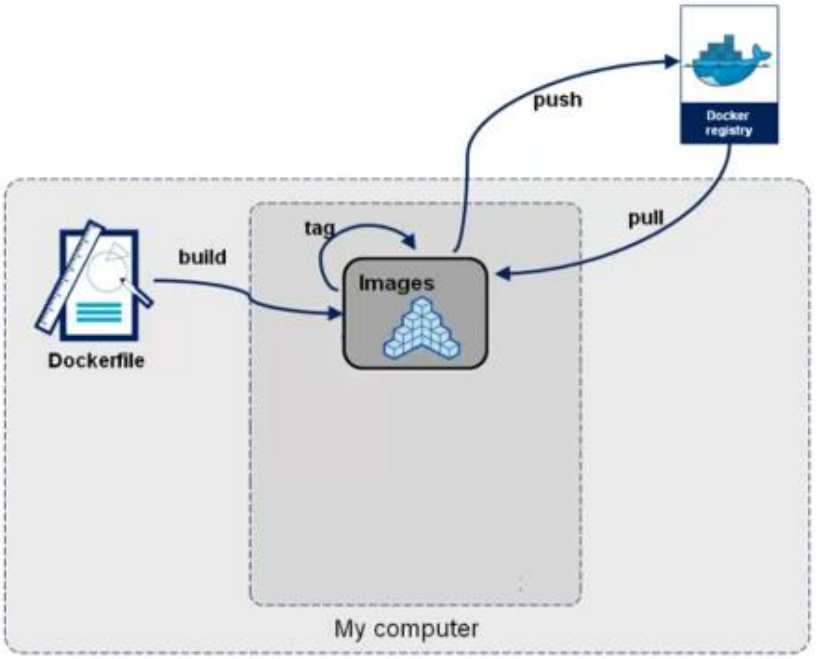
Les DockerFiles : Builder ses images

1. Fonctionnement

Le principe est qu'à partir d'un simple fichier texte, nous allons construire nos images (on appelle ça builder) en y insérant des instructions, des clés et des valeurs afin de déployer ces images.

Les mot clés seront FROM, RUN, ADD, ENTRYPOINT, …

```
FROM debian
RUN apt-get .
ADD file
ENTRYPOINT ..
```



Le FROM indiquera l'image de base par laquelle nous allons construire notre nouvelle image.
La commande RUN permettra d'exécuter des commandes dans cette image, souvent des installations ou des configurations
Pour ces configurations, on peut aussi pourquoi pas ajouter des fichiers à l'aide de la commande ADD
ENTRYPOINT permettra de lancer des commandes au moment de l'instanciation de notre conteneur
La construction de notre image s'effectuera à l'aide de la commande `docker build`
Le fichier contenant ces instructions se nommera `Dockerfile`

FROM : Définit l'image de base qui sera utilisée par les instructions suivantes.
LABEL : Ajoute des métadonnées à l'image avec un système de clés-valeurs, permet par exemple d'indiquer à l'utilisateur l'auteur du Dockerfile.
ARG : Variables temporaires qu'on peut utiliser dans un Dockerfile.
ENV : Variables d'environnements utilisables dans votre Dockerfile et conteneur.
RUN : Exécute des commandes Linux ou Windows lors de la création de l'image. Chaque instruction RUN va créer une couche en cache qui sera réutilisée dans le cas de modification ultérieure du Dockerfile.
COPY : Permet de copier des fichiers depuis notre machine locale vers le conteneur Docker.
ADD : Même chose que COPY mais prend en charge des liens ou des archives (si le format est reconnu, alors il sera décompressé à la volée).
ENTRYPOINT : comme son nom l'indique, c'est le point d'entrée de votre conteneur, en d'autres termes, c'est la commande qui sera toujours exécutée au démarrage du conteneur. Il prend la forme de tableau JSON (ex : CMD ["cmd1","cmd1"]) ou de texte.
CMD : Spécifie les arguments qui seront envoyés au ENTRYPOINT, (on peut aussi l'utiliser pour lancer des commandes par défaut lors du démarrage d'un conteneur). Si il est utilisé pour fournir des arguments par défaut pour l'instruction ENTRYPOINT, alors les instructions CMD et ENTRYPOINT doivent être spécifiées au format de tableau JSON.
WORKDIR : Définit le répertoire de travail qui sera utilisé pour le lancement des commandes CMD et/ou ENTRYPOINT et ça sera aussi le dossier courant lors du démarrage du conteneur.
EXPOSE : Expose un port.
VOLUMES : Crée un point de montage qui permettra de persister les données.
USER : Désigne quel est l'utilisateur qui lancera les prochaines instructions RUN, CMD ou ENTRYPOINT (par défaut c'est l'utilisateur root).

Exemple à commenter :

```
FROM centos
MAINTAINER admin admin@centos.local
RUN yum install -y net-tools
RUN echo "<H1> Bienvenue sur la formation Docker ESIEA</H1>" >
/usr/share/httpd/noindex/index.html
EXPOSE 80
CMD ["-D", "FOREGROUND"]
ENTRYPOINT ["/usr/bin/httpd"]
```

L'image précédente sera construite en appelant la commande suivante :

```
docker build –tag=httpd_esiea .
```

- Vérifiez le Dockerfile précédent en la buildant
- Vérifiez la création de votre image
- Créez le conteneur à partir de cette image tournant sur le port 8010
- Visualisez le résultat dans votre navigateur

TP : Enoncé

Création d'une image à partir d'un Dockerfile

1. Créez un fichier Dockerfile avec les instructions suivantes

- image de base: ubuntu:16.04

- Installation de l'utilitaire ping. Assurez-vous de mettre à jour la liste des dépots (apt-get update) et également d'utiliser l'option -y lors de l'installation du package iputils-ping (apt-get install -y iputils-ping)

- instruction ENTRYPOINT définie comme étant la commande ping

- instruction CMD définie par l'adresse IP 8.8.8.8 correspondant à un serveur DNS de Google

```
FROM ubuntu:16.04
RUN apt-get update -y && apt-get install -y iputils-ping
ENTRYPOINT ["ping"]
CMD ["8.8.8.8"]
```

2. A partir de ce Dockerfile, créez l'image ping:1.0

```
docker image build -t ping:1.0 .
```

3. Lancez un container basé sur l'image ping:1.0 sans lui donner de commande

Qu'observez-vous ?

Le processus lancé dans le container correspond à la concaténation des instructions ENTRYPOINT et CMD définies dans le Dockerfile, à savoir ping 8.8.8.8

4. Lancez un container basé sur l'image ping:1.0 en lui spécifiant la commande www.google.com

Que s'est-il passé ?
Comme un argument (www.google.com) a été spécifié à la suite du nom de l'image (ping:1.0), celui-ci à redéfini l'instruction CMD spécifiée dans le Dockerfile. Le processus lancé dans le container correspond à la concaténation de l'instruction ENTRYPOINT (celle du Dockerfile) et du paramètre spécifié sur la ligne de commande, c'est à dire à ping www.google.com

2. Stockage d'une image

Dans le TP précédent, nous avons créé une image nommée ping:1.0, nous allons voir ici ou cette image est stockée.

Reprenons le Dockerfile de l'exercice :

A partir de ce Dockerfile, l'image est buildée avec la commande suivante :

```
$ docker image build -t ping:1.0 .
Sending build context to Docker daemon  2.048kB
Step 1/4 : FROM ubuntu:16.04
 ---> 5e8b97a2a082
Step 2/4 : RUN apt-get update -y && apt-get install -y iputils-ping
 ---> Using cache
 ---> 4cd5304ad0fb
Step 3/4 : ENTRYPOINT ["ping"]
 ---> Using cache
 ---> d2846bbd30e8
Step 4/4 : CMD ["8.8.8.8"]
 ---> Using cache
 ---> 00a905f2bd5a
Successfully built 00a905f2bd5a
Successfully tagged ping:1.0
```

Pour lister les images présentes localement on utilise la commande "docker image ls" (on reverra cette commande un peu plus loin). Pour ne lister que les images qui ont le nom "ping" on le précise à la suite de "ls".

```
$ docker image ls ping

REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ping                1.0                 00a905f2bd5a        4 weeks ago         159MB
```

Notre image est constituée d'un ensemble de layers, il faut voir chaque layer comme un morceau de système de fichiers. L'ID de l'image (dans sa version courte) est 00a905f2bd5a, nous allons voir à partir de cette identifiant comment l'image est stockée sur la machine hôte (la machine sur laquelle tourne le daemon Docker).

Tout se passe dans le répertoire /var/lib/docker, c'est le répertoire au Docker gère l'ensemble des primitives (containers, images, volumes, networks, ...). Et plus précisément dans /var/lib/docker/image/overlay2/, overlay2 étant le driver en charge du stockage des images.

Note: si vous utilisez "Docker for Mac" ou "Docker for Windows", il est nécessaire d'utiliser la commande suivante pour lancer un shell dans la machine virtuelle dans laquelle tourne le daemon Docker. On pourra ensuite explorer le répertoire /var/lib/docker depuis ce shell.

```
docker run -it --privileged --pid=host debian nsenter -t 1 -m -u -n -i sh
```

Plusieurs fichiers / répertoires ont un nom qui contient l'ID de notre image comme on peut le voir ci-dessous :

```
/var/lib/docker/image/overlay2 # find . | grep 00a905f2bd5a
./imagedb/content/sha256/00a905f2bd5aa3b1c4e28611704717679352a619bcdc4f8f6851cf459dc05816
./imagedb/metadata/sha256/00a905f2bd5aa3b1c4e28611704717679352a619bcdc4f8f6851cf459dc05816
./imagedb/metadata/sha256/00a905f2bd5aa3b1c4e28611704717679352a619bcdc4f8f6851cf459dc05816/lastUpdated
./imagedb/metadata/sha256/00a905f2bd5aa3b1c4e28611704717679352a619bcdc4f8f6851cf459dc05816/parent
```

- Content : le premier fichier contient un ensemble d'information concernant cette image, notamment les paramètres de configuration, l'historique de création (ensemble des commandes qui ont servi à construire le système de fichiers contenu dans l'image), et également l'ensemble des layers qui la constituent. Une grande partie de ces informations peuvent également être retrouvées avec la commande :

```
docker image inspect ping:1.0
```

Parmi ces éléments, on a donc les identifiants de chaque layer :

```
"rootfs": {
    "type": "layers",
    "diff_ids": [
      "sha256:644879075e24394efef8a7dddefbc133aad42002df6223cacf98bd1e3d5ddde2",
      "sha256:d7ff1dc646ba52a02312b535446d6c9b72cd09fda0480524e4828554efb2f748",
```

"sha256:686245e78935e73b737c9a82111c3c7df35f5529d06ce8c2f9a7cd32ec90b456",

"sha256:d73dd9e652956dccbbef716de4b172cc15fff644cc92fc69d221cc3a1cb89a39",

"sha256:2de391e51d731ba02b708038a7f98b7103061b916727bcd165e9ee6402f4cdde",

"sha256:3045bfad4cfefecabc342600d368863445b12ed18188f5f2896c5389b0e84b66"

]

}

Si l'on considère la première layer (celle dont l'ID est 6448...), on voit dans /var/lib/docker/image/overlay2 qu'il y a un répertoire dont le nom correspond à l'ID de cette layer, celui-ci contient plusieurs fichiers :

/var/lib/docker/image/overlay2 # find . | grep '644879075e24394efef8a7dddefbc133aad42'

./layerdb/sha256/644879075e24394efef8a7dddefbc133aad42002df6223cacf98bd1e3d5ddde2

./layerdb/sha256/644879075e24394efef8a7dddefbc133aad42002df6223cacf98bd1e3d5ddde2/size

./layerdb/sha256/644879075e24394efef8a7dddefbc133aad42002df6223cacf98bd1e3d5ddde2/tar-split.json.gz

./layerdb/sha256/644879075e24394efef8a7dddefbc133aad42002df6223cacf98bd1e3d5ddde2/diff

./layerdb/sha256/644879075e24394efef8a7dddefbc133aad42002df6223cacf98bd1e3d5ddde2/cache-id

./distribution/v2metadata-by-diffid/sha256/644879075e24394efef8a7dddefbc133aad42002df6223cacf98bd1e3d

Ceux-ci contiennent différentes informations sur la layer en question. Parmi celles-ci, le fichier cache-id nous donne l'identifiant du cache qui a été généré pour cette layer.

/var/lib/docker/image/overlay2 # cat ./layerdb/sha256/644879075e24394efef8a7dddefbc133aad42002df6223cacf98bd1e3d5ddde2/cache-id

49908d07e177f9b61dc273ec7089efed9223d3798ad1d86c78d4fe953e227668

Le système de fichier construit dans cette layer est alors accessible dans le répertoire :

/var/lib/docker/overlay2/49908d07e177f9b61dc273ec7089efed9223d3798ad1d86c78d4fe953e227668/diff/

- LastUpdated : ce fichier contient la date de dernière mise à jour de l'image

/var/lib/docker/image/overlay2 # cat
./imagedb/metadata/sha256/00a905f2bd5...459dc05816/lastUpdated

2018-07-31T07:32:04.6840553Z

- parent : ce fichier contient l'identifiant du container qui a servi à créer l'image.

/var/lib/docker/image/overlay2 # cat ./imagedb/metadata/sha256/00a905f2bd5459dc05816/parent

sha256:d2846bbd30e811ac8baaf759fc6c4f424c8df2365c42dab34d363869164881ae

On retrouve d'ailleurs ce container dans l'avant dernière étape de création de l'image.

Step 3/4 : ENTRYPOINT ["ping"]

---> Using cache

---> d2846bbd30e8

Le container est celui qui a été commité pour créer l'image finale.

En résumé : il est important de garder en tête qu'une image est constituée de plusieurs layers. Chaque layer est une partie du système de fichiers de l'image finale. C'est le rôle du driver de stockage de stocker ces différentes layers et de construire le système de fichiers de chaque container lancé à partir de cette image.

3. Création d'images

Dans ce lab, nous allons voir comment créer une image à partir d'un container. Même si ce n'est pas l'approche recommandée, il est intéressant de la voir au moins une fois. Ensuite, nous créerons une image avec un Dockerfile et nous étudierons comment l'image est construite pour avoir une meilleure compréhension des briques sous-jacentes.

Création d'une image à partir d'un container
Nous commençons par lancer un shell intéractive dans un container basé sur l'image ubuntu. Nous lui donnons le nom fig.

```
$ docker container run -ti --name fig ubuntu bash
```

Nous installons le package figlet dans ce container.

```
apt-get update -y
apt-get install figlet
```

Une fois le package installé, nous sortons du container.

```
# exit
```

Nous créons une image à partir du container fig lancé précédemment et nous l'appelons myfiglet.

```
$ docker container commit fig myfiglet
```
Nous pouvons voir que l'image que nous venons de créer est lister avec la commande suivante

```
$ docker image ls
```
Nous pouvons maintenant lancer un container à partir de l'image myfiglet et lui spécifier la commande à lancer.

```
$ docker container run myfiglet figlet hello
```
Comme le package figlet est présent dans l'image, la commande retourne le résultat suivant.

```
 _     __
| |__  ___| | | ___
| '_ \/ _ \ | |/ _ \
| | | |  __/ | | (_) |
|_| |_|\___|_|_|\___/
```

Cet exemple montre que l'on peut créer un container y ajouter manuellement des packages et ensuite le commiter pour créer une nouvelle image. Nous pouvons ensuite utiliser cette image pour instancier de nouveaux containers qui contiendront les packages installés. Cette approche n'est pas recommandée et elle n'est pas très portable.

Dans la suite de ce lab, nous allons voir la méthode la plus utilisée pour créer une image. Celle-ci utilise un Dockerfile, fichier texte qui contient toutes les instructions pour construire une image.

Création d'une image à partir d'un Dockerfile
Pour illustrer ceci, nous allons utiliser une application ultra simple construite avec Node.js, qui ne fait qu'écrire un message sur la sortie standard en indiquant le nom du host sur lequel elle est lancée (le nom du container).

Créez un fichier index.js contenant le code suivant.

```
var os = require("os");
var hostname = os.hostname();
console.log("hello from " + hostname);
```

Nous allons Dockerizer cette application en commençant par écrire un Dockerfile. Nous utilisons alpine comme image de base, ajoutons le runtime Node.js et copions ensuite le code source. Nous spécifions également la commande à lancer dans le container.

Créez un fichier Dockerfile contenant les instructions suivantes.

```
FROM alpine
RUN apk update && apk add nodejs
COPY . /app
WORKDIR /app
CMD ["node","index.js"]
```
Nous construisons notre première image à partir de ce Dockerfile et l'appelons hello:1.0

```
$ docker image build -t hello:1.0
```

Nous pouvons ensuite lancer un container à partir de l'image que nous venons de créer.

```
$ docker container run hello:1.0
```
et obtenir un résultat similaire à celui ci-dessous (avec un ID différent)

hello from 92d79b6de29f
Il y toujours plusieurs façons d'écrire un Dockerfile, nous pouvons partir d'une distribution Linux et intaller un runtime (c'est ce que nous avons fait ci-dessus) ou alors utiliser une image de base qui contient déjà le runtime.

Pour illustrer cela, nous créons un nouveau Dockerfile dans le lequel nous spécifions mhart/alpine-node:6.9.4 comme image de base. Ce n'est pas une image officielle mais elle est cependant très connue et utilisée.

Créez un fichier Dockerfile-v2 contenant les instructions suivantes:

```
FROM mhart/alpine-node:6.9.4
COPY . /app
WORKDIR /app
CMD ["node","index.js"]
```
Il n'y a pas de grosses différences avec la première version du Dockerfile, on utilise simplement ici une image de base qui contient déjà le runtime Node.js. Dans cet exemple, ce n'est pas compliqué d'installé Node.js mais pour des environnements plus compliqués il est souvent intéressant d'utiliser une image de base contenant déjà ce que l'on veut.

Nous créons alors une nouvelle image basée sur Dockerfile-v2.

```
$ docker image build -f Dockerfile-v2 -t hello:2.0 .
```
Note: comme le fichier utilisé pour construire l'image s'appelle Dockerfile-v2, et non Dockerfile, nous devons le spécifier avec l'option -f.

Nous pouvons maintenant lancer un container à partir de cette image et obtenir le même résultat que précédemment (avec un autre ID).

```
$ docker container run hello:2.0
```