouvez interagir avec, via :
`docker exec -it <mysql-container-id> mysql -p`
Au prompt, donner le mot de passe (indiqué dans la commande précédente).

Q7.27 - Une fois le shell mysql ouvert, listez les bases disponibles avec la commande :
`mysql> SHOW DATABASES;` → Vous devez voir la table "todos"

Q7.28 - En vérifiant bien que vous êtes dans le dossier **app/** de la MV, lancer un conteneur basé sur Alpine [1] et qui va lancer l'appli qui se trouve dans ce dossier, via la connection à un volume sur ce dossier :

```
docker run -dp 3000:3000 \
   -w /app -v "$(pwd):/app" \
   --network todo-app \
   -e MYSQL_HOST=mysql \
   -e MYSQL_USER=root \
   -e MYSQL_PASSWORD=secret \
   -e MYSQL_DB=todos \
   node:12-alpine \
   sh -c "yarn install && yarn run dev"
```

Q7.29 - Vérifier le fonctionnement de l'appli avec le navigateur sur le port 3000.

1. `https://en.wikipedia.org/wiki/Alpine_Linux`