

Docker

Table des matières

I - Docker : présentation & vue d'ensemble.....	3
1. Docker : Vue d'ensemble.....	3
1.1. Docker et notion de conteneur.....	3
1.2. Introduction , présentation.....	4
1.3. "Container" vs "VM".....	6
1.4. Isolation des conteneurs.....	7
1.5. Référentiel d'images "docker" prêtes à l'emploi.....	8
1.6. "Docker workflow" (utilisation classique).....	8
1.7. Spécificités de "docker".....	9
1.8. Commandes de "docker".....	10
II - Gestion des images dockers (essentiel).....	12
1. Gestion d'images "docker".....	12
1.1. Informations sur la configuration "docker".....	12
1.2. Rechercher une liste d'images existantes.....	12
1.3. Structure composite d'une image.....	13
1.4. Télécharger une image disponible.....	13
1.5. transfert / copie d'images.....	14
1.6. informations sur une image.....	14
1.7. Vue d'ensemble sur gestion des images.....	14

III - Gestion des "container docker" (essentiel).....	15
1. Gestion des conteneurs "docker".....	15
1.1. Démarrage et arrêt d'un conteneur "docker".....	15
1.2. Stopper l'exécution un conteneur.....	15
1.3. Supprimer un conteneur.....	16
1.4. Mode interactif et lancement d'exécutions de commandes.....	16
1.5. Mapping de ports.....	17
IV - Création de nouvelles images "docker".....	18
1. Gestion d'images "docker".....	18
1.1. Rappel : Vue d'ensemble sur gestion des images.....	18
1.2. Création d'une nouvelle image "docker".....	19
1.3. Exemple "Dockerfile" pour "mysql + schema database".....	22
1.4. Exemple "Dockerfile" pour "spring-boot (java) application".....	22
1.5. Exemple "Dockerfile" pour "nginx + SPA application".....	23
1.6. Détails sur la structure des images "docker".....	24
V - Gestion des volumes "docker".....	25
1. Gestion des volumes "docker".....	25
1.1. Container sans volume configurés(comportement par défaut).....	25
1.2. Vue d'ensemble sur mappings de ports et de volumes.....	26
1.3. mapping de volumes.....	27
1.4. mappings directs de volumes "host".....	28
1.5. Mapping de volumes gérés par conteneur docker.....	30
VI - Network "docker", dialogues entre conteneurs.....	31
1. Gestion "network docker" et compose.....	31
1.1. Vue d'ensemble sur les communications réseaux (docker).....	31
1.2. Comportement du "bridge network" par défaut.....	32
1.3. "User-defined network" (docker).....	34
1.4. Mapping de ports.....	37
1.5. Exemples de paramétrages réseaux.....	37
1.6. Rôle de docker-compose.....	38
1.7. docker-compose vs docker stack deploy.....	38
1.8. docker stack.....	38
1.9. docker-compose.....	39
VII - Annexe – Bibliographie, Liens WEB + TP.....	42
1. Bibliographie et liens vers sites "internet".....	42
2. TP.....	42

I - Docker : présentation & vue d'ensemble

1. Docker : Vue d'ensemble

Docker est une technologie de conteneur logiciel (nouvelle déclinaison plus optimisée de la virtualisation) .

1.1. Docker et notion de conteneur

Micro-conteneurs "Docker"



Technologie légère et efficace de virtualisation
avec comme principaux apports :

- **Coûts optimisés** (car moins consommateur)
- **Déploiements rapides**
- **Portatibilité** (linux/windows , isolation interne , compatibilité externe)

Principes de la "conteneurisation"



Vis à vis de plusieurs conteneurs co-localisés, permet un partage d'un système d'exploitation hôte unique, avec ses ressources (*fichiers binaires, librairies, pilotes de périphériques, mémoire vive, ...*)



Conteneur = **environnement d'exécution** logicielle **complet** comportant :

- micro o.s. (debian, centos ou ...)
- librairies / dépendances
- logiciel de base (ex : mysql, node, ...)
- configuration logicielle
- le code d'une application ou d'un service

1.2. Introduction , présentation

Docker permet de créer des environnements (appelées containers) de manière à isoler des applications.

Docker repose sur le *kernel* **Linux** et sur deux de ses grandes fonctionnalités :

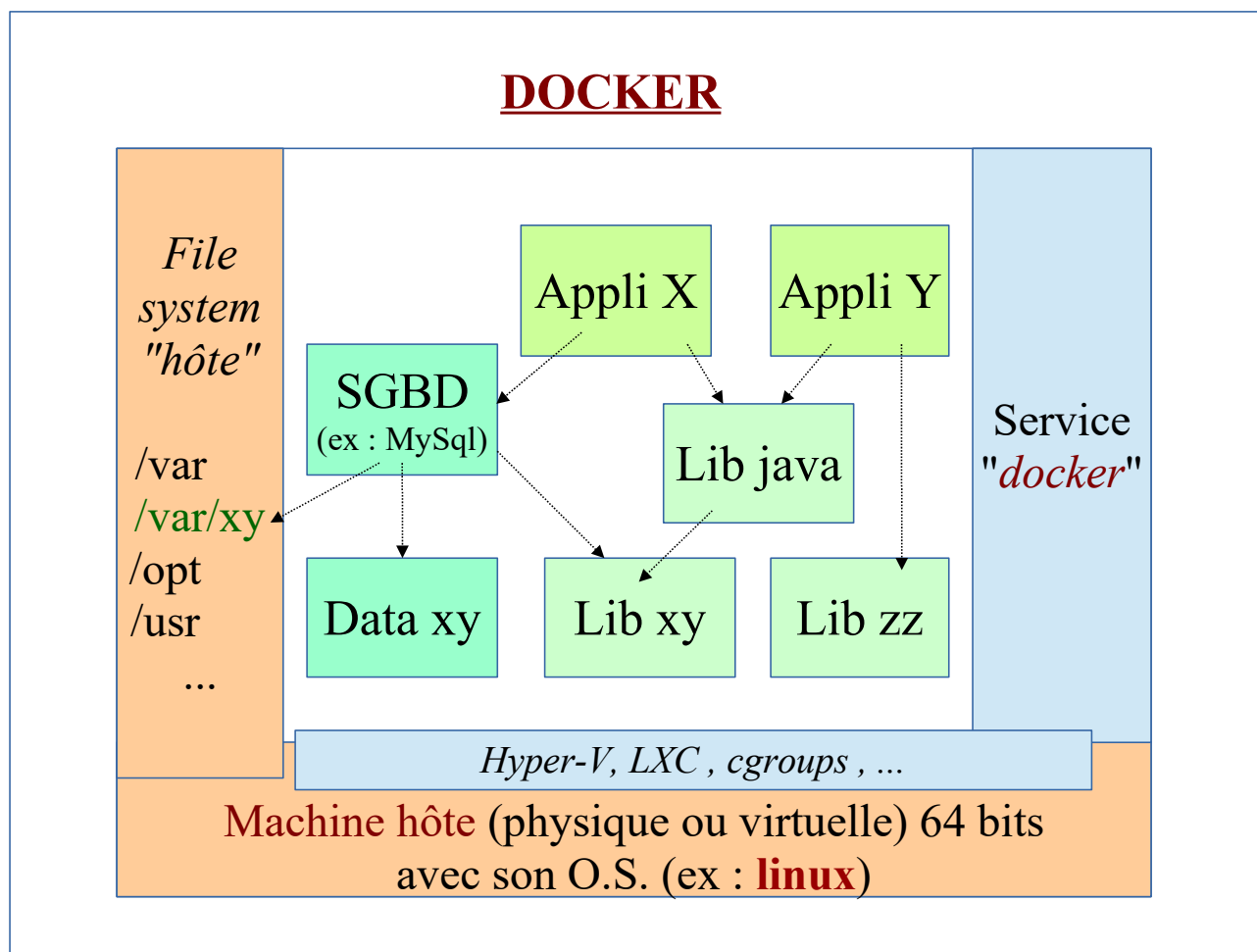
- * les containers LXC : L'idée est d'exécuter une tâche dans un environnement isolé.
- * **cgroups** qui va avoir pour objectif de gérer/partager les ressources (utilisation de la RAM, CPU entre autres).

Historique: **Docker** est un produit développé par la société du même nom. Initialement développé par un ingénieur français, **Solomon Hykes**, le produit a été dévoilé en mars 2013 .

En seulement 3 ans, Docker est devenue une technologie incontournable pour la mise en œuvre du "cloud computing" et est utilisée (avec quelques variantes) par tous les géants de l'informatique (Google, Microsoft, ...).



l'icône de docker est une baleine (Moby Dock) .



Les premières versions de "**Docker**" ne fonctionnaient que sur "**Linux**".

Au fil des années qui passent, "**Docker**" est de **mieux en mieux intégré sur le système "windows" de Microsoft**.

Historiquement, en 2015, pour faire fonctionner Docker sur windows 10 Home (ou autre), il fallait utiliser une machine virtuelle spéciale "**Docker ToolBox**" (basée sur VirtualBox et "**boot2docker**" : un noyau linux minimaliste mais suffisant pour faire fonctionner en RAM un grand nombre de conteneurs "docker"). Ceci n'était pas prévu pour la production mais pour un environnement de développement.

Aujourd'hui "**Docker**" fonctionne nativement sur la version "**professionnel**"/"**serveur**" de **windows 10 et partiellement sur un "windows 10" basique**.

Ceci dit "Docker for windows" est prévu pour faire parfaitement fonctionner des images "dockers pour windows".

Faire fonctionner une image "docker/linux" sous windows est pour l'instant partiellement réalisable (avec plein de restrictions).

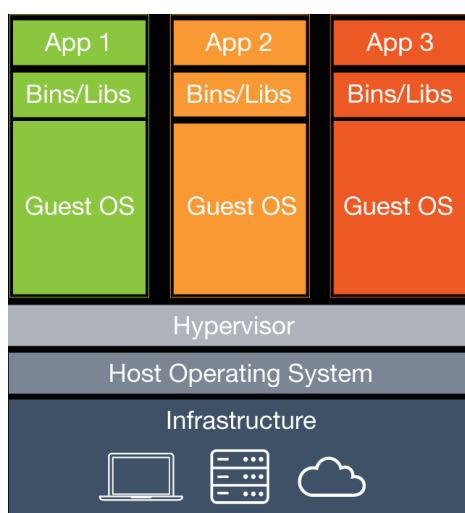
Les futures versions de l'OS "windows" de Microsoft vont évoluer (en se rapprochant de linux) de façon à ce qu'un conteneur "docker/linux" puisse à terme fonctionner sur "windows" ;

1.3. "Container" vs "VM"

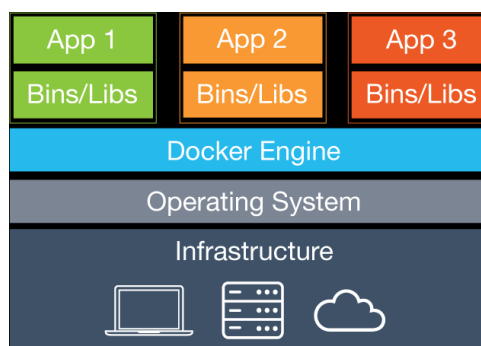
Bien que proche d'une VM (Machine Virtuelle) , **Docker n'est pas une VM** . Sa technologie de "Container" permet de bien mieux partager les ressources d'une machine physique (RAM , CPU , ...)

Evolution de la virtualisation ("VM" → "Conteneur")

V.M. (lourdes)
avec "**O.S. complets internes**"
gérées par hyperviseur
(ex : VmWare , VirtualBox)



Conteneurs (très légers)
avec "applications et dépendances"
gérées moteur de conteneurs
(ex : Docker)



A bas niveau l'ordinateur (cpu/ ram / *bios*, ...) doit être prévu pour partager les ressources matérielles entre plusieurs conteneurs lorsqu'il sera en partie contrôlé par l' **hyperviseur "docker"** .

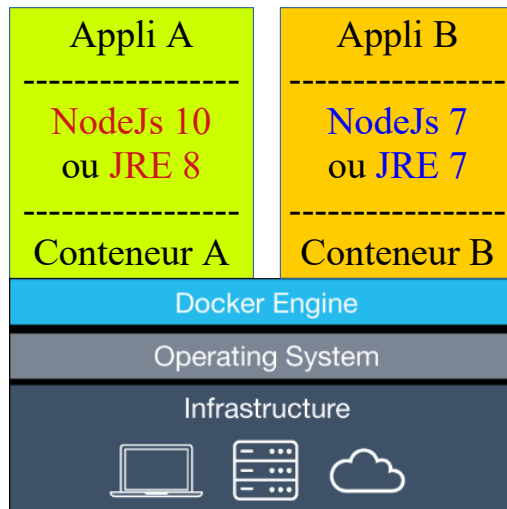
Chaque conteneur sera isolé des autres d'un point de vu "logique" et "fonctionnement interne". Par contre, les différents "conteneurs" qui vont s'exécuter sur un même ordinateur vont partager très efficacement les ressources de l'ordinateur (RAM, CPU, ...) :

- **très faible sur-consommation mémoire** (comparée à celle d'une VM)
- **démarrage très rapide** (comparé à celui d'une VM)

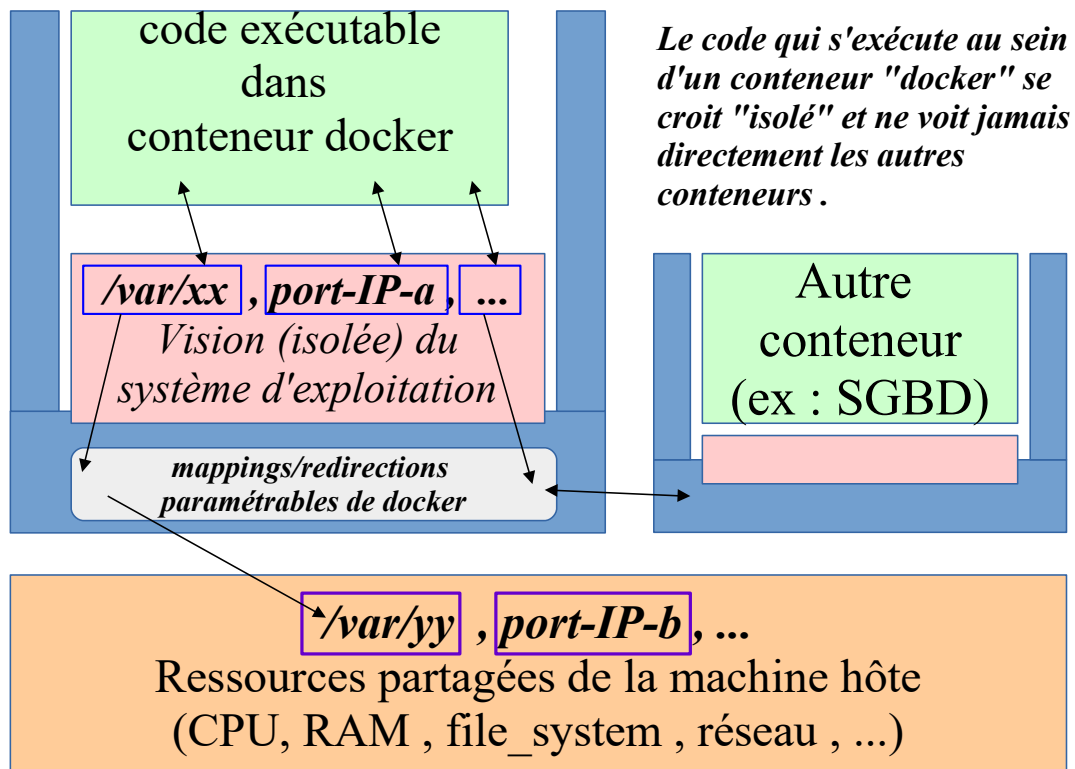
NB : Etant donné que toutes les distributions linux partagent un même type de noyau , un conteneur docker "debian" peut sans problème fonctionner sur un système "redHat" ou "centOs" et vice versa.

1.4. Isolation des conteneurs

Isolation des conteneurs



"Isolation virtuelle"

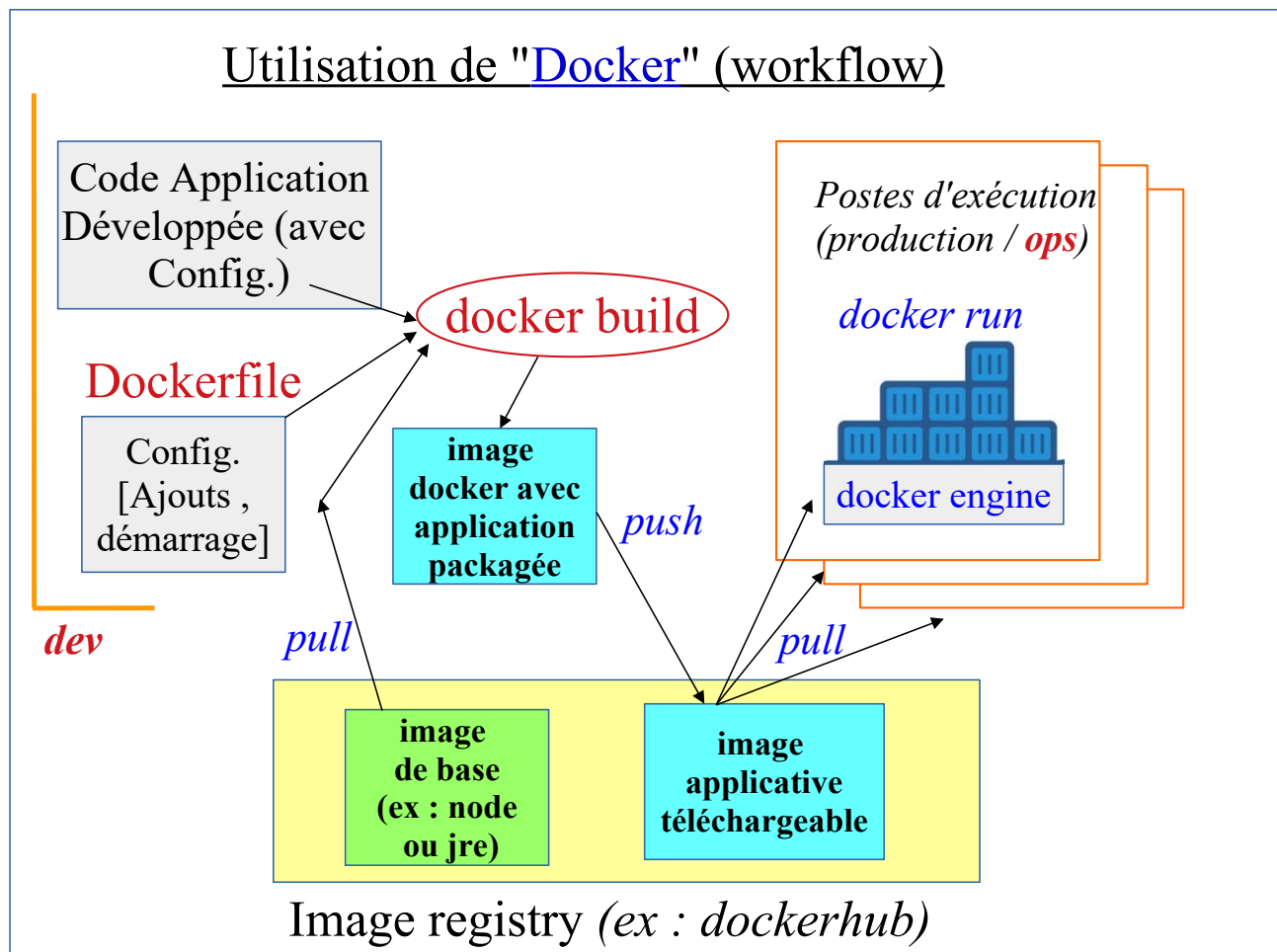


1.5. Référentiel d'images "docker" prêtes à l'emploi

Le site <https://hub.docker.com> permet de télécharger (gratuitement) toute une **série d'images "docker" prêtes à être installée sur n'importe quelle distribution linux** (fedora, redhat , debian , ubuntu , ...).

On y trouve une centaine d'images officielles ("debian" , "mysql" , "postgresql" , "mongo" , "node" , "tomcat" , "jenkins" , ...) et une multitudes d'images "non officielles" .

1.6. "Docker workflow" (utilisation classique)



1.7. Spécificités de "docker"

Spécificités d'un conteneur "docker"

Points forts :

- démarrage assez rapide
- instanciations , démarrages , arrêts , suppressions facilement contrôlables à distance (via api REST) .
Ce qui apporte beaucoup de souplesse dans la gestion des ressources (vite allouées, vite libérées)
- isolation , portabilité (tous linux et bientôt windows)

Points délicats (à avoir en tête)

- tailles de certaines images quelquefois importantes (presque 1Go)
- nécessite réseau à relativement haut débit
- attention aux conflits potentiels entre les numéros de ports (heureusement reconfigurables) entre les éléments internes de différents conteneurs co-localisés.

Les 2 versions actuelles de "docker"

- Docker CE : Community Edition
- Docker EE : Enterprise Edition

1.8. Commandes de "docker"

Avant la version 1.13 certaines commandes étaient pas très structurées (ex : "docker ps" pour afficher la liste des conteneurs et "docker images" pour afficher la liste des images .

A partir de la version 1.13 , on peut :

- soit lancer les commandes historiques (des toutes premières versions) "docker ps, dockers images"
- soit lancer des commandes plus structurées respectant le format

"docker typeElement nomAction" :

exemples :

docker container ls , docker container run

docker image ls , docker image pull ...

Actions les plus courantes (applicables ou pas en fonction du type d'élément):

ls	lister
rm	remove (supprimer)
start / stop	démarrer / arrêter
run	lancer (créer et démarrer) (souvent un container)
inspect	afficher les détails
exec	lancer une commande (selon le contexte)
logs	afficher les logs

docker --help

Usage: docker [OPTIONS] COMMAND

A self-sufficient runtime for containers

Options:

--config string Location of client config files (default
"/root/.docker")
-D, --debug Enable debug mode
-H, --host list Daemon socket(s) to connect to
-l, --log-level string Set the logging level
("debug"|"info"|"warn"|"error"|"fatal")
(default "info")
--tls Use TLS; implied by --tlsverify
--tlscacert string Trust certs signed only by this CA (default
"/root/.docker/ca.pem")
--tlscert string Path to TLS certificate file (default
"/root/.docker/cert.pem")
--tlskey string Path to TLS key file (default
"/root/.docker/key.pem")
--tlsverify Use TLS and verify the remote
-v, --version Print version information and quit

Management Commands:

builder Manage builds
config Manage Docker configs
container Manage containers
engine Manage the docker engine

image	Manage images
network	Manage networks
node	Manage Swarm nodes
plugin	Manage plugins
secret	Manage Docker secrets
service	Manage services
stack	Manage Docker stacks
swarm	Manage Swarm
system	Manage Docker
trust	Manage trust on Docker images
volume	Manage volumes

Commands:

attach	Attach local standard input, output, and error streams to a running container
build	Build an image from a Dockerfile
commit	Create a new image from a container's changes
cp	Copy files/folders between a container and the local filesystem
create	Create a new container
diff	Inspect changes to files or directories on a container's filesystem
events	Get real time events from the server
exec	Run a command in a running container
export	Export a container's filesystem as a tar archive
history	Show the history of an image
images	List images
import	Import the contents from a tarball to create a filesystem image
info	Display system-wide information
inspect	Return low-level information on Docker objects
kill	Kill one or more running containers
load	Load an image from a tar archive or STDIN
login	Log in to a Docker registry
logout	Log out from a Docker registry
logs	Fetch the logs of a container
pause	Pause all processes within one or more containers
port	List port mappings or a specific mapping for the container
ps	List containers
pull	Pull an image or a repository from a registry
push	Push an image or a repository to a registry
rename	Rename a container
restart	Restart one or more containers
rm	Remove one or more containers
rmi	Remove one or more images
run	Run a command in a new container
save	Save one or more images to a tar archive (streamed to STDOUT by default)
search	Search the Docker Hub for images
start	Start one or more stopped containers
stats	Display a live stream of container(s) resource usage statistics
stop	Stop one or more running containers
tag	Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE
top	Display the running processes of a container
unpause	Unpause all processes within one or more containers
update	Update configuration of one or more containers
version	Show the Docker version information
wait	Block until one or more containers stop, then print their exit codes

Run 'docker COMMAND --help' for more information on a command.

II - Gestion des images dockers (essentiel)

1. Gestion d'images "docker"

Rappel: la source des principales images disponibles est <https://hub.docker.com> .

1.1. Informations sur la configuration "docker"

docker version

---> 18.09.6 ou autre

docker info

--> system info (ex: /var/lib/docker , <https://index.docker.io/v1/> , ...)

docker images ou bien **docker image ls**

---> affiche la liste des images locales :

<i>REPOSITORY</i>	<i>TAG</i>	<i>IMAGE ID</i>	<i>CREATED</i>	<i>SIZE</i>
tomcat	8.0-jre8	b9d9b29582c6	3 days ago	332.7 MB
mysql	latest	18f13d72f7f0	3 days ago	383.4 MB
debian	latest	ddf73f48a05d	3 days ago	123 MB
hello-world	latest	c54a2cc56cbb	12 weeks ago	1.848 kB

1.2. Rechercher une liste d'images existantes

docker search --filter=stars=10 mysql

---> recherche la liste des images bien notées (au moins 10 étoiles) comportant "mysql" :

<i>NAME</i>	<i>DESCRIPTION</i>	<i>STARS</i>	<i>OFFICIAL</i>	<i>AUTOMATED</i>
mysql	MySQL is a widely used,	3127	[OK]	
mysql/mysql-server	Optimized MySQL Server	203		[OK]
centurylink/mysql	Image containing mysql.	46		[OK]
sameersbn/mysql		38		[OK]

1.3. Structure composite d'une image

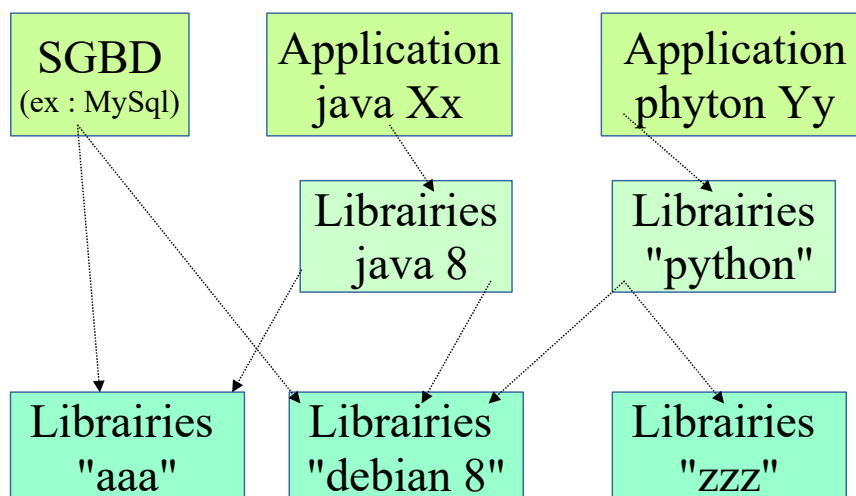
Une image "docker" est très souvent construite à partir d'une autre image (FROM ...)
Ces relations d'héritage sont visibles au sein des fichiers "**Dockerfile**".

Exemple : l'image "**tomcat :8.0-jre8**" est basée sur l'image "**openjdk:8-jre**" elle même basée sur une image "**debian/jessie**".

Un fichier "**Dockerfile**" permet essentiellement de paramétrer un **delta** entre une image de base et certains éléments installés et configurés en plus (apt-get ... , chmod ...).

Ceci permet une bonne ré-utilisation des images (ou sous-images) et offre une grande souplesse dans la configuration des versions.

Partages automatiques de bibliothèques selon "héritages" entre les images dont les conteneurs sont issus .



* L'image de l'application "Java Xx" hérite de l'image "java 8" qui hérite elle même de "debian 8". Beaucoup d'autres images héritent de "debian 8".

* Lors de l'exécution le conteneur "debian 8" sera partagé et réutilisé .

1.4. Télécharger une image disponible

docker pull mysql ou bien **docker image pull mysql**

--> télécharge l'image "mysql" (environ 384 MB)

docker pull debian

--> télécharge l'image "debian" (environ 123 MB)

docker image pull tomcat:8.0-jre8

--> télécharge l'image "tomcat" avec le tag 8.0-jre8 (environ 332 MB)

1.5. transfert / copie d'images

docker save

- exporte une image au format tar.gz

docker load

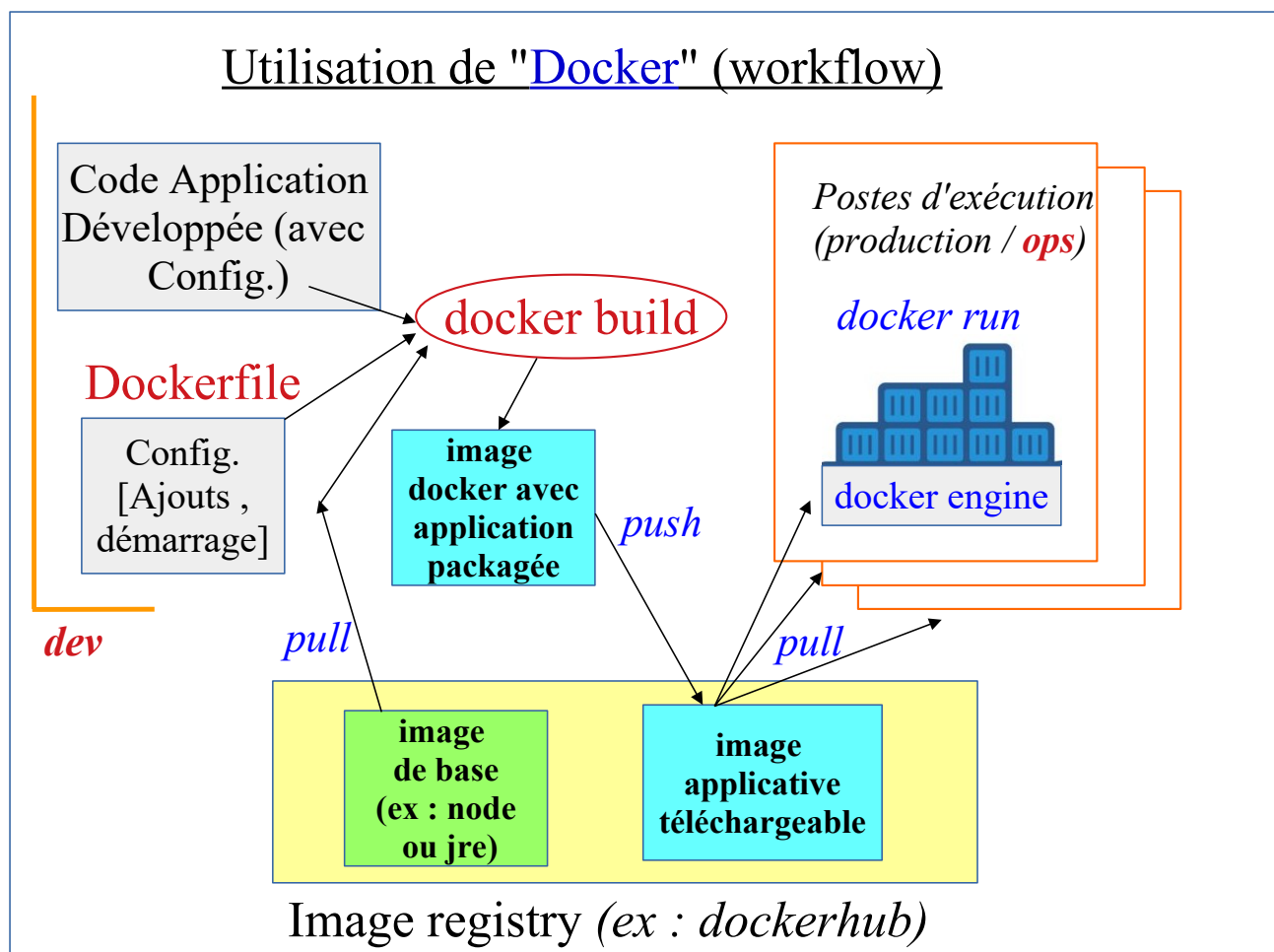
- importe une image au format tar.gz

1.6. informations sur une image

docker image inspect

docker image history

1.7. Vue d'ensemble sur gestion des images



NB : La création de nouvelle image docker sera abordée dans un chapitre ultérieur.

III - Gestion des "container docker" (essentiel)

1. Gestion des conteneurs "docker"

1.1. Démarrage et arrêt d'un conteneur "docker"

docker ps -l

--> affiche les containers docker qui sont en cours (ou qui ont fonctionné) :

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
f53851ab521d	debian	"/bin/bash"	24 seconds ago	Up 23 seconds		sharp_goodall

docker container ls --> affiche la liste des conteneurs actifs

docker container ls -a --> affiche la liste de tous les conteneurs (actifs ou stoppés)

docker container ls -q --> affiche que les IDs des conteneurs actifs (pratique pour les arrêter)

docker run --name "xy-container-name"

lance successivement deux commandes :

docker create --name "xy-container-name"

docker start xy-container-name

L'option fondamentale **-d** permet un démarrage en **mode détaché (background , &)** . Ce qui est indispensable pour lancer un serveur qui ne s'arrête pas aussitôt tout en ne bloquant pas la console de démarrage .

L'option **-e** permet de fixer la valeur d'une variable d'environnement (prévue pour être analysée par le container lancé)

Exemple :

docker run --name mysql-container -e MYSQL_ROOT_PASSWORD=root -d mysql

(ou bien **docker container run ...**)

NB : sans option **--name** , le conteneur lancé a un nom par défaut que l'on ne contrôle pas bien

docker container inspect xy-container-name

- retourne au format JSON un énorme paquet d'informations dont l'adresse IP du conteneur (ex : "IPAddress": "172.17.0.4")

1.2. Stopper l'exécution un conteneur

docker stop containerId (ou bien docker container stop containerId)

docker stop f53851ab521d

1.3. Supprimer un conteneur

docker rm *containerId* (ou bien docker container rm *containerId*)
docker rm f53851ab521d

1.4. Mode interactif et lancement d'exécutions de commandes

Lancement d'un nouveau conteneur (éphémère) en mode interactif :

```
docker container run -t -i debian
```

NB: option **-i** pour allouer un pseudo terminal TTY
option **-t** pour garder ce pseudo-terminal ouvert

Exemple (pas très utile):

```
root@debian9-cloud:/home/power-user# docker container run -t -i debian
root@5d21966f5f29:/# ls
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var
root@5d21966f5f29:/# exit
exit
root@debian9-cloud:/home/power-user#
```

Au sein de l'exemple précédent, les commandes **"ls"** et **"exit"** ont été exécutées au sein du conteneur docker lancé (basé sur une image debian) .

Lancement d'une seule commande au sein d'un conteneur existant :

```
docker container run --name node-container -d xyz/node-web-app
```

```
docker container ls
```

```
docker container exec node-container ls
```

```
root@debian9-cloud:/home/power-user# docker container exec node-container ls
Dockerfile
node_modules
package-lock.json
package.json
server.js
root@debian9-cloud:/home/power-user# docker container exec node-container pwd
/usr/src/app
root@debian9-cloud:/home/power-user# █
```

Lancement d'un shell interactif au sein d'un conteneur existant :

```
docker container exec -ti node-container sh
```



```
root@debian9-cloud:/home/power-user# docker container exec -ti node-container sh
# pwd
/usr/src/app
# ls
Dockerfile  node_modules  package-lock.json  package.json  server.js
# exit
root@debian9-cloud:/home/power-user#
```

--> ceci peut être très utile et pratique pour du "debug" .

1.5. Mapping de ports

Mapping statique de numéro de port: **-p** NUM_PORT_HOST:NUM_PORT_CONTAINER

NB :

- il existe aussi l'option **-P** NUM_PORT_CONTAINER qui permet un mapping dynamique mais ceci est moins contrôlable (à adapter selon le contexte).
- l'option **-h** permet de choisir le *pseudo "hostname"* correspondant au conteneur lancé

Exemple :

```
docker run --name mysql-container \
  -e MYSQL_ROOT_PASSWORD=root \
  -h mysql.container.host \
  -p 3307:3306 \
  ... \
  -d mysql:5.7
```

Cet exemple montre un lancement d'un conteneur mysql

A l'intérieur du conteneur, mysql fonctionne avec le port par défaut 3306

A l'extérieur du conteneur (sur la machine hôte) , le port associé est 3307 (par exemple pour éviter un conflit)

Autres exemples :

```
docker container run -d -p8080:80 nginx
```

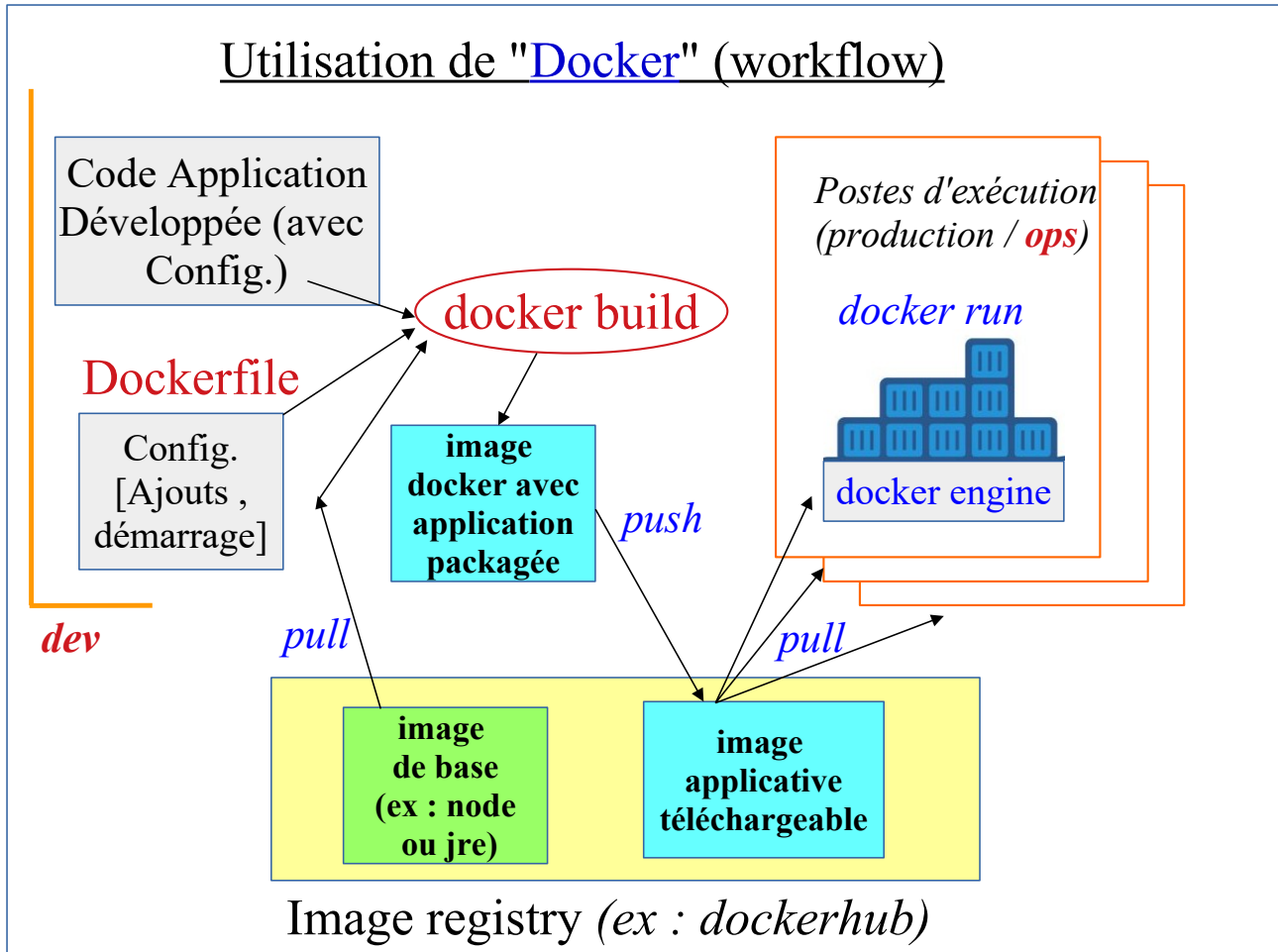
où nginx est une image de serveur HTTP (concurrent sérieux et asynchrone de Apache Http Server)

```
docker run --name oracle-container \
  -p 18081:8080 -p 1521:1521 \
  -h oracle.container.host \
  ... \
  -d sath89/oracle-xe-11g
```

IV - Création de nouvelles images "docker"

1. Gestion d'images "docker"

1.1. Rappel : Vue d'ensemble sur gestion des images



1.2. Création d'une nouvelle image "docker"

Premier exemple basique: image avec nodeJs et une mini application web "hello world" .

Phase préliminaire (codage et test de l'application sous linux-debian):

installation de "nodejs" et "npm":

```
su
curl -sL https://deb.nodesource.com/setup_10.x | bash -
apt-get install -y nodejs
exit
node --version
npm --version
```

création du projet "hello" et installation du module "express":

```
npm init
npm install --save express
more package.json
nano server.js
```

server.js

```
'use strict';
const express = require('express');
const PORT = 8080;
const HOST = '0.0.0.0';
const app = express();
app.get('/', (req, res) => {
  res.send('Hello world\n');
});
app.listen(PORT, HOST);
console.log(`Running on http://${HOST}:${PORT}`);
```

nano package.json

```
{
  "name": "hello",
  "version": "1.0.0",
  "description": "hello app for docker",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "author": "developpeur fou",
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.0"
  }
}
```

premier test (sans docker) :

npm run start (ou *npm start* ou *node server.js*)
et ouvrir *http://localhost:8080* depuis un navigateur web (ex : *firefox*)
Ctrl-C

Configuration "Dockerfile" de la nouvelle image "docker" :

touch Dockerfile

nano **Dockerfile**

```
FROM node:8
# this new image will be create from parent image = node:8 (stable)

# Create app directory inside docker image
WORKDIR /usr/src/app

# Install app dependencies
# A wildcard is used to ensure both package.json AND package-lock.json are copied
# where available (npm@5+)
COPY package*.json ./

RUN npm install
# If you are building your code for production
# RUN npm ci --only=production

# Bundle app source
COPY . .

#setting MODE ENV-VARIABLE to "prod" (not "dev")
ENV MODE=prod

EXPOSE 8080
CMD [ "npm", "start" ]
```

touch .dockerignore

nano **.dockerignore**

```
node_modules
npm-debug.log
```

Construction de l'image docker "xyz/node-web-app" :

su

```
docker build -t xyz/node-web-app .
```

ou bien

```
docker image build -t xyz/node-web-app .
```

NB: l'option **-t xyz/node-web-app** sert simplement à "taguer" l'image construite

l'argument important **.** désigne le répertoire courant contenant **Dockerfile** et **.dockerignore**

docker images

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
xyz/node-web-app	latest	8b55063e4bd1	5 minutes ago	897MB
...				

Lancement d'un conteneur pour tester l'image construite :

port 49160 sur machine hôte mappé avec port 8080 de l'appli dans conteneur docker

```
docker run -p 49160:8080 -d xyz/node-web-app
```

ou bien

```
docker container run -p 49160:8080 -d xyz/node-web-app
```

#vérification :

```
docker ps (ou bien docker container ls)
```

```
docker logs 8053a5aaa9a7_or_containerId_display_by_pocker_ps
```

```
curl http://localhost:49160
```

--> *Hello world*

```
curl -i http://localhost:49160
```

*# ou bien ouvrir un navigateur avec l'url **http://localhost:49160***

Autre variante possible pour "Dockerfile" :

```
FROM centos:latest
MAINTAINER Name Here <username@localhost>
RUN rpm -Uvh http://mirror.pnl.gov/epel/7/x86_64/e/epel-release-7-5.noarch.rpm
RUN yum install nodejs npm -y
COPY ./src /opt/src
RUN cd /opt/src; npm install
EXPOSE 8080
CMD ["node", "/opt/src/server.js"]
```

Explications des différences :

- L'image parente "centos:latest" est plus basique (linux sans nodejs)
- Il faut y ajouter l'installation de nodejs (via *yum* sur *centos* , ce serait via *apt-get* sur *debian/ubuntu*)
- Le fichier "Dockerfile" est (sur le poste "hôte") à coté d'un répertoire /src comportant lui même server.js et package.json
- Au sein de l'image , on a pas utilisé la commande **WORKDIR** et on est donc situé à la racine (dans l'interprétation des "chemins relatifs internes") .
- L'application est lancée en direct via node (sans passer par npm run start) .

1.3. Exemple "Dockerfile" pour "mysql + schema database"

schema.sql

```
CREATE DATABASE IF NOT EXISTS deviseApiDb charset=utf8;
USE deviseApiDb;

CREATE TABLE IF NOT EXISTS devise(
    code VARCHAR(34) ,
    monnaie VARCHAR(64),
    tauxChange double ,
    PRIMARY KEY(code)) ENGINE=InnoDB;
```

Dockerfile

FROM mysql:5.7

from parent image = mysql:5.7(stable and compatible with nodeJs mysql2 client)

```
ENV MYSQL_DATABASE=deviseApiDb \
    MYSQL_ROOT_PASSWORD=root
```

ADD schema.sql /docker-entrypoint-initdb.d

EXPOSE 3306

1.4. Exemple "Dockerfile" pour "spring-boot (java) application"

Dockerfile

FROM openjdk:8

this new image will be create from parent image = openjdk:8(stable)

Create app directory inside docker image

WORKDIR /usr/app

COPY target/deviseApi.jar ./

#optimisation possible : <https://spring.io/guides/topicals/spring-boot-docker>

EXPOSE 8080

CMD ["java","-jar","deviseApi.jar"]

avec .jar préalablement fabriqué via maven (**pom.xml**) et le plugin maven suivant :

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration>
    <mainClass>org.mycontrib.api.MySpringBootApplication</mainClass>
    <layout>ZIP</layout>
  </configuration>
</plugin>
```

1.5. Exemple "Dockerfile" pour "nginx + SPA application"

Dockerfile

FROM nginx:1.16

this new image will be create from parent image = nginx:1.16(stable)

#adding ping and curl command for debug via docker run -it /bin/bash

#RUN apt-get update && apt-get install -y iputils-ping && apt-get install -y curl

#copy angular app directory:

COPY /dist/devise-app /usr/share/nginx/html/devise-app

#copy docker-nginx.conf for reverse-proxy devise-api (node/express or springBoot)

COPY docker-nginx.conf /etc/nginx/conf.d/default.conf

CMD ["nginx", "-g", "daemon off;"]

.dockerignore

```
src
e2e
node_modules
```

Le contenu du fichier ".dockerignore" précédent est **spécifique à une application angular** (où src et node_modules ne servent qu'à construire (via **ng build --prod --base-href.**) une application dans le répertoire "dist" .

docker-nginx.conf

```
server {
    listen      80;
    server_name localhost;

    #NB: ordre important dans ce fichier : du plus precis au plus general
    #syntaxes basees sur regexp

    # docker run ... --network mynetwork --network-alias=devise.api.service xyz/devise-api
    # docker run -p 80:80 --name devise-ngapp-container --network mynetwork xyz/devise-ngapp

    #config pour rediriger les appels WS-REST vers api rest (nodeJs ou SpringBoot ou ...)
    #NB: resolver 127.0.0.11 refer to embedded docker DNS service
    #(used for resolving devise.api.service : backend docker container)
    location ~ ^/devise-app/deviseApi/(.*){
        resolver 127.0.0.11;
        proxy_pass http://devise.api.service:8282/deviseApi/$1?$args;
    }
}
```

```
#config pour les pseudo-urls de angular (router)
#toutes url/uri en /devise-app/xxxx (autre que devise-app/deviseApi/... plus haut)
#entraînera une redirection vers /devise-app/index.html :
location ~ ^/devise-app/(.*){
    root /usr/share/nginx/html;
    index index.html index.htm;
    try_files $uri $uri/ /devise-app/index.html;
}

location / {
    root /usr/share/nginx/html;
    index index.html index.htm;
}

error_page 500 502 503 504 /50x.html;
location = /50x.html {
    root /usr/share/nginx/html;
}
}
```

1.6. Détails sur la structure des images "docker"

Layers ...

image = plein de "layers" en "read only"

container = utilisation des layers d'une image + layer en "read/write"

partage de layers entre images (selon héritage ou ...)

création d'une image depuis un container et quelques modifs --> ***docker commit***

DockerFile = méthode conseillée

....

cache ...

... prune pour supprimer fichiers inutiles et gagner de l'espace disque ...

V - Gestion des volumes "docker"

1. Gestion des volumes "docker"

1.1. Container sans volume configurés (comportement par défaut)

Un conteneur docker génère certains fichiers (ex : logs, ...) durant son fonctionnement. Ces fichiers sont par défaut stockés sous le répertoire ***var/lib/docker/volumes*** de la machine hôte .

Lorsque le container héberge un serveur de base de données (ex : oracle, mysql , postgres, ..., mongodb, ...) , il est essentiel de bien gérer la persistance des données associée à un conteneur.

Sans "volume docker" , il est possible d'effectuer un mapping entre certains répertoires (virtuels) internes de l'image et des répertoires réels du système de fichiers de la machine hôte .

Cette solution a tout de même certains inconvénients :

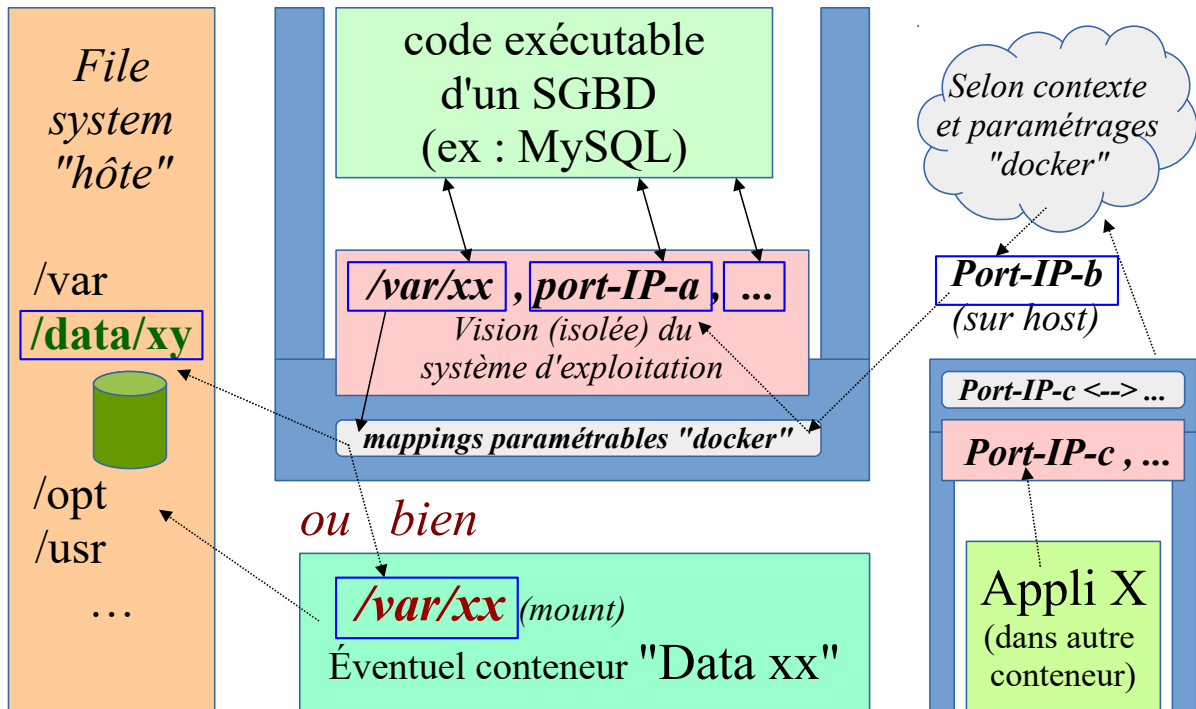
- moindre portabilité (selon différences entre machines hôtes)
- sauvegardes/restaurations délicates
- partages délicats

La solution conseillée pour gérer la persistance des données correspond à des "volumes dockers" qui sont partageables et sauvegardables/restaurables .

NB : L'instruction **VOLUME** dans le "**Dockerfile**" permet de spécifier des répertoires/volumes virtuels du conteneur docker qui seront ultérieurement mappés à des répertoires/volumes externes au conteneur (par défaut ou pas) .

1.2. Vue d'ensemble sur mappings de ports et de volumes

Mappings de ports "ip" et de volumes



...

...


...

1.3. mapping de volumes

Mappings volumes ("host")

```
docker run ... -v volumePathInHost:volumePathInContainer ...
```

```
Exemple: docker run --name mysql-container \  
-e MYSQL_ROOT_PASSWORD=root \  
-v /storage/docker/mysql-datadir:/var/lib/mysql \  
-d mysql
```



À préparer (accessible ?) et à *ultérieurement régulièrement sauvegarder !*
(ne sera pas ré-initialisé à l'arrêt , ni à la suppression du conteneur) .

Partage/réutilisation de volumes ("autre conteneur")

Exemple:

```
docker run -d --name db2 \  
--volumes-from db1 training/postgres
```

1.4. mappings directs de volumes "host"

(solution basique pas idéale)

Exemple :

1a_run_mysql_docker_container.sh

```
#!/bin/bash

#linux path:
export MYSQL_DATA_STORAGE_DIR=/storage/docker/mysql-datadir

sudo mkdir -p ${MYSQL_DATA_STORAGE_DIR}

# /script in mysql-container is mapped to
# /home/poweruser/Bureau/docker-scripts/mysql-scripts of host
# via -v option of "docker run ..." (launched before "docker exec ...")

#linux path:
export MYSQL_SCRIPTS_IN_HOST=/home/poweruser/Bureau/docker-scripts/mysql-scripts

export MYSQL_SCRIPTS_IN_CONTAINER=/scripts

docker stop mysql-container
docker rm mysql-container

docker run --name mysql-container \
    -e MYSQL_ROOT_PASSWORD=root \
    -h mysql.container.host \
    -v ${MYSQL_DATA_STORAGE_DIR}:/var/lib/mysql \
    -v ${MYSQL_SCRIPTS_IN_HOST}:${MYSQL_SCRIPTS_IN_CONTAINER} \
    -d mysql:5.7

#NB: default username=root
```

2a_exec_init_db_mysql_docker_container.sh

```
#!/bin/bash

# /script in mysql-container is mapped to
# /home/poweruser/Bureau/docker-scripts/mysql-scripts of host
# via -v option of "docker run ..." (launched before "docker exec ...")

export MYSQL_SCRIPTS=/scripts
docker exec mysql-container bash -c ${MYSQL_SCRIPTS}/init-db.sh
```

avec (sur machine "hôte") :

```
.../docker-scripts/mysql-scripts/init-db.sh
```

```
export MYSQL_SCRIPTS=/scripts
mysql -u root -proot < ${MYSQL_SCRIPTS}/mydb.sql
```

et

```
.../docker-scripts/mysql-scripts/mydb.sql
```

```
CREATE DATABASE IF NOT EXISTS test;
USE test;
DROP TABLE IF EXISTS Customer;
CREATE TABLE Customer (id integer primary key auto_increment, name VARCHAR(64));
INSERT INTO Customer(id,name) VALUES (1 , "first customer");
select * from Customer;
```

```
#ou interactivement :
#docker exec -it mysql-container bash

# with in bash interactive console :
# mysql -u root -p root -e 'CREATE DATABASE IF NOT EXISTS test;USE test;DROP TABLE IF EXISTS
Customer;CREATE TABLE Customer(id integer primary key auto_increment,name VARCHAR(64));INSERT INTO
Customer(id,name) VALUES (1 , "first customer");'
# exit
```

1.5. Mapping de volumes gérés par conteneur docker

(solution sophistiquée conseillée)

NB :

- Historiquement , avant la version 1.9 (de novembre 2015) , un volume "docker" se gérait via "docker create -v" .
- Depuis la version 1.9 , la commande "**docker volume ...**" est préférable .

...

```
docker volume create --name mysql-data-volume
```

```
docker run -p 3306:3306 -d --name devise-db-container \
-v mysql-data-volume:/var/lib/mysql \
--network mynetwork --network-alias=devise.db.service xyz/devise-db
```

backup_mysql_data.sh

```
#!/bin/bash
# lancement d'un container de type "debian:9" sans nom car execution très courte
# /backup est un alias interne pour $(pwd)
# --volumes-from devise-db-container permet d'accéder aux volume du "data_container" devise-db-container
# ou bien -v mysql-data-volume:/var/lib/mysql permet d'associer le chemin interne /var/lib/mysql
# au data volume préparé et nommé "mysql-data-volume"
# la commande tar cvf sera lancée pour créer backup.tar dans $(pwd)

echo "stopping mysql-container";
docker container stop devise-db-container
echo "pause before saving data , ... , enter to continue";read suite;

#docker run --rm \
#    --volumes-from devise-db-container \
#    -v $(pwd):/backup debian:9 \
#    tar cvf /backup/mysql-backup.tar /var/lib/mysql

docker run --rm \
-v mysql-data-volume:/var/lib/mysql \
-v $(pwd):/backup debian:9 \
tar cvf /backup/mysql-backup.tar /var/lib/mysql

echo "pause before restarting mysql-container , ... , enter to continue";read suite;
docker container start devise-db-container
```

restore_mysql_data.sh

```
....

docker run --rm \
-v mysql-data-volume:/var/lib/mysql \
-v $(pwd):/backup debian:9 \
tar xvf /backup/mysql-backup.tar

...
```

VI - Network "docker", dialogues entre conteneurs

1. Gestion "network docker" et compose

1.1. Vue d'ensemble sur les communications réseaux (docker)

docker network ls

<i>NETWORK ID</i>	<i>NAME</i>	<i>DRIVER</i>
7fca4eb8c647	bridge	bridge
9f904ee27bf5	none	null
cf03ee007fb4	host	host

<i>type de réseau géré par docker</i>	<i>caractéristiques du type de réseau</i>
bridge (<i>fréquent, par défaut</i>)	communication réseau possible avec machine hôte , autres machines et autre conteneurs (isolation paramétrable via mapping de ports , ...)
none (<i>rare</i>)	pas de communication réseau . pour isolation radicale
host (<i>rare</i>)	le conteneur docker réutilise directement le réseau de la machine hôte sans isolation
overlay (<i>si docker swarm</i>)	pour prise en compte du clustering en mode "docker swarm" (kubernetes) .
macvlan (<i>rare</i>)	possibilité de configurer adresse MAC (de bas niveau)
via plugin (<i>à développer</i>)	pour communications très spécifiques
"user defined network" (généralement basé sur "bridge")	variante de "bridge" avec meilleur contrôle des communications entre conteneurs dockers installés sur même machine hôte .

1.2. Comportement du "bridge network" par défaut

Lancement de 2 conteneurs "c1" et "c2" basés sur l'image debian , lancés en mode détaché avec la commande "bash" (en mode -it) . On pourra ultérieurement se rattacher à un des shells (bash) de ces conteneurs via la commande "**docker attach c1_ou_c2**"

```
docker run -d -it --name c1 debian bash
```

```
docker run -d -it --name c2 debian bash
```

docker container ls

7e83ffd23774	debian	"bash"	About a minute ago	Up About a minute	c2
8936d501ac9e	debian	"bash"	About a minute ago	Up About a minute	c1

docker network inspect bridge

```
-->
[
  {
    "Name": "bridge",
    "Id": "d91412adbd698864f218b1e00c3353b9f42d708e305585c5bbb81626902184d2",
    "Created": "2019-06-19T10:53:55.370153663+02:00",
    "Scope": "local",
    "Driver": "bridge",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": null,
      "Config": [
        {
          "Subnet": "172.17.0.0/16",
          "Gateway": "172.17.0.1"
        }
      ]
    },
    "Internal": false,
    "Attachable": false,
    "Ingress": false,
    "ConfigFrom": {
      "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {
      "7e83ffd237740ee18e098fbdd661af0c7dc6f4c4f78efafb26f960cb6796c4d9": {
        "Name": "c2",
        "EndpointID": "a8bdbc47720eed7d3b54d7b80c28d78a7326ff8c757ad6ee8f5a1c66da5baf61",
        "MacAddress": "02:42:ac:11:00:03",
        "IPv4Address": "172.17.0.3/16",
        "IPv6Address": ""
      },
      "8936d501ac9ee79cb7f89ecae0ac492ab5c3885b3e498777936cea82180482": {
        "Name": "c1",
        "EndpointID": "42a4b12606653e5668e47762958ce02c6e663ba62d4a27aac50e72b0e35423c4",
        "MacAddress": "02:42:ac:11:00:02",
        "IPv4Address": "172.17.0.2/16",
        "IPv6Address": ""
      }
    },
    "Options": {
      "com.docker.network.bridge.default_bridge": "true",
      "com.docker.network.bridge.enable_icc": "true",
      "com.docker.network.bridge.enable_ip_masquerade": "true",
      "com.docker.network.bridge.host_binding_ipv4": "0.0.0.0",

```



```
"com.docker.network.bridge.name": "docker0",
"com.docker.network.driver.mtu": "1500"
},
"Labels": {}
}
]
```

==> les conteneurs "c1" et "c2" se voient attribuer des adresses ip par défaut (ici 172.17.0.2 pour c1 et 172.17.0.2 pour c2).

docker attach c1

```
#
# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
5: eth0@if6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
        valid_lft forever preferred_lft forever
```

```
# ping -c 2 www.google.com
PING www.google.com (172.217.22.132) 56(84) bytes of data.
64 bytes from par21s12-in-f4.1e100.net (172.217.22.132): icmp_seq=1 ttl=55 time=6.57 ms
64 bytes from par21s12-in-f4.1e100.net (172.217.22.132): icmp_seq=2 ttl=55 time=6.48 ms
```

```
--- www.google.com ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1000ms
rtt min/avg/max/mdev = 6.486/6.531/6.577/0.092 ms
```

==> une machine externe (ex : www.google.com) est accessible depuis le conteneur (par défaut en mode "bridge")

```
# ping -c 2 172.17.0.3
PING 172.17.0.3 (172.17.0.3) 56(84) bytes of data.
64 bytes from 172.17.0.3: icmp_seq=1 ttl=64 time=0.076 ms
64 bytes from 172.17.0.3: icmp_seq=2 ttl=64 time=0.053 ms
```

```
--- 172.17.0.3 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1021ms
rtt min/avg/max/mdev = 0.053/0.064/0.076/0.014 ms
```

==> le conteneur "c2" est accessible depuis le conteneur c1 via un accès selon l'adresse ip de c2 (par défaut en mode "bridge")

```
# ping -c 2 c2
ping: c2: Name or service not known
```

==> le conteneur "c2" n'est pas accessible depuis le conteneur c1 via un accès selon son nom "c2" (par défaut en mode "bridge")

```
# exit
```

docker container stop c1 c2
docker container rm c1 c2

1.3. "User-defined network" (docker)

La création d'un nouveau réseau virtuel docker personnalisé s'effectue via la commande suivante :

```
docker network create --driver bridge my-net-xy
```

Le nouveau réseau **my-net-xy** est alors visible (en mode driver=bridge) via "docker network ls" .

Test de comportement :

```
docker run -d -it --name c1 --network my-net-xy debian bash
```

```
docker run -d -it --name c2 --network my-net-xy --network-alias=c2bis debian bash
docker run -d -it --name c3 debian bash
docker run -d -it --name c4 --network my-net-xy debian bash
docker run -d -it --name c5 debian bash
docker network connect bridge c4
docker network connect --alias c5bis my-net-xy c5
```

=>

- les conteneurs c1,c2,c4 sont connectés au réseau **my-net-xy** dès leurs démarrages via l'option **--network my-net-xy**
- le conteneur c2 est dès sa création associé à un second nom qui sera vu sur le réseau **my-net-xy** via l'option **--network-alias=c2bis**
- les conteneur c3 et c5 sont ici volontairement démarrés sans l'option **--network** et sont donc connectés au réseau "bridge" par défaut
- le conteneur c4 est connecté à un deuxième réseau ici "bridge" via la commande "docker network connect"
- le conteneur c5 est connecté à un deuxième réseau ici "**my-net-xy**" via la commande "docker network connect" . l'option **--alias c5bis** fait qu'il y sera vu soit via le nom c5 soit via le nom c5bis .

docker network inspect bridge

```
[ { "Name": "bridge", ...
  "IPAM": { ... "Config": [ { "Subnet": "172.17.0.0/16", "Gateway": "172.17.0.1" } ]
}, ...
  "Containers": {
    "1890d91d87e21a4b6e1e38182604470df35b971f0eae01f0e60f5c63b84d4b34": {
      "Name": "c3", ...
      "IPv4Address": "172.17.0.2/16", "IPv6Address": ""
    },
    "7184af79fe2b53b46fb46a0c4f74bce50d2fcbd50eb3429c702b7a12d23173ed": {
      "Name": "c4", ...
      "IPv4Address": "172.17.0.3/16", "IPv6Address": ""
    },
    "7184af79fe2b53b46fb46a0c4f74bce50d2fcbd50eb3429c702b7....": {
      "Name": "c5", ...
```

```

        "IPv4Address": "172.17.0.4/16",      "IPv6Address": ""
    }
},
...
}
]

```

docker network inspect my-net-xy

```

[
  {
    "Name": "my-net-xy",    ...    "Driver": "bridge",    "EnableIPv6": false,
    "IPAM": {    ...,    "Config": [    { "Subnet": "172.19.0.0/16", "Gateway": "172.19.0.1"    }    ]    },
    ...
    "Containers": {
      "232ff5a96bd40319eed8cebac0bab4f33f7a15fa21bb927e69ac5281bca3cabd": {
        "Name": "c2", ... , "IPv4Address": "172.19.0.3/16", "IPv6Address": ""
      },
      "4d709f52a939d498cc3ee2bb47b3a1b57c47bafac21c1f774960b7a06922a191": {
        "Name": "c1", ... , "IPv4Address": "172.19.0.2/16", "IPv6Address": ""
      },
      "7184af79fe2b53b46fb46a0c4f74bce50d2fcbd50eb3429c702b7a12d23173ed": {
        "Name": "c4", .... , "IPv4Address": "172.19.0.4/16", "IPv6Address": ""
      },
      "e6be889cf100f79e49ee12a2ec8e4990a2a8ca2a0a0522dea73e21d7b47eac4b": {
        "Name": "c5", ..., "IPv4Address": "172.19.0.5/16", "IPv6Address": ""
      }
    }, ...
  }
]

```

docker attach c4

ping -c 2 c1

ING c1 (172.19.0.2) 56(84) bytes of data.

64 bytes from c1.my-net-xy (172.19.0.2): icmp_seq=1 ttl=64 time=0.085 ms

64 bytes from c1.my-net-xy (172.19.0.2): icmp_seq=2 ttl=64 time=0.043 ms

--- c1 ping statistics ---

2 packets transmitted, 2 received, 0% packet loss, time 1021ms

rtt min/avg/max/mdev = 0.043/0.064/0.085/0.021 ms

==> le conteneur "c1" est bien accessible (selon son nom) depuis le conteneur "c4" car il sont tous les deux connectés au même réseau docker personnalisé my-net-xy .

```
# ping -c 2 c3
```

```
ping: c3: Name or service not known
```

==> **c4 et c3 sont connectés au réseau commun "bridge" mais cela ne suffit pas pour un accès via le nom du conteneur ("bridge" est le réseau par défaut. ce n'est pas un réseau personnalisé).**

```
# ping -c 2 172.17.0.2
```

```
PING 172.17.0.2 (172.17.0.2) 56(84) bytes of data.  
64 bytes from 172.17.0.2: icmp_seq=1 ttl=64 time=0.080 ms  
64 bytes from 172.17.0.2: icmp_seq=2 ttl=64 time=0.054 ms  
--- 172.17.0.2 ping statistics ---  
2 packets transmitted, 2 received, 0% packet loss, time 1015ms  
rtt min/avg/max/mdev = 0.054/0.067/0.080/0.013 ms
```

==> **le conteneur "c3" est tout de même accessible depuis "c4" via son adresse ip .**

```
ping -c 2 c2bis
```

```
PING c2bis (172.19.0.3) 56(84) bytes of data.  
64 bytes from c2.my-net-xy (172.19.0.3): icmp_seq=1 ttl=64 time=0.075 ms  
64 bytes from c2.my-net-xy (172.19.0.3): icmp_seq=2 ttl=64 time=0.060 ms  
  
--- c2bis ping statistics ---  
2 packets transmitted, 2 received, 0% packet loss, time 1030ms  
rtt min/avg/max/mdev = 0.060/0.067/0.075/0.011 ms
```

==> **le conteneur c2 est également visible depuis son alias "c2bis" spécifié dès son démarrage via l'option --network-alias=c2bis de la commande docker run**

```
ping -c 2 c5bis
```

```
PING c5bis (172.19.0.5) 56(84) bytes of data.  
64 bytes from c5.my-net-xy (172.19.0.5): icmp_seq=1 ttl=64 time=0.107 ms  
64 bytes from c5.my-net-xy (172.19.0.5): icmp_seq=2 ttl=64 time=0.055 ms  
  
--- c5bis ping statistics ---  
2 packets transmitted, 2 received, 0% packet loss, time 1018ms  
rtt min/avg/max/mdev = 0.055/0.081/0.107/0.026 ms
```

==> **le conteneur c5 est également visible depuis son alias "c5bis" spécifié lors de sa connexion au réseau personnalisé my-net-xy via l'option --network-alias=c2bis de la commande docker network connect**

```
# exit
```

```
docker container stop c1 c2 c3 c4
```

```
docker container rm c1 c2 c3 c4
```

```
docker network rm my-net-xy
```

Conclusion :

Connecter 2 conteneurs "docker" (co-localisés sur la même machine) sur un même réseau personnalisé permet d'accéder à un conteneur depuis un autre via son nom et nom seulement via son adresse ip (allouée dynamiquement selon adresses déjà utilisées)

```
docker network create --driver bridge my-net-xy
```

```
docker run -d --name c1 --network my-net-xy [ --network-alias=c1.host ] image1  
docker run -d --name c2 --network my-net-xy [ --network-alias=c2.host ] image2
```

NB : l'option **-h** de "docker container run" ne fait que fixer le **"hostname" interne** du conteneur .

1.4. Mapping de ports

Mappings de ports "ip"

```
docker run -d ... -p hostPort:containerPort myServerContainer
```

Exemple: `docker run -d ... --publish 6603:3306 mysql`
`docker run -d ... -p 6603:3306 mysql`

→ Ceci peut être utile pour éviter des conflits entre plusieurs instances.
 L'option -p (ou bien --publish) peut être utilisée si besoin plusieurs fois

Anciens Mappings (liaisons) d'adresses IP

```
docker run --link myServerContainer:serverHostAlias-  
myClientContainerImage
```

→ l'option --link des premières versions de docker est maintenant
 considérée comme obsolète (à remplacer par --network ou autre équivalent)

1.5. Exemples de paramétrages réseaux

création d'un réseau virtuel docker personnalisé (pour bien communiquer entre les 3 conteneurs):

```
docker network create --driver bridge mynetwork
```

Lancement d'un conteneur basé sur une image héritant de mysql :

```
docker run -p 3306:3306 -d --name devise-db-container \  
--network mynetwork --network-alias=devise.db.host xyz/devise-db
```

--> le code interne du conteneur suivant pourra se connecter à la base mysql de ce conteneur via
devise.db.host et **3306** .

Lancement d'un conteneur basé sur une image héritant de node (nodeJs):

```
docker run -p 8282:8282 -d --name devise-api-container \  
--network mynetwork --network-alias=devise.api.host xyz/devise-api
```

--> le code interne du conteneur suivant pourra se connecter à l'appli nodeJs de ce conteneur via
 l'url **http://devise.api.host:8282**

Lancement d'un conteneur basé sur une image héritant de nginx (intégrant appli. SPA/angular):

```
docker run -p 80:80 -d --network mynetwork \  
--name devise-ngapp-container xyz/devise-ngapp
```

--> **http://localhost:80**

1.6. Rôle de docker-compose

docker-compose (ou bien le nouvel équivalent "docker stack") permet d'automatiser le déploiement et le démarrage de conteneurs "docker" en configurant un fichier de configuration globale (**docker-compose.yml**)

Ainsi , en un seul fichier de configuration, on peut paramétrer un assemblage cohérent de conteneurs "docker" permettant de prendre en charge les différentes couches logicielles d'une application (GUI/web , REST-api , DataBase) .

En interne , docker-compose va automatiquement lancer les commandes "docker" de bas niveaux "docker network ..." , "docker container run ..." avec les bonnes options de façon à obtenir un tout cohérent au sein duquel les éléments communiquent bien entre eux.

1.7. docker-compose vs docker stack deploy

docker-compose existe depuis longtemps en tant que fonctionnalité annexe (codée en python).

"**docker stack**" n'existe qu'au sein des version récentes de docker et nécessite au minimum la version 3 des fichiers .yaml de composition .

"**docker stack**" est intégré dans le coeur de "**docker engine**" .

Vis à vis de "docker-compose" , la principale restriction de "docker stack" est l'impossibilité de construire dynamiquement de nouvelles images : il faut utiliser des images existantes.

"docker stack" ne fonctionne qu'en mode "swarm" (pour clusters).

"docker-compose" peut fonctionner avec ou sans "swarm" .

1.8. docker stack

docker stack deploy --compose-file docker-compose.yml my-stack

ou

docker stack deploy -c docker-compose.yml my-stack

1.9. docker-compose

NB : docker compose doit être installé en plus de docker engine.

exemple :

```
apt-get install docker-compose
```

NB : **docker-compose** utilise par défaut le fichier *docker-compose.yml*

déploiement/démarrage d'une stack :

```
docker-compose up &
```

arrêt d'une stack :

```
docker-compose down &
```

NB : Selon la version de docker-compose installée, utilisée il faudra peut-être placer version : '2.0' dans le fichier *docker-compose.yml*.

Exemple de fichier *docker-compose.yml* :

```
# docker-compose file for "Angular app in nginx + REST api in nodeJs/express + mysql DB"
# version must be '3.0' or '3.1' for docker stack and must be '2.0' for (old) docker-compose

version: '2.0'

networks:
  mynetwork:
    driver: bridge

services:
  db:
    image: xyz/devise-db
    ports:
      - 3306:3306
    networks:
      mynetwork:
        aliases:
          - devise.db.host
          - devise.db.service

  backend-api:
    image: xyz/devise-api
    ports:
      - 8282:8282
    networks:
      mynetwork:
        aliases:
          - devise.api.host
          - devise.api.service
```

```
frontend:
  image: xyz/devise-ngapp
  ports:
    - 80:80
  networks:
    mynetwork:
```

Attention : au sein des fichiers **.yaml** , pas de ~~tabulation~~ mais des doubles espaces pour les indentations

ANNEXES

VII - Annexe – Bibliographie, Liens WEB + TP

1. Bibliographie et liens vers sites "internet"

2. TP