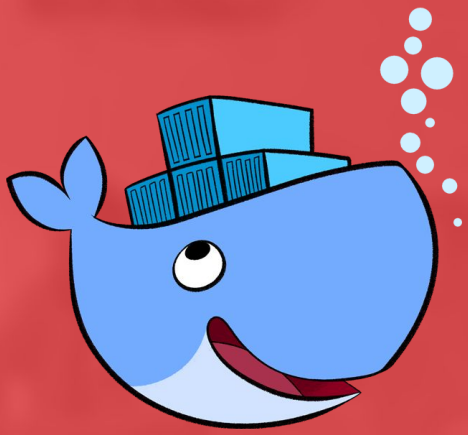
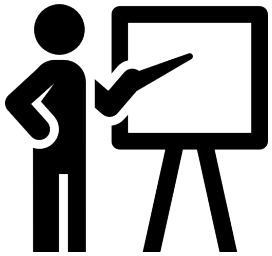


Bienvenue au cours de Docker

Présentation



[sfeir]

Djamel RAAB

Data Engineer

Et vous ?
Prénom
Nom
Job

[E] School

Déroulement de la formation

Feuille de présence (obligatoire)

C’est quand la pause ?
Quand est-ce qu’on mange ?
Tour de table …

Plan de cours

La formation:

- Introduction
- Getting started
- Images et Containers
- Ports
- Volumes
- Dockerfile
- Network

Pour aller plus loin:

- Docker Compose
- Docker Swarm
- Bonus...

Préparation de la formation

Docker - Evaluation Théorique

https://docs.google.com/forms/d/e/1FAIpQLSc2wqTCq6HSUC0mxfKR4-Hxrhz7qKTi_fyhn1vI5URK6Nwtzg/viewform?usp=sharing

Créez un compte sur le Docker Hub

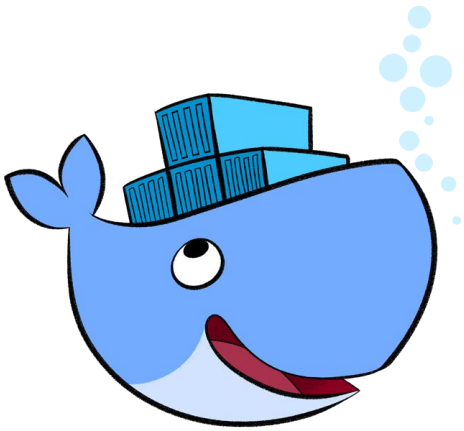
https://hub.docker.com/

```
$ git clone https://github.com/raab-d/ESGI-K8S-5IABD1(5IABD2)
```

```
$ docker run -ti --rm busybox echo "Hello world"
```

Introduction

Qu’est-ce que Docker ?



La logistique



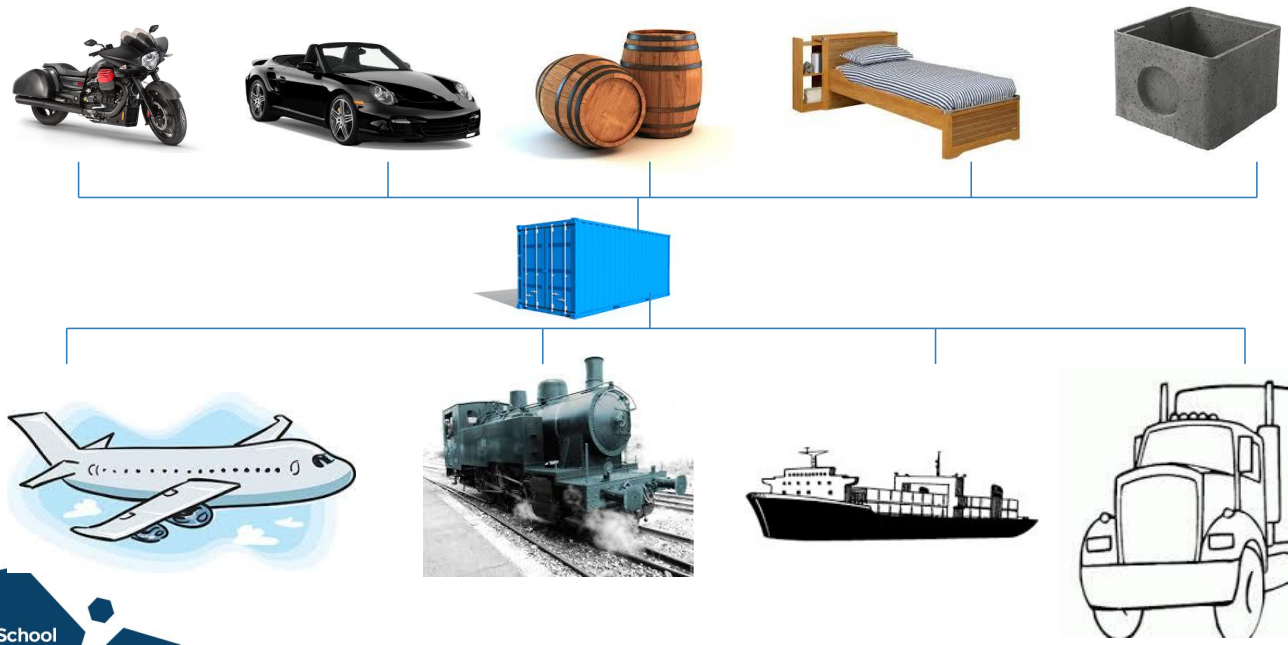






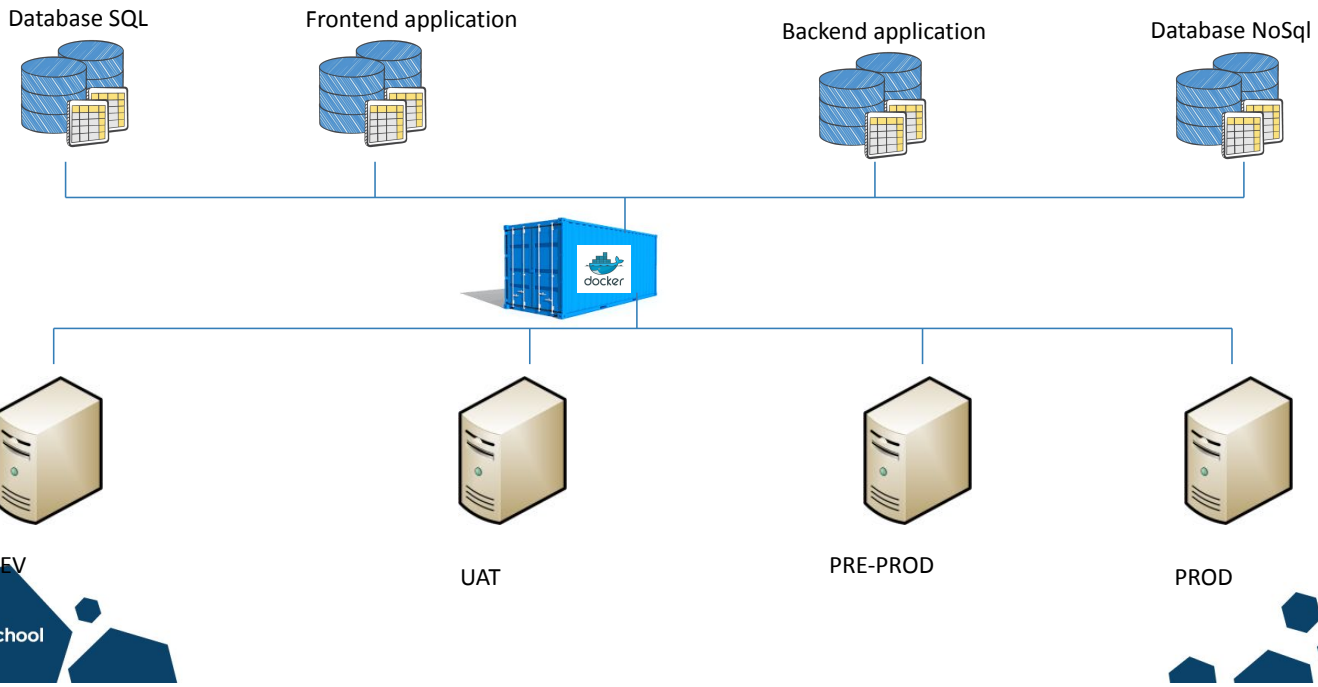

[E] School

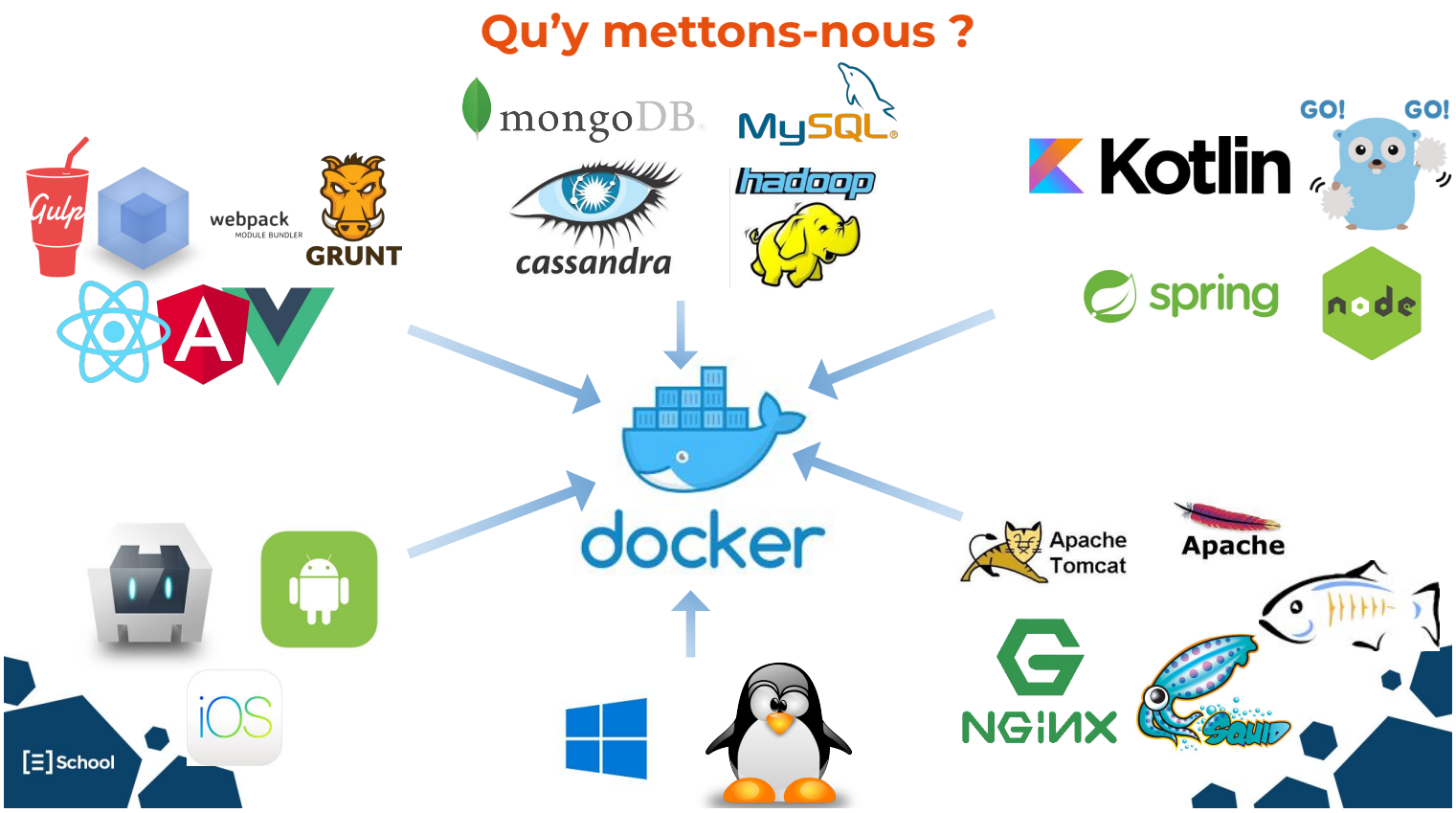
La logistique et les containers



[E] School

L’informatique et les containers




Qu'y mettons-nous ?
mongoDB
MySQL
cassandra
hadoop
Gulp
webpack
MODULE BUNDLER
GRUNT
Kotlin
GO!
GO!
spring
node
docker
Apache Tomcat
Apache
NGINX
SQUID
iOS
School

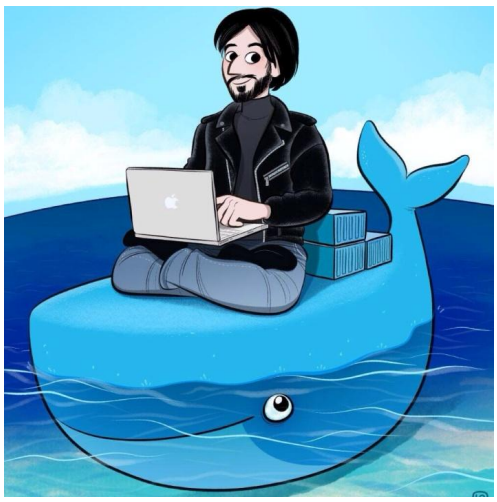
Qu’est-ce que Docker ?

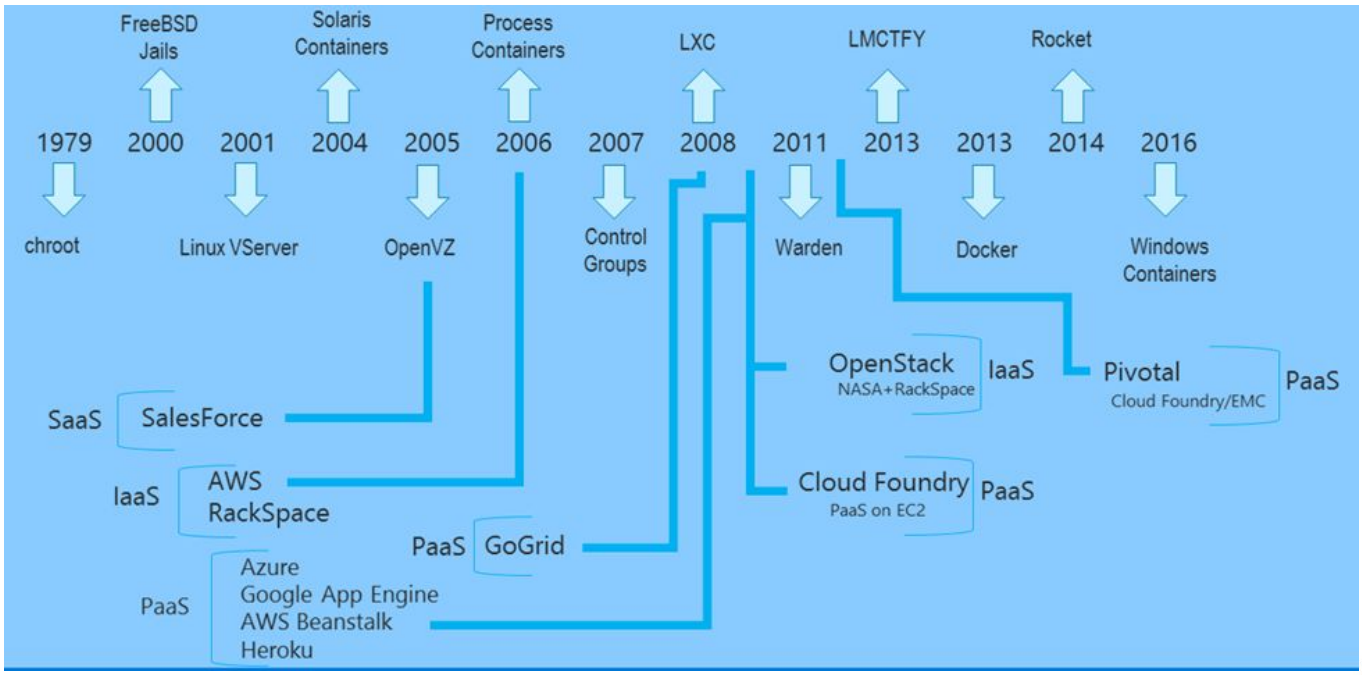
Docker permet de **packager une application**
avec **l’ensemble de ses dépendances**
dans une **unité standardisée** pour le déploiement de logiciels :

les **CONTAINERS**

D’où vient Docker ?

- Solomon Hykes @DotCloud
- Programmé en Go
- Open source
- 1ère version : 13 mars 2013




FreeBSD Jails
Solaris Containers
Process Containers
LXC
LMCTFY
Rocket
1979
2000
2001
2004
2005
2006
2007
2008
2011
2013
2013
2014
2016
chroot
Linux VServer
OpenVZ
Control Groups
Warden
Docker
Windows Containers
SaaS
SalesForce
IaaS
AWS
RackSpace
PaaS
GoGrid
PaaS
Azure
Google App Engine
AWS Beanstalk
Heroku
OpenStack
NASA+RackSpace
IaaS
Pivotal
Cloud Foundry/EMC
PaaS
Cloud Foundry
PaaS on EC2
PaaS

Introduction...

Getting started

Image vs Containers

Classes vs instances

```java
public class Point2D {

    private final int x;

    private final int y;

    public Point2D(int x, int y) {
        this.x = x;
        this.y = y;
    }

}
```

```java
public class Main {

    public static void main(String[] args) {

        Point2D point2D = new Point2D( x: 2,  y: 4);

    }
}
```

Les images

- Listez les images présentes sur votre machine :
  - `> docker image ls`

- Récupérez l'image "busybox" :
  - `> docker image pull busybox`

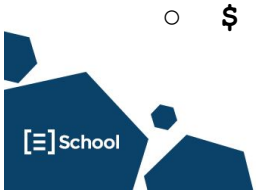
Les Containers

- Listez les containers actifs sur votre machine :
  - `> docker container ls`

- Listez tous les container sur votre machine :
  - `> docker container ls -a`

Premières exécutions

- Instanciez un container “Hello world” à partir de l’image busybox :
  - `> docker container run busybox echo "Hello World"`
- Instanciez un shell interactif à partir de l’image busybox :
  - `> docker container run -i -t busybox`
- Exécutez un “Hello world” dans ce shell puis quittez le container :
  - `$ echo "Hello World"`
  - `$ exit`



Containers éphémères

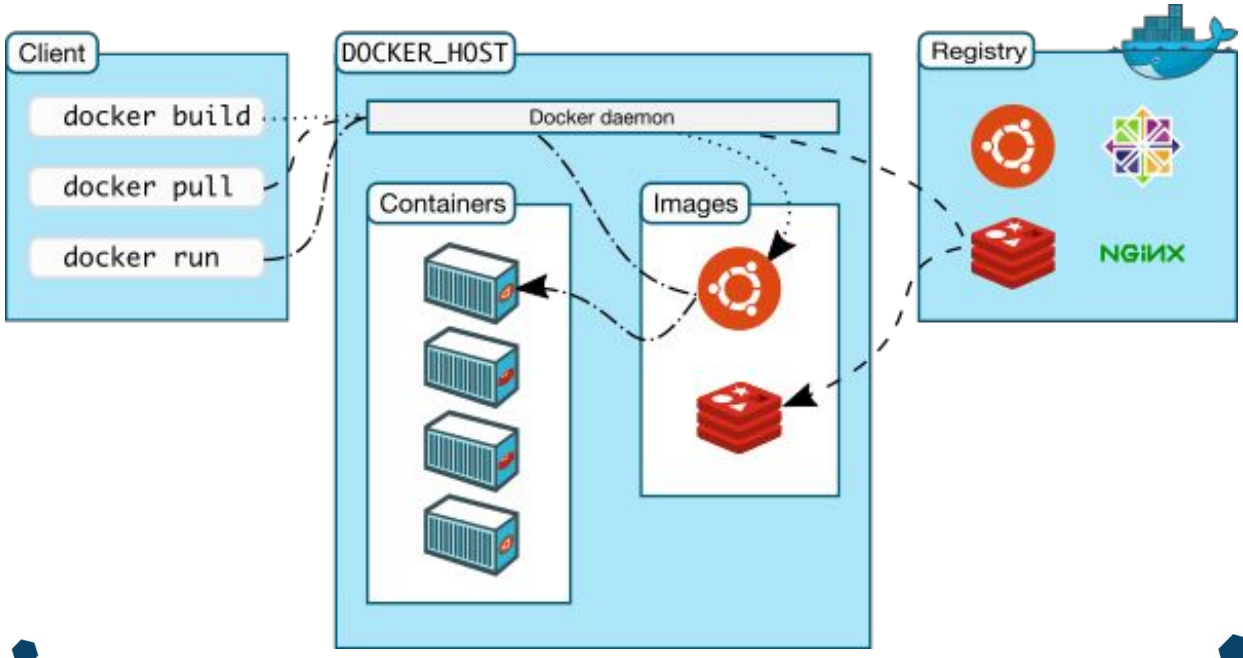
- Instanciez un image busybox interactif, supprimé à l'arrêt (**--rm**) :
  - `> docker container run --rm -it busybox`
- Dans ce container, créez des fichiers, puis quittez :
  - `$ echo "hello" > /docker-test.txt`
  - `$ exit`
- Recréez un container identique et affichez le contenu du fichier :
  - `> docker container run --rm -it busybox`
  - `$ cat /docker-test.txt`

Nettoyage des images

- Listez les images puis supprimez l'image busybox :
  - `> docker image ls`
  - `> docker image rm busybox`

- **Pour info**, pour supprimer toutes les images non utilisées par des containers :
  - `> docker image prune`

Conclusion


Client
docker build
docker pull
docker run
DOCKER_HOST
Docker daemon
Containers
Images
Registry
NGINX

Docker “management commands”

$ docker <OBJECT> <COMMAND>

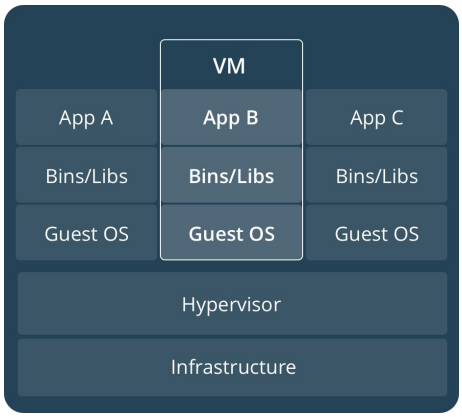
image	ls, pull, rm, prune	Manage images
container	ls, run, stop, rm, prune	Manage containers

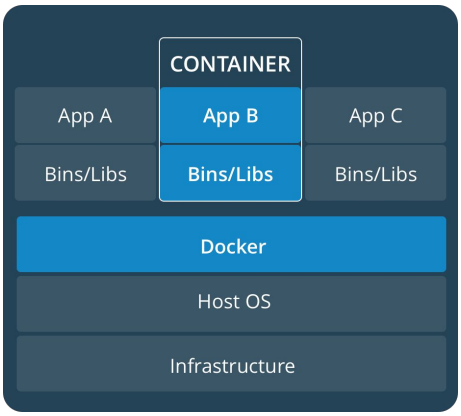
Lab 1 hands on docker

Introduction
Getting Started

Images et Containers

VM vs Containers





VM et Containers

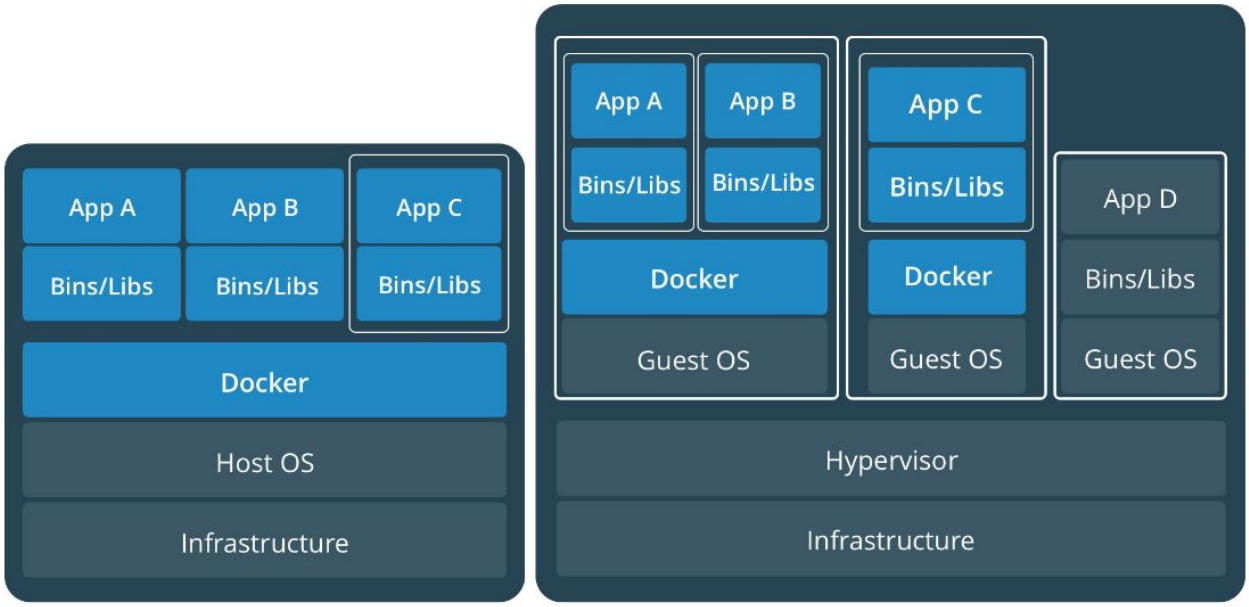
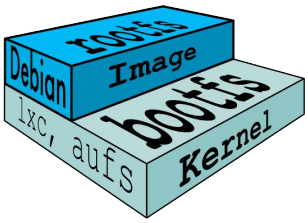
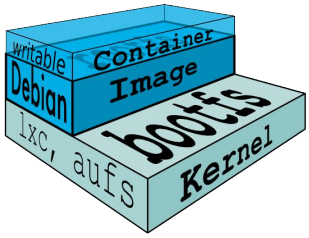


Image layers

- Image en lecture seule → immutabilité
- Copy-On-Write strategy





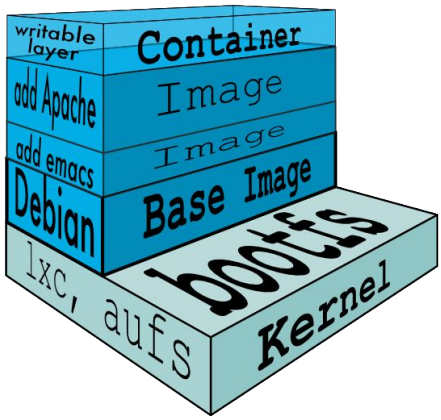
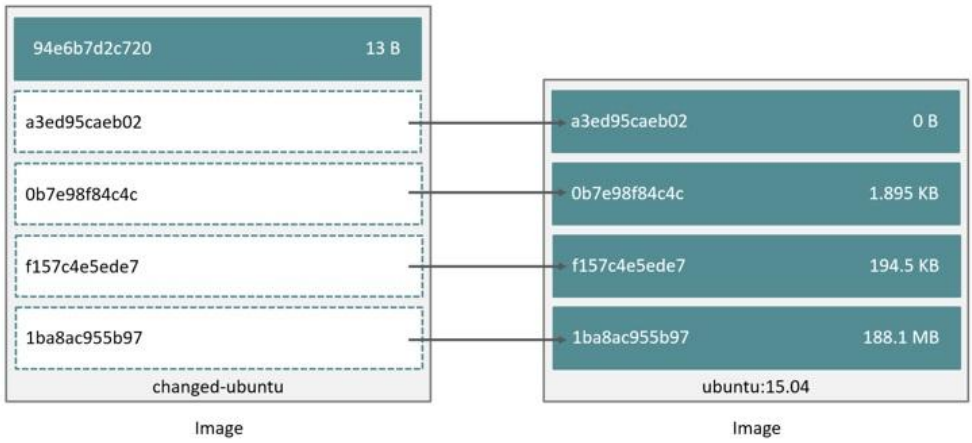


Image layers

- Espace disque
- Performances



Votre première image

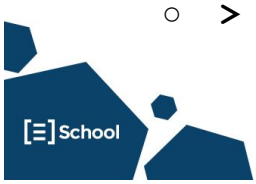
- Récupérez une image alpine et lancez-là en mode interactif
  - `> docker image pull alpine:3.5`
  - `> docker container run -it --name node_ctn alpine:3.5`

- Installez nodejs puis quittez le container :
  - `$ apk add --update nodejs`
  - `$ exit`

- Créez une nouvelle image à partir du container :
  - `> docker container commit node_ctn mynodejs`

Partagez des images

- Taggez votre image :
  - **`> docker image tag`** mynodejs <dockerHubId>**`/nodejs:1.0`**
- Listez vos images locales. Que constatez-vous ?
  - **`> docker image ls`**
- Publiez votre image vers votre dépôt sur le Hub Docker
  - **`> docker login`**
  - **`> docker image push`** <dockerHubId>**`/nodejs:1.0`**



Images et tags

- Image ID = sha256 du contenu
- Tag :
  - pour identifier facilement une image
  - pour déterminer le dépôt d'origine

`[[`registry url`/]`username`/]`image`[:`tag`|@sha256]`

`hub.docker.com/library/`node`:latest`

`hub.docker.com/`zbbfufu/node:8.0

registry.sfeir.com:9000/taiebm/cordova:5.0-test

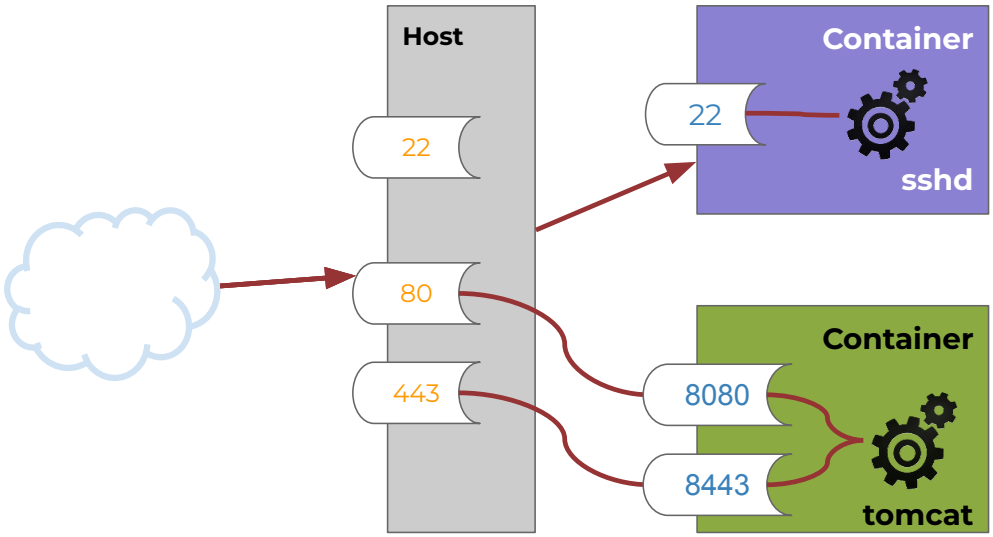
Docker “management commands”

$ docker <OBJECT> <COMMAND>

image	ls, pull, rm, prune, **push, tag**	Manage images
container	ls, run, stop, rm, prune, **commit**	Manage containers


Introduction
Getting Started
Images et Containers
Ports

Port binding

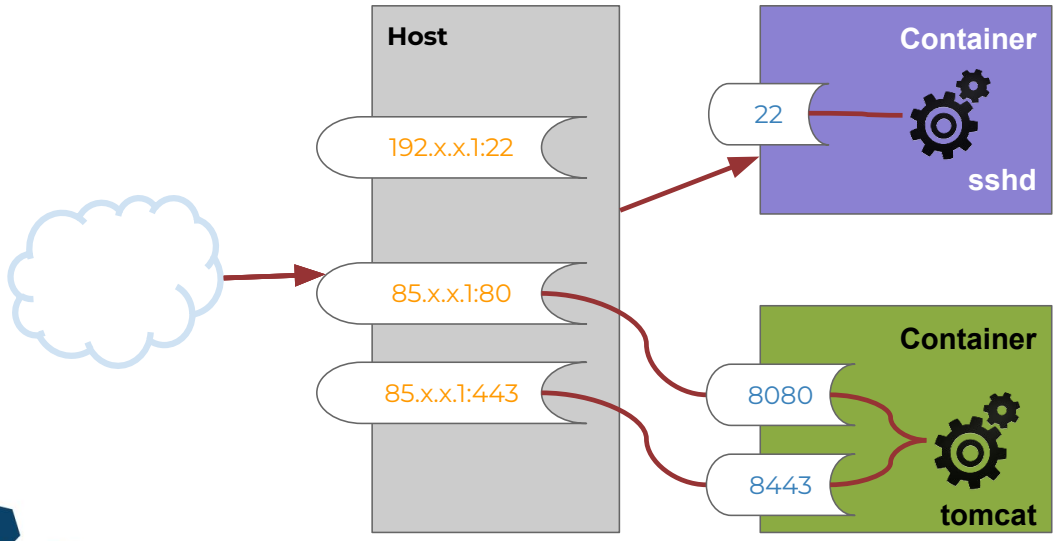


-p 22:22

-p 80:8080

-p 443:8443

Port binding



-p 192.x.x.1:22:22

-p 85.x.x.1:80:8080

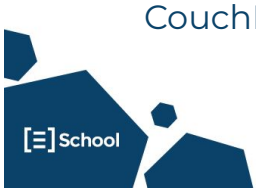
-p 85.x.x.1:443:8443

Publication de ports

- Récupérez l'image couchdb taguée 2.1 :
  - `> docker image pull couchdb:2.1`

- Instanciez un container couchdb1 détaché en exposant le port 5984 sur le port 5984 du host :
  - `> docker container run --name couchdb1 -d` -p 5984:5984 `couchdb:2.1`

- Ouvrez votre navigateur sur l'url http://localhost:5984 , CouchDB affiche sa version.



Docker “management commands”

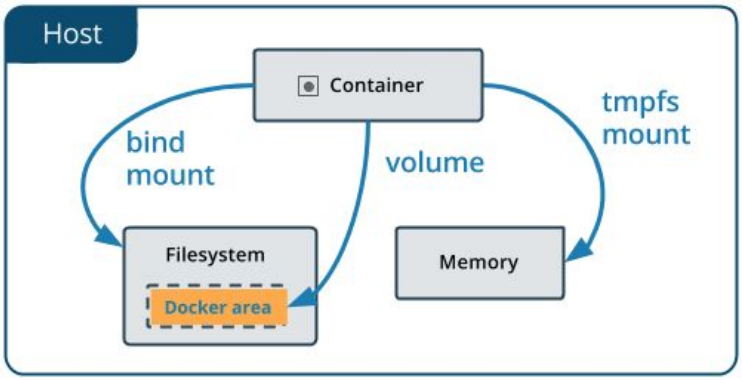
$ docker <OBJECT> <COMMAND>

image	ls, pull, rm, prune, push, tag
container	ls, run **-p**, stop, rm, prune, commit

Lab 2 first docker image

Volumes

Stocker des fichiers : les volumes



Volumes managés

- L'image couchdb1 déclare un volume qui est créé et géré par Docker.
  Listez les volumes gérés par docker :
  - **> docker** volume ls

```
→ Docker git:(main) docker volume ls
DRIVER    VOLUME NAME
local     351dfec9032cb55c019866c85fa6b1fc515c1a88d09da85982419816553e387e
```

- Inspectez le container couchdb1 pour identifier le volume utilisé :
  - **> docker** container inspect **--format "{{json .Mounts}}" couchdb1**

```
→ Docker git:(main) docker container inspect --format "{{json .Mounts}}" couchdb1
[{"Type":"volume","Name":"351dfec9032cb55c019866c85fa6b1fc515c1a88d09da85982419816553e387e","Source":"/var/lib/docker/volumes/351dfec903
2cb55c019866c85fa6b1fc515c1a88d09da85982419816553e387e/_data","Destination":"/opt/couchdb/data","Driver":"local","Mode":"","RW":true,"Pr
opagation":""}]
```

- On peut réutiliser les volumes d'un autre container.
  Lancez un container busybox avec les volumes du container couchdb1 :
  - **> docker container run -it --rm** --volumes-from=couchdb1 **busybox**
  - **$ ls /opt/couchdb/data**

```
/ # ls /opt/couchdb/data
_dbs.couch        _nodes.couch      _replicator.couch  _users.couch
/ #
```

Volumes managés - nettoyage

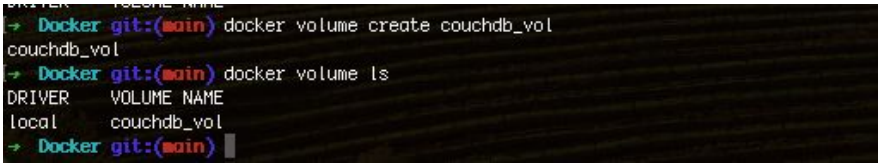
- Arrêtez couchdb1 et supprimez le container ainsi que ses volumes (rm **-V**)
  - **> docker container stop couchdb1**
  - **> docker** container rm -v **couchdb1**

- Vérifiez que le volume a bien été supprimé :
  - **> docker volume ls**

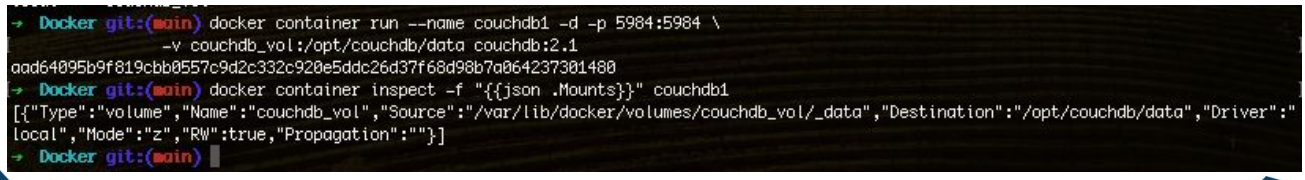
```
/ # exit
→  Docker git:(main) docker container stop couchdb1
couchdb1
→  Docker git:(main) docker container rm -v couchdb1
couchdb1
→  Docker git:(main) docker volume ls
DRIVER    VOLUME NAME
→  Docker git:(main)
```

Volumes nommés

- Vous allez créer un volume nommé et l'utiliser pour le container couchdb1
- Créez un volume nommé :
  - **> docker volume** create couchdb_vol
  - **> docker volume** ls

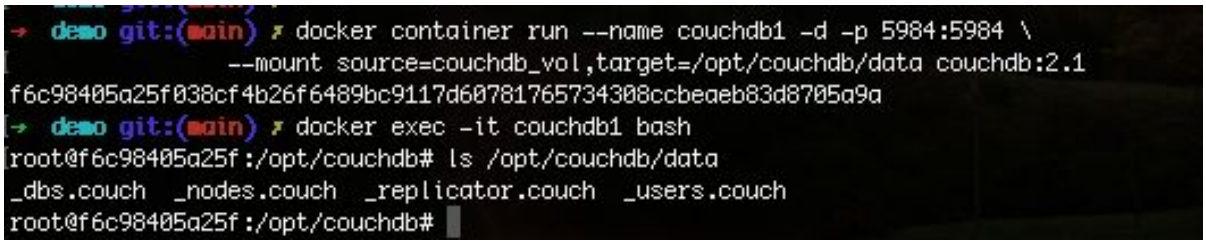


- Recréez un container couchdb1, utilisant cette fois le volume nommé
  - **> docker container run --name couchdb1 -d -p 5984:5984 **
    -v couchdb_vol:/opt/couchdb/data couchdb:2.1
- Inspectez le nouveau container couchdb1 :
  - **> docker container inspect -f "{{json .Mounts}}" couchdb1**

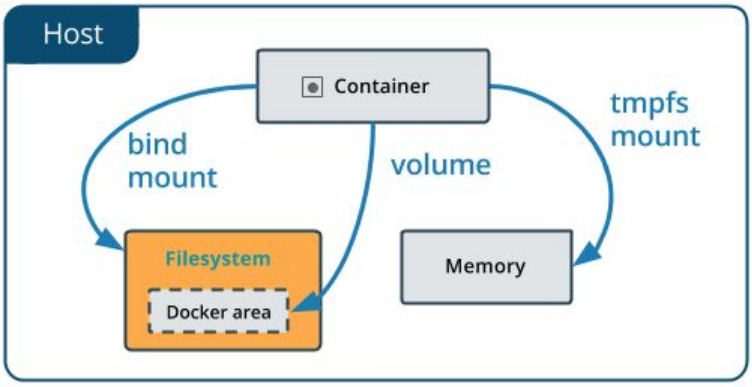


Volumes nommés

- Quelle sont les différence entre -v (--volume) et --mount ?
- Recréez un container couchdb1, utilisant cette fois l'option --mount

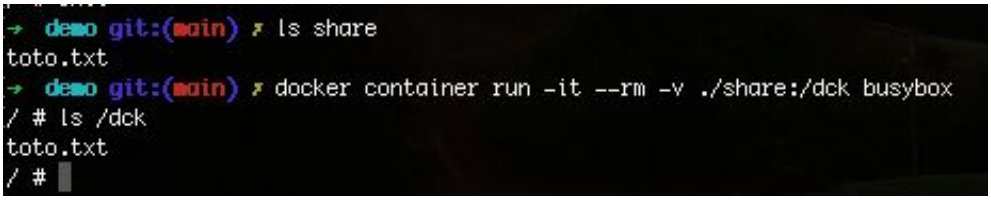


Stocker des fichiers : bind mounts



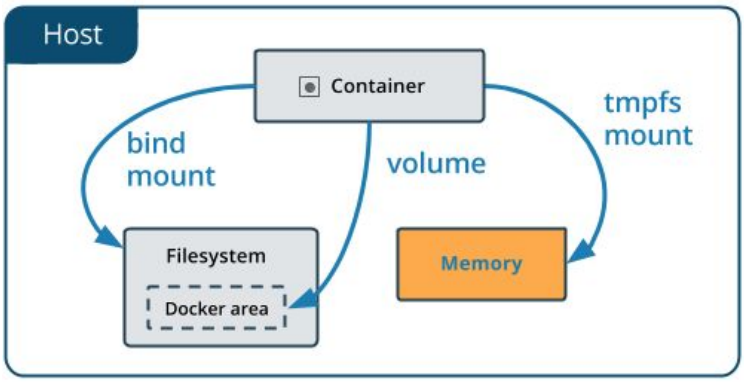
bind mount et astuce

- Lancez un container busybox intéractif en montant le dossier share t du host qui contient le fichier toto.tx vers le dossier /dck du container :
  - **> docker container run -it --rm -v ./share:/dck busybox**
  -
- Regarder le contenu du volume :
  - **$ ls /dck**



- *Note : Le dossier* `share` *est situé dans la VM où tourne le démon Docker.*

Stocker des fichiers : en mémoire



- Pour lancer un container avec un filesystem en mémoire (tmpfs) :
  - **> docker container run -ti** --tmpfs /test **busybox /bin/sh**
  - **> mount | grep test**

```
demo git:(main) ✗ docker container run -ti --tmpfs /test busybox /bin/sh

/ # mount | grep test
tmpfs on /test type tmpfs (rw,nosuid,nodev,noexec,relatime)
/ #
```

Docker “management commands”

$ docker <OBJECT> <COMMAND>

image	ls, pull, rm, prune, push, tag
container	ls, run -p **--rm -v --volumes-from**, stop, rm, **kill, inspect**
volume	ls, create, rm

Lab 3 volumes


Images et Containers
Ports
Volumes
Dockerfile

Dockerfile

- “Image as code”
- Document texte

```
# Create image with you nodejs image
# (replace zbbfufu by your docker id)
FROM zbbfufu/nodejs:1.0

# Declare ports listening in the container
EXPOSE 9000

# Define variables
ENV appDir=/app

# Define current working directory (build & run)
WORKDIR ${appDir}

# Copy all files from “build context” to /app folder
COPY . /app

# Build the application
RUN npm install

# Declare container startup command and options
ENTRYPOINT node
CMD server.js
```

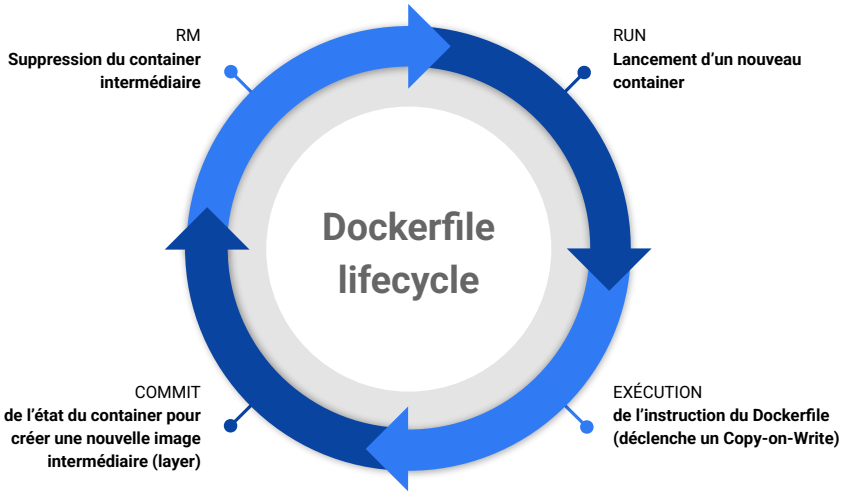
Dockerfile lifecycle

- Chaque **instruction** dans le Dockerfile créer un nouveau layer dans l'image finale qui sera créée.

- Au final, au travers de ces instructions Docker ne fait que réutiliser des features déjà existantes.
  - **run** d'un container
  - **exécution** de l'instruction (ce qui déclenche le Copy-On-Write)
  - **commit** de l'état du container pour créer une nouvelle image transitive
  - **rm** du container intermédiaire

Dockerfile lifecycle

- Chaque **instruction** dans le Dockerfile créer un nouveau layer dans l'image finale qui sera créée.



Docker build console output

```
Sending build context to Docker daemon  2.048kB
Step 1/4 : FROM busybox
 ---> d8233ab899d4
Step 2/4 : ARG CONT_IMG_VER
 ---> Running in 1e56aa344f65
Removing intermediate container 1e56aa344f65
 ---> 3a8255d58f7a
Step 3/4 : ENV CONT_IMG_VER ${CONT_IMG_VER:-v1.0.0}
 ---> Running in f40cdd5a551a
Removing intermediate container f40cdd5a551a
 ---> b80dbb9af46d
Step 4/4 : RUN echo $CONT_IMG_VER
 ---> Running in 15966b838883
v1.0.0
Removing intermediate container 15966b838883
 ---> 40a0d16a7d2a
Successfully built 40a0d16a7d2a
Successfully tagged test-arg:latest
```

Dockerfile lifecycle

- Pour permettre à Docker de gagner en efficacité, celui-ci utilise un cache sauf si l'on précise lors du **build**
  - `> docker build --no-cache=true .`

- Docker **invalide** ce cache dans plusieurs cas qu'il vaut mieux avoir en tête
  - comparaison entre image parent et image fils pour voir si une instruction est la même
  - pour les instructions **ADD** et **COPY**, le cache est invalidé sur le checksum des fichiers ajoutés est différent
  - le check du cache ne s'effectue pas sur les fichiers à l'intérieur du container

FROM

- **Initialise une nouveau “build stage” :** défini l’image de base qui servira de base pour créer la notre. Celle-ci peut être disponible localement ou non et dans ce cas, un Docker ira cherche sur un public repositories.

- On peut placer un **ARG** avant un **FROM** pour définir un argument qui servira dans le **FROM**.

```
ARG  CODE_VERSION=latest
FROM base:${CODE_VERSION}
CMD  /code/run-app
```

ARG

- **ARG** permet de définir des arguments à passer à Docker au moment du *build* d'une image.

- On peut définir des valeurs par défauts directement depuis le dockerfile

```
FROM busybox
ARG version=1.0.0
```

- On défini la valeur de ces arguments au moment du build.
  - **`> docker build --build-arg foo=some_value .`**

LABEL

- **LABEL** permet de définir des metadatas associées à une image Docker

- Les metadatas sont renseignés sous la forme clé ⇒ valeur.

```
LABEL maintainer=”Julien KLAER”
LABEL version=1.0.0
```

- On peut inspect les labels associés à notre image avec.
  - **`> docker inspect`** image

COPY and ADD

- **COPY** permet de copier des fichiers locaux vers le container à partir du build context.

```
COPY file.txt /tmp/destination
```

- **ADD** permet de copier des fichiers locaux, remote et dézipper directement le contenu d'un tar dans le container.

```
ADD application.tar /opt
```

- Par défaut les fichiers copiés appartiennent à **root** mais on peut préciser
  - le user et le groupe doit exister au préalable
  - `COPY --chown=<user>:<group> <src> <dest>`

RUN

- **RUN** permet d'exécuter des commandes sur le container intermédiaire afin de le faire muter pour écrire un nouveau layer sur une image.

```
RUN apt-get update && apt-get dist-upgrade -y && apt-cache clean
```

- Bonne pratique : toujours minimiser la taille des layers

EXPOSE

- **EXPOSE** permet de documenter l'exposition d'un ou plusieurs ports ainsi que de leurs protocols

- Les metadatas sont renseignés sous la forme clé ⇒ valeur.

```
EXPOSE 80/tcp
EXPOSE 80/udp
```

- Pour effectuer le port forwarding il faut faire comme d'habitude
  - **`> docker container run --name couchdb1 -d`** -p 5984:5984 **`couchdb:2.1`**

VOLUME

- **VOLUME** permet de définir un point de montage qui sera effectivement créé lors de l'instanciation d'un nouveau container

- Pour créer un volume il faut utiliser l'instruction.

```
VOLUME /myVolume
```

WORKDIR

- **WORKDIR** permet de définir le répertoire d'exécution des prochaines instructions **RUN**, **CMD**, **ENTRYPOINT**, **COPY** et **ADD**

- On s'en sert simplement de cette manière

```
WORKDIR /opt
```

USER

- **USER** permet de définir le prochain **UID** et optionnellement **GID** dans lesquelles les instructions **RUN**, **CMD** et **ENTRYPOINT** seront exécutées

- Il est aussi très facile de s'en servir.

```
USER webserver:application
```

ENV

- **ENV** permet de définir des **variables d'environnements** qui seront disponible à **l'exécution** du container et lors des prochaines phases du *build*.

- On les définit et s'en sert de la manière suivante.

```
ENV application="my-application"
ENV environment="development"
RUN pass secrets/${environment}/${application}
```

- Il est possible de setter et même overrider ces variables lorsque l'on run un container.

```
> docker container run -e environment="development" busybox
```

ENTRYPOINT and CMD

- **ENTRYPOINT** permet de définir quel exécutable le container va lancer à son instanciation.

- **CMD** est utilisé principalement pour fournir des options par défaut pour l'instruction **ENTRYPOINT**.

- Voici un exemple d'utilisation des deux instructions ensemble.

```
FROM ubuntu
ENTRYPOINT [“top”, “-b”]
CMD [“-c”]
```

Lab 4 dockerfile