Travaux pratiques 01 - Docker

Individuel - Rendu non attendu

Durant ces travaux pratiques, l'objectif est de se familiariser avec Docker afin d'en tirer profit pour la suite du cours mais aussi de comprendre cette technologie largement utilisée dans le contexte professionnel concernant tout le cycle de vie des applications, du développement à la mise en production en passant par les tests.

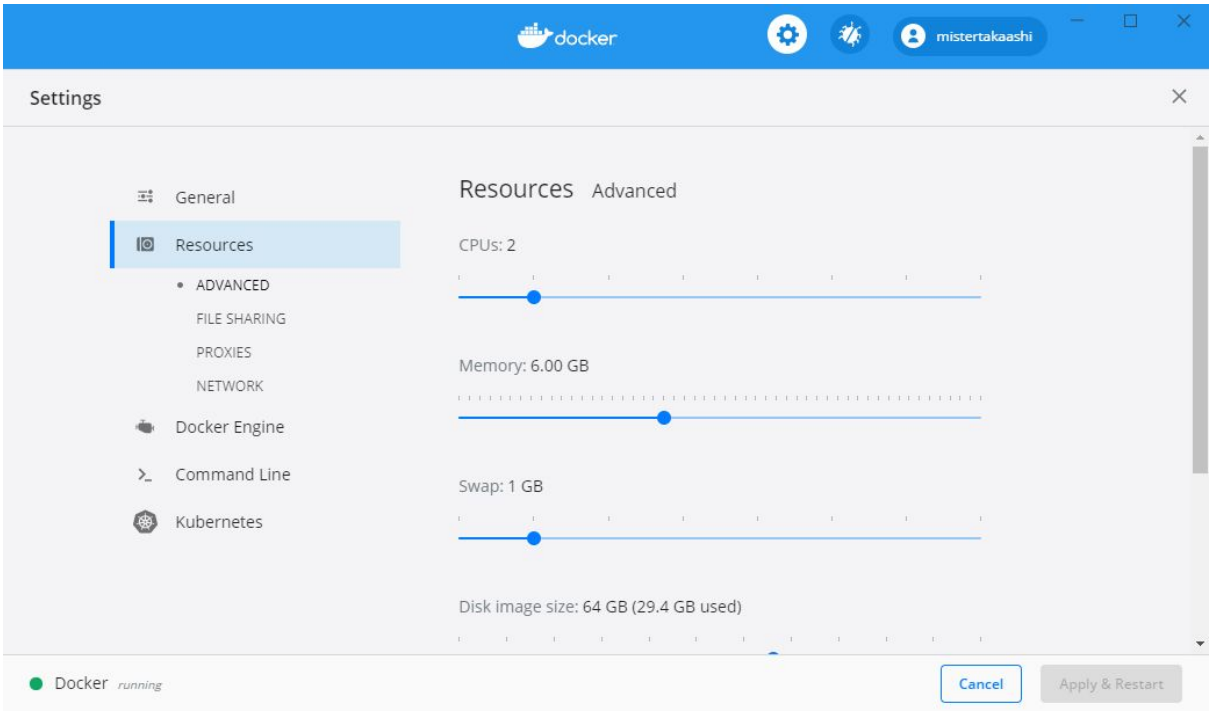
Attention, les commandes de ce TP contiennent pour la plupart sudo qui est utilisé dans le cadre de linux, si vous utilisez Windows, sudo n'est évidemment pas à ajouter.

Mise en place de Docker

La première étape est d'installer Docker selon votre plateforme (Windows, Linux ou Mac). Les versions Windows et Mac disposent d'une interface graphique appelée Docker Desktop, ce n'est pas le cas pour linux.

Si vous êtes sur Windows ou Mac, la première étape est de valider les paramètres de Docker. Lancez Docker Desktop qui devrait se place dans le System Tray de l'OS.

- Dans Resources > Advanced
    - Choisissez les ressources que vous souhaitez allouer à la machine virtuelle Linux qui vous permet d'émuler Docker.



- Dans Resources > File Sharing

- Assurez vous bien d'autoriser votre système d'exploitation à accéder aux disques durs sur lesquels pour travaillerez

Si tout va bien, Docker doit être en train de tourner: Sur windows et mac:

Sur Linux, si le deamon n'est pas lancé, vous pouvez le lancer avec systemcl

```
# Starts the docker deamon
sudo systemctl start docker

# Check the docker deamon status
sudo systemctl status docker
● docker.service - Docker Application Container Engine
    Loaded: loaded (/usr/lib/systemd/system/docker.service; disabled;
vendor preset: disabled)
  Active: active (running) since Wed 2020-02-19 15:43:15 CET; 3s ago
    Docs: https://docs.docker.com
Main PID: 2431 (dockerd)
   Tasks: 28 (limit: 4915)
  Memory: 183.8M
  CGroup: /system.slice/docker.service
          ├─2431 /usr/bin/dockerd -H fd://
                                  └─2440    containerd    --config
/var/run/docker/containerd/containerd.toml --log-level info
```

Pour vérifier que Docker est bien en état de fonctionnement, lancez le container hello-world

```
# Launching the hello-world image pulled from Docker Hub
sudo docker run hello-world

Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest:
sha256:9572f7cdcee8591948c2963463447a53466950b3fc15a247fcad1917ca215a2f
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/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

Le run s'effectue en deux étapes, le **pull** et le **run** réel. En premier lieu, Docker va vérifier que l'image que vous venez de demander (ici hello-world) existe avec le tag demandé en local sur la machine (non précisé ici, donc *latest*). Si cette image n'existe pas, elle sera tirée depuis le repository Docker Hub.

Il est possible de tirer manuellement l'image via:

```
# Pulling the hello-world image from Docker Hub
sudo docker pull hello-world
```

Précisions ici les différentes notions de Docker:

- Image: La configuration et le système de fichier utilisé pour créer le container.
- Container: Instance d'une image
- Deamon Docker: Le service tournant sur l'hôte permettant la création et l'interaction avec les containers

Utiliser une image du Docker Hub

Il nous est maintenant possible d'utiliser toutes les images déjà créées par la communauté à partir du Hub de Docker.

Essayons de tirer une image de Alpine Linux, une distribution extrêmement légère utilisée comme base pour de nombreuses autres images.

```
# Pull Alpine image from Docker Hub
sudo docker pull alpine:latest
```

Remarquons dans ce pull que le tag désiré a été spécifié, il s'agit de *latest*. Sur la [page de Docker Hub de Alpine], il est possible de consulter les tags disponibles.

Alpine est maintenant disponible pour un run sur notre machine. On peut lister les images via la commande:

```
# List all images locally available
sudo docker images
```

Elle devrait indiquer:

```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
alpine              latest              e7d92cdc71fe        4 weeks ago         5.59MB
hello-world         latest              fce289e99eb9        13 months ago       1.84kB
```

Lançons maintenant l'image de alpine sur notre deamon Docker. Une image docker représente un contexte d'exécution, il est donc possible de lancer une commande à l'intérieur de ce contexte. Listons les fichiers de Alpine Linux:

```
# Run the ls command inside a container with alpine linux context
sudo docker run alpine ls -l
total 56
drwxr-xr-x    2 root     root          4096 Jan 16 21:52 bin
drwxr-xr-x    5 root     root           340 Feb 19 15:23 dev
drwxr-xr-x    1 root     root          4096 Feb 19 15:23 etc
drwxr-xr-x    2 root     root          4096 Jan 16 21:52 home
drwxr-xr-x    5 root     root          4096 Jan 16 21:52 lib
drwxr-xr-x    5 root     root          4096 Jan 16 21:52 media
drwxr-xr-x    2 root     root          4096 Jan 16 21:52 mnt
drwxr-xr-x    2 root     root          4096 Jan 16 21:52 opt
dr-xr-xr-x  124 root     root             0 Feb 19 15:23 proc
drwx------    2 root     root          4096 Jan 16 21:52 root
drwxr-xr-x    2 root     root          4096 Jan 16 21:52 run
drwxr-xr-x    2 root     root          4096 Jan 16 21:52 sbin
drwxr-xr-x    2 root     root          4096 Jan 16 21:52 srv
dr-xr-xr-x   13 root     root             0 Feb 19 15:23 sys
```

```
drwxrwxrwt    2 root     root          4096 Jan 16 21:52 tmp
drwxr-xr-x    7 root     root          4096 Jan 16 21:52 usr
drwxr-xr-x   12 root     root          4096 Jan 16 21:52 var
```

La commande run a ici lancé un container, c'est à dire une instance de l'image Alpine demandée et dans ce container la commande **ls -l** a été lancée et docker a terminé son exécution.

Essayons maintenant d'accéder au shell de notre container en lançant **sh**:

```
# Run the sh command inside a container with alpine linux context
sudo docker run alpine /bin/sh
```

Il est important de constater ici que docker nous a instantanément rendu la main, là où on s'attendait à pouvoir interagir avec notre container. Les shell se terminent à la suite du jeu d'instruction qui leur a été demandé d'être exécuté, à moins que ceux-ci soient lancés à partir d'un terminal interactif. On peut lancer un terminal interactif à partir de Docker avec l'option: **-it**, comme:

```
# Run an interactive sh command inside a container with alpine linux
context
sudo docker run -it alpine /bin/sh
```

Via cette commande, on se situe dans le contexte de notre container, on peut donc exécuter n'importe quelle commande disponible dans ce contexte. Il est important de comprendre l'importance du contexte sur Docker, vous pouvez exécuter ici les commandes habituelles sur Linux. Vous pouvez sortir de ce shell interactif avec CTRL + A + D ou la commande **exit**.

Listons maintenant les différents containers tournant sur notre machine avec:

```
# List all running containers on host
sudo docker ps
CONTAINER ID         IMAGE                COMMAND                CREATED
STATUS              PORTS               NAMES
```

Alors qu'on vient de lancer un container sur notre hôte, il n'apparaît pas dans la liste rendue par la commande **ps**, c'est tout à fait normal, il est à noter que **docker ps** ne renvoie que les containers dans l'état "running" ce qu n'est plus le cas de notre container puisqu'on vient de quitter son shell interactif, il s'est donc arrêté. On peut lister tous les containers via l'argument **-a**:

```
# List all containers on host
sudo docker ps -a
```

```
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS                      PORTS               NAMES
60695ce21683        alpine              "/bin/sh"           13 seconds
ago      Exited (0) 9 seconds ago                        silly_robinson
```

On constate bien la présence de notre container dans son état “Exited”.

Lancer et accéder à une application Web

Essayons maintenant d’utiliser Docker pour mettre en place une mini infrastructure Web et accéder à un site internet hébergé par sur un container docker.

Nous allons utiliser l’image dockersamples/static-site directement disponible sur le Docker Hub. Lançons l’image:

```
# Run the static-site image from Docker hub
sudo docker run dockersamples/static-site

# Run the static-site image from Docker hub with detached option
sudo docker run -d dockersamples/static-site
```

Maintenant, comment accéder à notre site depuis notre machine hôte ? Il va falloir rajouter quelques paramètres à notre run, c’est ici que va intervenir la notion de **port binding**.
Assurez vous d’avoir lancé le container en mode détaché via l’option **-d** et faites un **docker ps** dont voici le résultat.

Il nous faut donc arrêter ce container pour le relancer avec les bons arguments, pour arrêter un container détaché, il n’est évidemment pas possible de taper exit ou bien d’effectuer un signal SIGTERM via CTRL + C.
Exécutons un **docker ps**:

```
sudo docker ps
CONTAINER ID            IMAGE                                   COMMAND
CREATED             STATUS              PORTS               NAMES
0123226dc8d8        dockersamples/static-site    "/bin/sh -c 'cd /usr…"
6 seconds ago             Up 5 seconds                80/tcp, 443/tcp
reverent_carson
```

Constatons ici trois informations importantes:

- Chaque container a un UID qui lui est attribué par le deamon Docker à son lancement, il en va de même pour toutes les ressources créées par Docker.
- Chaque container à un nom, vous pouvez lui donner le nom que vous souhaitez avec l’argument **--name**, le nom est un alias de l’UID, les commandes que nous allons exécuter avec l’UID peuvent être lancées avec le nom du container.

- Les ports exposés par le container sont mentionnés sur ce container, on constate que l'image **dockersamples/static-site** expose deux ports TCP: 80 (HTTP), et 443 (HTTPS). On peut aussi obtenir ces informations en inspectant l'image utilisée **sudo docker inspect dockersamples/static-site**.

On peut maintenant arrêter et supprimer le container:

```
# Stops the container with UID 0123226dc8d8
sudo docker stop 0123226dc8d8

# Removes the container with UID 0123226dc8d8
sudo docker rm 0123226dc8d8
```

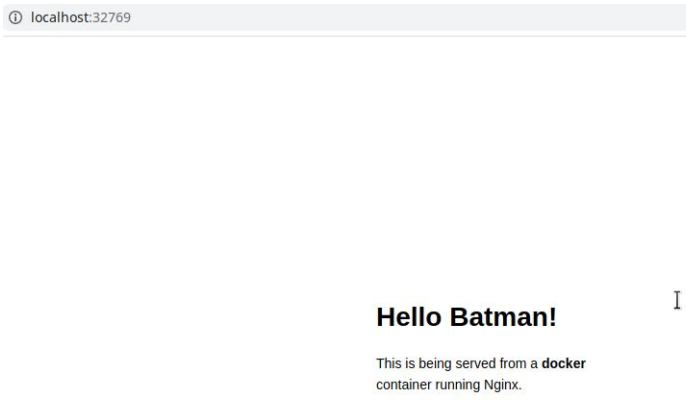
Relançons notre image avec les arguments nécessaires à son bon fonctionnement:

```
# Run the static-site image from Docker hub
# --name: Set the container name
# -d: Detached mode
# -P: Publish all exposed port to host with random ports
# -e: Add environment variable in the container context
sudo docker run --name static-site -d -P -e AUTHOR="Batman"
dockersamples/static-site
```

Si on exécute un **docker ps**, on verra les binding de port que Docker a effectué pour nous:

```
# List all running containers on host
sudo docker ps
CONTAINER ID            IMAGE                                COMMAND
CREATED                         STATUS                           PORTS
NAMES
53b5cd2844c3        dockersamples/static-site    "/bin/sh -c 'cd /usr…"
5 seconds ago           Up 3 seconds            0.0.0.0:32769->80/tcp,
0.0.0.0:32768->443/tcp   static-site
```

On constate ici que le port 80 est servi sur mon hôte via le port 32769, on peut donc sur un navigateur se rendre sur http://localhost:<votre port>:



Cette publication automatique des ports est fonctionnelle mais peut vite se trouver limiter dans le cas où un tiers à besoin d'avoir un port fixe pour un container, heureusement il est aussi possible de binder les ports manuellement. Arrêtons et supprimons ce container:

```
# Stops the static-site container
sudo docker stop static-site

# Removes the static-site container
sudo docker rm static-site
```

On peut utiliser l'argument **-p** pour assigner les binding de port:

```
# Run the static-site image from Docker hub
# -p: Binds the host port with the container port > host:container
sudo docker run --name static-site -e AUTHOR="Batman" -d -p 8888:80
dockersamples/static-site
```

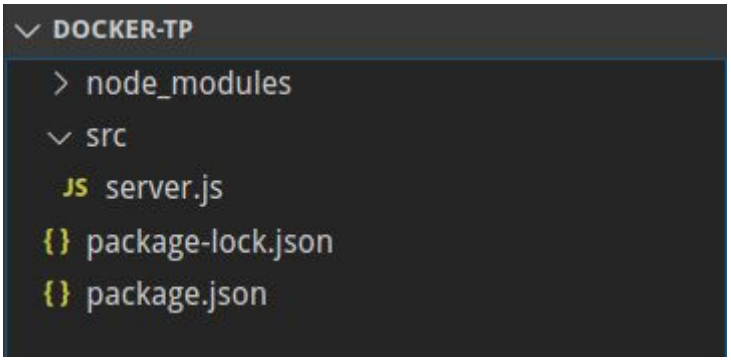
```
# List all running containers
sudo docker ps
CONTAINER ID            IMAGE                                COMMAND
CREATED                         STATUS                          PORTS
NAMES
4dc5544499b1        dockersamples/static-site    "/bin/sh -c 'cd /usr…"
6 seconds ago       Up 5 seconds         443/tcp, 0.0.0.0:8888->80/tcp
static-site
```

Notre container expose donc maintenant son port 80 sur notre port 8888.

Créer une image Docker

Commencez par créer un petit serveur Express en Node.JS exposant une seule route '/' avec un simple message, en lançant un **npm start** votre serveur devrait être accessible sur le port 3000 et afficher un message. Notre objectif va être de "Dockerizer" votre projet Node.JS.

Votre projet devrait ressembler à ça:



Nous allons maintenant écrire le **Dockerfile** de votre projet. Ce fichier indique les étapes de création de l'image de votre projet. Il s'agit simplement d'un fichier contenant les commandes que le deamon Docker exécutera à la création de votre image.

Commençons par choisir de quelle image notre Dockerfile va dériver, ici nous choisirons alpine dans sa version 3.11

```
# The image of which our drift like Takumi
FROM alpine:3.11
```

On va ensuite lancer la commande **apk add** dans le container pour installer nodejs et npm

```
# Install nodejs and npm
RUN apk add --update nodejs npm
```

On va ensuite se positionner dans le répertoire où sera stockée notre application avec WORKDIR

```
# We set the workdir
WORKDIR /usr/src/app/
```

On ajoute notre fichier package.json au container et on lance l'installation des modules

```
# Install node module needed by the app
ADD package.json /usr/src/app/package.json
RUN npm install
```

On ajoute le reste de notre application dans le répertoire de travail

```
# Add the application files
ADD src/server.js /usr/src/app/src/
```

On précise à l'image quel port sera exposé par l'application

```
# Expose the application port
EXPOSE 3000
```

Et on précise quelle commande sera lancée par défaut lorsqu'un utilisateur de l'image la lancera

```
# Run the application
CMD ["npm", "start"]
```

Il est à noter la différence entre **RUN** et **CMD**, il ne peut y avoir qu'un seul **CMD** par Dockerfile, et cette commande n'est pas exécutée à la création de l'image.

Votre Dockerfile devrait ressembler à

```
# The image of which our Dockerfile drifts (like Takumi)
FROM alpine:3.11

# Install nodejs and npm
RUN apk add --update nodejs npm

# We set the workdir
WORKDIR /usr/src/app/

# Install node module needed by the app
ADD package.json /usr/src/app/package.json
RUN npm install

# Add the application files
ADD src/server.js /usr/src/app/src/

# Expose the application port
EXPOSE 3000

# Run the application
CMD ["npm", "start"]
```

Nous pouvons maintenant demander à Docker de construire notre image avec la commande:

```
# Build our image and tag it with docker-tp name
sudo docker build . -t docker-tp
```

Une fois construite, on peut la lancer via:

```
# Run the local docker-tp image with port binding
sudo docker run -p 8080:3000 docker-tp
```

Vous pouvez accéder à votre application via le port 8080 sur votre machine.

Utiliser docker-compose

Il est évident que lorsqu'un projet implique plus d'un ou deux container à gérer ou inclu des paramètres réseaux, volumes, etc, utiliser docker run pour chaque container peut être fastidieux. Docker-compose va permettre de gérer le cycle de vie d'un ensemble de containers, docker-compose va en réalité effectuer les différentes commandes run, stop et rm pour chaque container, volume et networks.

Les containers sont spécifiés par un fichier docker-compose.yaml regroupant les différents containers. Voici le fichier de configuration pour notre application faite au dessus:

```
version: '3'
services:
 web:
   build: .
   ports:
     - "8080:3000"
 redis:
   image: "redis:alpine"
```

On peut lancer ce fichier avec la commande

```
# Run docker-compose stack
sudo docker-compose up

# Stop docker-compose stack
sudo docker-compose down
```

Toutes les options disponibles dans la commande "docker run" sont disponibles via le docker-compose, on peut donc évidemment spécifier ports, volumes, tty, ressources, etc...

Gestion des volumes

Les dernier point à noter est la gestion des données dans les containers, il s'agit d'une partie importante à comprendre de Docker. Etant donné le cycle de vie des containers, lorsqu'un fichier est créé dans un container, il est propre à ce container et ne peut être accédé à partir de l'hôte puisqu'il est encapsulé dans un volume créé par Docker.

Le problème qui se pose est donc le suivant: Comment partager des dossiers entre la machine hôte et un container docker alors que le container porte son propre système de fichier et se veut hermétique à l'hôte. La réponse tient dans la création de volume partagés entre Docker et l'hôte.

Essayons de créer un dossier de log custom, l'objectif étant que lorsque que quelqu'un se connecte à notre application en NodeJS, on log sa connexion dans le fichier logs/access.log

On peut maintenant lancer l'image docker de notre application puisque nous avons changé les sources.

```
# Run the local docker-tp image
#  -v: Add a volume to the container, bind ./logs on the host to
/usr/src/app/logs
sudo docker run -p 8080:3000 -v $(pwd)/logs:/usr/src/app/logs docker-tp
```

Maintenant, on peut voir sur notre hôte le fichier access.log qui est modifié en temps réel par le container.

Vous savez maintenant tout (ou presque) de docker et de l'orchestration de container, mis à part le networking, les stacks, swarm, kubernetes, le milliers d'argument de docker, etc...