LICENCE DEV
Docker-compose

Patrice Gommery – Février 2022

PRE-REQUIS

Accéder au VM dédiées à Docker dans la salle H205.
Avoir reçu son **VMID** (N° de machine dans la salle H205)

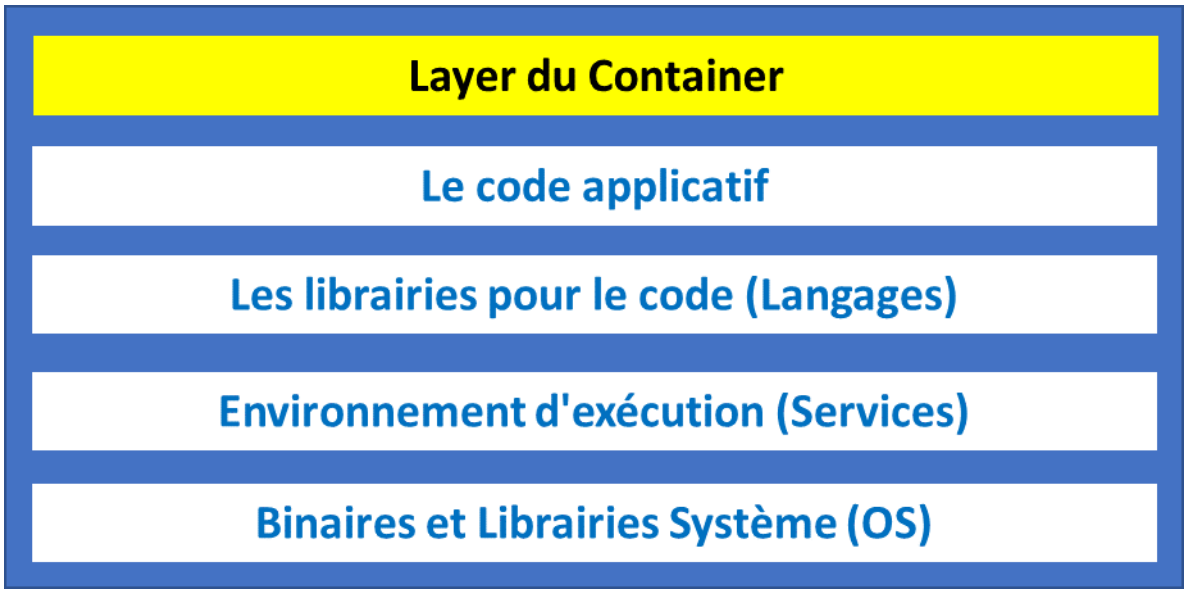
Si vous êtes dans la salle H205 (sur un poste de la salle) :
Ouvrez un terminal et connectez vous avec ssh root@172.16.**VM.ID**
Rappel, le mot de passe est votre code étudiant URCA.

Si vous n'êtes pas sur un des postes de la salle H205 :
Ouvrez un terminal et connectez vous avec :
ssh vpnmmi@vpnmmi.h205.online (Demandez le mot de passe à l'enseignant)
ssh licdev@192.168.3.2 (Même mot de passe que vpnmmi)
et pour terminer : **ssh root@172.16.VM.ID (mot de passe : codeEtudiantUrca)**

Préambule :

Dans cette partie de l'introduction à Docker, nous reviendrons sur la notion de volume, importante à comprendre pour garantir une persistance des données. Nous aborderons aussi un autre composant de Docker : docker-compose . Une application liée à docker (écrite en python) qui nous permettra de créer des stacks de containers.

Volume Docker :

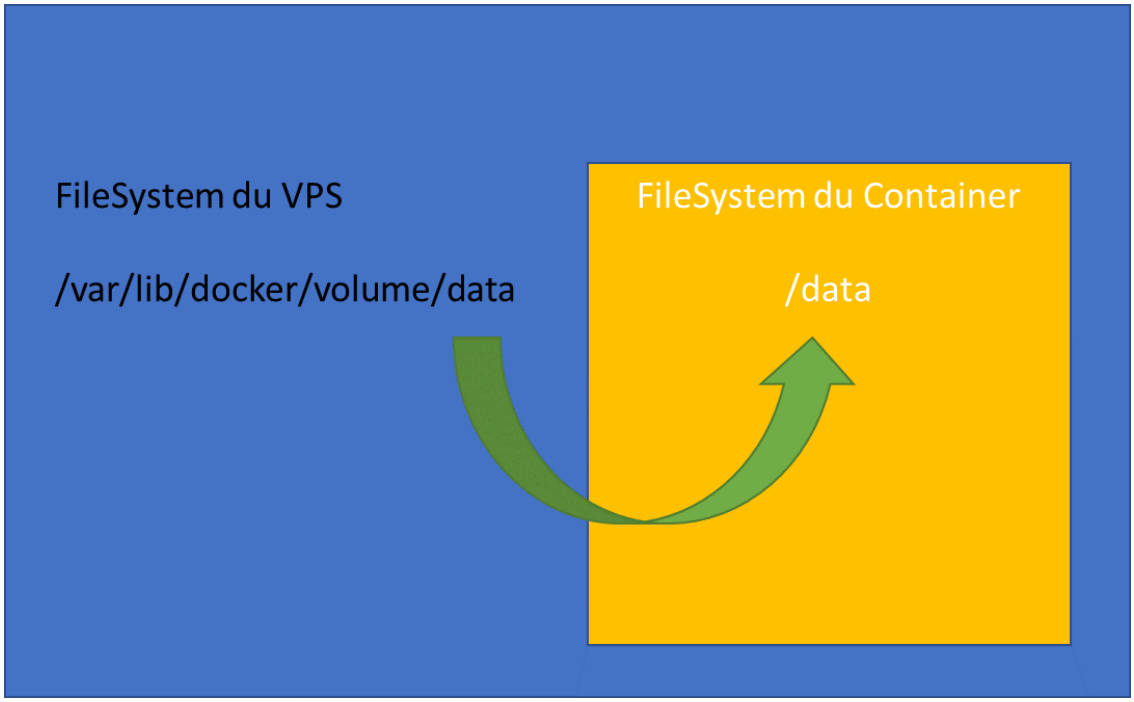


Nous avons donc vu dans le TP précédent que lorsqu'un container est instancié à partir d'un image, Docker créait automatiquement une couche supplémentaire qui va donc permettre d'écrire dans le container. Que ce soit pour stocker des données utilisateurs ou tout simplement changé la configuration du système, des services ou des applications installés à partir de l'image initiale.

Mais que devient cette couche et les données qu'elle contient si l'on supprime le container ? Vous vous en doutez, elles sont supprimées avec le container. Donc toutes les modifications ou données ajoutées sont perdues. Même si l'on recréait un container en partant de la même image comme nous l'avons vu dans le TP précédent, les données n'existent plus car elles étaient enregistrées dans la layer du container, pas dans l'image.

Comment faire pour conserver des données après la suppression d'un container ? C'est là qu'intervient la notion de **volumes** qui va donc nous permettre de garantir **une persistance des données.**

Qu'est-ce qu'un volume Docker ? Par défaut il s'agit d'un emplacement (un dossier) créé sur la machine hôte (votre poste de travail) qui est partagé avec le container. En terme Linuxien, on parle de monté le volume dans le container.



Par exemple, dans le schéma ci-dessus, le Volume data est monté dans le dossier **/data** du container. Concrètement, cela signifie simplement que tout ce qui sera écrit dans **/data** sera en vérité écrit dans le dossier **/var/lib/docker/volume/data** de l'hôte. Si l'on supprime le container, les données seront conservées sur l'hôte. Si l'on recréait un container avec le même point de montage, celui-ci aura accès aux données comme celui qui avait été supprimé.

Pour bien comprendre le principe, essayons de voir comment cela fonctionne.

SANS VOLUME : PAS DE PERSISTANCE DES DONNEES

Créez un premier container avec les caractéristiques suivantes :
- Nom de container : **test1**
- Image de base : **alpine**
- Mode de démarrage : **Terminal interactif (Pas de background)**

A l'invite de commandes du terminal du container:
- installez **nano** avec la commande :

```
apk add nano
```

- Créez un fichier nommé **Hello1.txt** avec un peu de contenu
- Quittez le terminal avec **exit**

De retour sur l'invite de commandes votre machine hôte:
- Cherchez le fichier **Hello1.txt** avec la commande

```
find / -name Hello1.txt
```

/var/lib/docker/overlay2/3acc0e8cbc4ca4a75a57c726fb9f835fbfa2fe4fbc6615dba3a50d76795abca8/diff/**Hello1.txt**

Normalement, vous devriez trouver votre fichier dans un sous-dossier **/var/lib/docker/overlay2** qui est le dossier de stockage des couches utilisées par les containers, donc les "layers containers". Si vous éditez ce fichier, vous pouvez constater qu'il s'agit bien du votre.

Maintenant supprimez le container et relancez la commande **find** comme précédemment.
Que constatez-vous ? Effectivement votre fichier n'est plus présent sur la machine hôte, il a été supprimé en même temps que le container.

Partie 1 : VOLUMES

Voyons maintenant, comment la création d'un volume peut nous apporter une **persistance des données**.

AVEC L'OPTION -V

Créez un premier container avec les caractéristiques suivantes :
- Nom de container : **test2**
- Image de base : **alpine**
- Mode de démarrage : **Terminal interactif (Pas de background)**
- **et ajoutez (avant le nom de l'image) l'option -v documents**

A l'invite de commandes du terminal du container:
- installez nano avec la commande **apk add**
- Créez un fichier nommé **Hello2.txt** avec un peu de contenu
- Quittez le terminal avec **exit**

De retour sur l'invite de commandes votre machine hôte:
- Cherchez le fichier **Hello2.txt** avec la commande **find**

/var/lib/docker/overlay2/7cdc59ee8016fe22d428975290582eded874b7f188b1a5baaa4ceb92aaa39936/diff/**Hello2.txt**

Si vous êtes observateur, vous devriez constater (ID mise à part) que votre fichier est dans le même dossier **overlay2** que le fichier Hello1.txt du test précédent. D'après-vous sera-t-il conservé si l'on détruit le container ? Simple à vérifier : **supprimez le container** et relancez la commande **find**.

Encore une fois, plus de fichier **Hello2.txt**.
Alors à quoi sert l'option -v ou avons nous commis une erreur ?

Créez un nouveau container **test3** avec les mêmes options que test2 et regardons ce qui se passe dans Docker. Installer nano (apk add) et créez un fichier **Hello3a.txt** avant de quitter le container.

Avant de continuer, installer le paquet **jq** sur votre VM. **jq** est un parseur json pour bash qui nous permettra de mieux lire la configuration de Docker.

Maintenant, inspectons notre container pour savoir si un volume a été créé et où ?

```
docker container inspect -f "{{ json .Mounts }}" test3 | jq
```


```
[
  {
    "Type": "volume",
    "Name": "ce64c14ded8707ca287c47c17cd9543846ee3c3f8ffea6d18b5a555bcf392072",
    "Source": "/var/lib/docker/volumes/ce64c14ded8707ca287c47c17cd9543846ee3c3f8ffea6d18b5a555bcf392072/_data",
    "Destination": "documents",
    "Driver": "local",
    "Mode": "",
    "RW": true,
    "Propagation": ""
  }
]
```


Effectivement, Docker à bien créer un volume pour ce container.
Ce volume est dans le dossier **/var/lib/docker/volumes**
et son nom (ou plutôt son ID ici) est dans l'exemple :
ce64c14ded8707ca287c47c17cd9543846ee3c3f8ffea6d18b5a555bcf392072
le sous-dossier **_data** étant celui qui finalement contiendra réellement les données.

Mais à quel point de montage correspond-t-il dans le container ?
Placez-vous dans le dossier indiqué par source dans le résultat de l'inspection
et créez un fichier **Hello3b.txt**

```
cd /var/lib/docker/volumes/...ID../_data
nano Hello3b.txt
```

Ensuite relancer le container **test3** en mode interactif.
Utilisez la commande find pour chercher le fichier Hello3.txt dans le container.

```
docker container start -i test3
find / -name Hello3b.txt
```

```
/documents/Hello3b.txt
```

Notre fichier est bien visible dans notre container. Le point de montage du volume étant **/documents** . Si vous regardez dans le résultat de l'inspection précédente, vous verrez que cela correspondait à l'entrée "**Destination**".

Si vous lister le contenu du dossier **/Documents** , vous voyez bien votre fichier **Hello3b.txt**
Mais où est donc notre fichier **Hello3a.txt** ? Effectuons une nouvelle recherche find pour le découvrir :

```
find / -name Hello3a.txt
```

```
/Hello3a.txt
```

Notre fichier est dans la racine du container, donc en dehors du point de montage. C'est la raison pour laquelle lors de notre test2, les fichiers Hello2.txt n'avait pas survécu à la suppression du container.

Pas encore convaincu ? **Quitter** le container **test3** avec un **exit** et vérifions où sont nos deux fichiers dans l'arborescence de Docker

```
find /var/lib/docker -name Hello*
```

```
/var/lib/docker/overlay2/587c977d153217de3066b4475aa723d167ecbf842faa6286b686e6526af4ab5a/diff/Hello3a.txt
/var/lib/docker/volumes/2d79cdcd82adfc87442f1c2b6022d52af853f61ddb568158d7144716efc534bb/_data/Hello3b.txt
```

Nos deux fichiers sont bien présents, mais **Hello3a.txt est dans le dossier overlay2** (donc une Layer Container) et le fichier **Hello3b.txt est dans un volume** . Supprimons maintenant le container et recommençons l'opération.

```
docker container rm test3
find /var/lib/docker -name Hello*
```

```
/var/lib/docker/volumes/2d79cdcd82adfc87442f1c2b6022d52af853f61ddb568158d7144716efc534bb/_data/Hello3b.txt
```

Le fichier **Hello3b.txt** est toujours présent car conservé dans volume.
Le fichier **Hello3a.txt** a été supprimé en même temps que le container.

Si l'on revient à la création de note container et surtout à l'option : **-v documents**
que nous avons utilisée : **documents** désignait donc ici le point de montage dans le container et non le nom du volume . Le volume a donc juste été identifié par un ID comme nous avons pu le voir avec le résultat de l'inspection. Ce qui je le consent , n'est pas très pratique. En effet si nous voulions réutiliser notre volume avec un nouveau container , il faudrait taper ceci :

```
docker container run -ti --name test4 -v ID_VOLUME:/documents alpine
```

Si vous lancez cette commande avec le bon ID (reprenez l'inspection précédente) vous retrouverez votre fichier **Hello3b.txt** dans le dossier **/documents** du container.

Si nous voulons donné un nom au volume et choisir correctement le point de montage il vaut donc mieux utiliser la commande :

```
docker container run -d --name test5 -v docs:/documents alpine
```

Dans ce cas un **volume** nommé **docs** sera créé dans le dossier **/var/lib/docker/volumes** et le point de montage sera bien **/documents** dans le container.

CREATION DE VOLUMES DANS UN DOCKERFILE

Une autre possibilité est la déclaration des volumes directement dans un **Dockerfile** et donc dans l'image. Reprenons par exemple, le fichier **Dockerfile** de l'exercice final du TP précèdent et déclarons un volume pour héberger nos pages, donc ayant comme point de montage : **/var/www/html**

```
FROM debian:buster-slim
ARG DOCUMENTROOT="/var/www/html"
ARG APT_FLAGS="-y -q"
RUN apt update -y && apt install ${APT_FLAGS} apache2
COPY app ${DOCUMENTROOT}
```

```
EXPOSE 80
VOLUME ["${DOCUMENTROOT}"]
WORKDIR ${DOCUMENTROOT}
ENTRYPOINT ["apachectl"]
CMD ["-D","FOREGROUND"]
```

Si nous recréons une image à partir de ce nouveau Dockerfile, puis un container à partir de cette image, un volume sera alors automatiquement monté sur le point de montage **/var/www/html.**

Créez une nouvelle image nommé mmisite:2.0 en partant de ce Dockerfile
Créez ensuite un container avec les caractéristiques suivantes :
- Nom de container : **test6**
- Image de base : **mmisite:2.0**
- Mode de démarrage : **Background**
- **Redirection du port 80 du container sur le port 8080 de l'hôte**

Ouvrez ensuite un navigateur avec l'URL : **172.16.VM.ID:8080**
Vous devriez retrouver votre page **index.html** créée lors du dernier TP

Regardons maintenant ou se trouve le volume qui a été créé avec le container.

```
docker container inspect -f "{{ json .Mounts }}" test6 | jq
```

```
[
  {
    "Type": "volume",
    "Name": "9bbf643d5e4fb8c4ee46d60949ea97da6861c061d4aff8c3d3f6e2f0614e1e72",
    "Source": "/var/lib/docker/volumes/9bbf643d5e4fb8c4ee46d60949ea97da6861c061d4aff8c3d3f6e2f0614e1e72/_data",
    "Destination": "/var/www/html",
    "Driver": "local",
    "Mode": "",
    "RW": true,
    "Propagation": ""
  }
]
```

Nous voyons bien ici le volume créé avec le point de montage (Destination) sur le container.
Malheureusement, il n'est pas possible de nommer explicitement le volume avec cette méthode, le nom du volume sera donc un ID créé aléatoirement avec le container.

Placez-vous dans le dossier indiqué par l'entrée **Source** dans le résultat de l'inspection
vous devriez retrouvé la page index.html et la feuille de style copiées dans l'image.

```
cd /var/lib/docker/volumes/...ID../_data
ls -l
```

GESTION DES VOLUMES

Comme pour les containers et les images, Docker offre aussi la possibilité de gérer complément les volumes présents sur la machine. La syntaxe sera : docker volume COMMAND

ls	Lister les volumes
create	Créer un volume
rm	Supprimer un volume
inspect	Inspecter un volume
prune	Supprimer tous les volumes orphelins

Comme d'habitude, commençons par le plus simple lister les volumes existants :

```
docker volume ls
```

```
DRIVER              VOLUME NAME
local               cc0a6e0853c3faa07ec72b27e9b3b87fc34597f274bdfd4195737e199362c5ed
local               de19db09b721c78293ffe118e343720d13a9eb7db1d6620adca5963900252a4e
local               docs
local               f82aaa99ffc8a87ea8684dfed7d10500caff40c47e2ddd960253d543112988e0
local               fcf028ac494c480cf62a3eb925ea47a7f85c6efbbe1adf31f09937af6f8a6ca2
```

En fonction des différents tests et des manipulations que nous avons effectuées depuis les premiers TP sous Docker, la commande vous retournera une liste plus ou moins longue de container. Vous remarquerez qu'ils utilisent un driver nommé **local** . Ce driver définit ou sont situés les volumes. Ici ils sont bien sur stockés en local sur la machine hôte (votre VM), mais il existe d'autres drivers qui permettent de stocker les volumes sur des machines distantes en utilisant des protocoles comme NFS ou même SSH.

Si nous inspectons un de ces volumes, par exemple documents :

```
docker volume inspect documents
```

```
[
    {
        "CreatedAt": "2020-05-17T22:48:10+02:00",
        "Driver": "local",
        "Labels": null,
        "Mountpoint": "/var/lib/docker/volumes/documents/_data",
        "Name": "documents",
        "Options": null,
        "Scope": "local"
    }
]
```

Nous retrouvons les informations sur son emplacement réel sur notre disque ("Mountpoint"), sa date de création et le driver utilisé (**local** par défaut)

Vous pouvez aussi créer vos propres volumes . Vous pourrez ensuite raccrocher ces volumes à des containers avec l'option -v lors de leur création.

```
docker volume create html
docker container run -d --name test7 -v html:/usr/local/apache2/htdocs -p 8081:80 httpd
```

Ici, nous créons un container avec l'image officielle httpd (Apache) et montons le volume html dans le dossier /usr/local/apache2/docs . Ce répertoire est le dossier par défaut des pages web pour cette image. Si vous créez une page index.html dans le dossier /var/lib/docker/volumes/html/_data vous devriez pouvoir l'afficher sur votre navigateur avec l'URL : **172.16.VM.ID:8081**

```
cd /var/lib/docker/volumes/html/_data
nano index.html
```

Avec cette méthode, vous avez une plus grande maitrise de vos volumes, puisque vous les nommez en les créant et pouvez les attacher aux points de montage désirés dans les containers.

Pour finir, il est important de régulièrement faire un peu de ménage dans les volumes. Puisqu'ils ne sont pas supprimés avec les containers, ils peuvent rapidement prendre beaucoup de place sur votre disque, surtout si vous faite de nombreux essais. Deux commandes sont à notre disposition : rm et prune

La première vous demande de spécifier le nom du volume à supprimer. Exemple :

```
docker volume documents
```

Notez que le volume ne sera supprimé que si aucun container n'y est associé. Si c'est le cas, il faut d'abord supprimer le ou les containers associés aux volumes
La seconde commande permet tout simplement de supprimer tous les volumes inutilisés

```
docker volume prune
```

```
WARNING! This will remove all local volumes not used by at least one container.
Are you sure you want to continue? [y/N]
```

Même si cette commande peut paraître radicale, il quand même conseillé de l'utiliser régulièrement pour libérer de l'espace disque sur votre machine. Attention toutefois à bien sauvegarder vos éventuelles données qui pourraient se trouver dans un volume et que vous voudriez pouvoir réutilisées.
Pour finir cette partie, faites un peu de ménage dans vos containers, vos images et vos volumes.
Vous pouvez supprimer tous les containers tests que nous avons utilisés ainsi que les volumes associés.

Partie 2 : Docker-Compose

Dernière partie de notre introduction à Docker : docker-compose
Docker compose est un script écrit en python qui va nous permettre de créer des stacks (piles) de containers en utilisant un seul fichier et une seule commande. L'ensemble formant une application complète avec toutes les interactions nécessaires entre les différents containers.

Pour illustrer cette fonctionnalité, nous reprendrons notre exemple wordpress, mais cette fois-ci, sans installer mariadb, mais en utilisant deux containers : Un pour apache-php-wordpress (l'image wordpress), l'autre pour le moteur de base de données (une image mysql).

INSTALLATION DE DOCKER COMPOSE

Si le script docker-compose n'est pas installé avec le Docker Engine. Vous devez le télécharger à partir du github de Docker et l'installer sur votre système. La version actuelle est la v**2.2.3**

```
cd /root
wget "https://github.com/docker/compose/releases/download/v2.2.3/docker-compose-linux-x86_64"
mv docker-compose-linux-x86_64 /usr/local/bin/docker-compose
chmod  +x  /usr/local/bin/docker-compose
ln  -s  /usr/local/bin/docker-compose  /usr/bin/docker-compose
```

Pour finir testez votre installation

```
docker-compose --version
```

```
Docker Compose version v2.2.3
```

LE FICHIER docker-compose.yml

Un peu à la façon du Dockerfile, docker-compose utilise un fichier de description pour construire les différents containers de notre stack et les connecter entre eux. Ce fichier ce nomme tout simplement docker-compose.yml . Il est au format yaml, un format de description des données couramment utilisé de nos jours. Vous l'avez peut-être déjà vu dans les cours de développement.

Ce fichier va donc contenir toutes la description de notre stack avec une approche très orientée micro-services. C'est à dire une séparation de chaque service dans un container . Dans notre exemple : Le service d'hébergement dans un container, le service bases de données dans un autre. Le fichier docker-compose.yml va donc être une description de ces services avec , pour chacun d'entre-eux , l'image utilisée, le container à créer etc. ...

Voyons concrètement à quoi va ressembler notre fichier **docker-compose.yml**

```
version: '3.8'
services:
  web:
    image: wordpress:5.4.1-php7.4-apache
    container_name : wordpress
    restart: always

    ports:
      - 8085:80

    depends_on:
      - db

  db:
    image: mariadb:10.4
    container_name: mariadb
    restart: always

    volumes:
      - db-volume:/var/lib/mysql

    environment:
      MYSQL_ROOT_PASSWORD: 123456
      MYSQL_DATABASE: demowp
      MYSQL_USER: demowp
      MYSQL_PASSWORD: 123456

volumes:
  db-volume:
```

Avant de créer le fichier et de l'utiliser, quelques explications sur le contenu du fichier :

version: '3.8' — Cette information est obligatoire, elle dépend de la version du Docker Engine installée sur la machine qui va exécuter le docker-compose. Pour connaitre les versions : https://docs.docker.com/compose/compose-file/

Ensuite, nous retrouvons la description de nos **services : web** et **db**
Pour chacun des services, nous avons :

image: — qui désigne l'image utilisée pour le container de ce service
container_name: — le nom du container qui sera créé pour ce service
restart: always — qui garantira que ce service sera toujours actif
En cas d'arrêt, le container sera relancé automatiquement

Puis viennent des options facultatives qui sont liées au besoin du service :
Pour le service web, nous avons par exemple :

ports: **8085:80** — qui indique qu'il faut démarrer le service en redirigeant le port 80 du container vers le port 8085 de l'hôte

depends_on : **db** — qui indique clairement que ce service dépend du service base de données que nous avons appelé **db**

Pour le service base de données (db) :

volumes: db-volume:/var/lib/mysql — Ici, nous indiquons le point de montage. Le volume étant déclaré dans la section volumes du fichier.

environment: — ici nous surchargeons les variables d'environnement du container avec les informations de configuration nécessaire pour créer une base de données et un utilisateur dans mariadb. Ce sont ces informations que nous réutiliserons lorsque nous finaliserons l'installation de wordpress.

Pour finir, nous trouvons en fin de fichier :

volumes: db-volume — Le volume db-volume sera créé pour ensuite être monté sur le dossier indiqué dans le service db.

L'ensemble du fichier forme donc la description complète de notre stack applicative.
Avant de le créer, quelques recommandations :

- Attention à bien respectez l'indentation des différents arguments
- L'indentation est faite par 4 espaces (et non une tabulation)
- Le fichier doit toujours commencer par l'indication de la version.

Pour continuer créez le fichier et vérifier sa syntaxe

```
nano docker-compose.yml
docker-compose config
```

La commande docker-compose config doit vous renvoyer la description de vos services.
En cas d'erreur (mots-clés inconnus, mauvaise indentation, etc..) elle vous renvoie une erreur indiquant la ligne ou le mot posant un problème. Corriger votre fichier si nécessaire et relancer config pour vérifier vos modifications. **Pas la peine de continuer, si une erreur persiste**.

Exemple d'erreur :

```
ERROR: yaml.parser.ParserError: while parsing a block mapping
  in "./docker-compose.yml", line 3, column 5
expected <block end>, but found '<block mapping start>'
  in "./docker-compose.yml", line 9, column 9
```

ATTENTION : Ici Line 3 désigne un problème dans la description de l'entrée de la ligne (dans notre cas web:), par une erreur sur la ligne elle-même. Par exemple, l'erreur ci-dessus était provoqué par une mauvaise indentation du mot clé ports.

Le résultat de la commande si tout va bien :

```
services:
  db:
    container_name: mariadb
    environment:
      MYSQL_DATABASE: demowp
      MYSQL_PASSWORD: 123456
      MYSQL_ROOT_PASSWORD: 123456
      MYSQL_USER: demowp
    image: mariadb:10.4
    restart: always
    volumes:
    - db-volume:/var/lib/mysql:rw
  web:
    container_name: wordpress
    depends_on:
    - db
    image: wordpress:5.4.1-php7.4-apache
    ports:
    - published: 8085
      target: 80
    restart: always
version: '3.8'
volumes:
  db-volume: {}
```

Notre fichier étant correct, nous allons pouvoir passer à la construction et au lancement de notre stack. La commande à utiliser est simplement :

```
docker-compose up -d
```

Docker va alors faire un pull des images nécessaires et construire les deux containers décrit dans notre image, ainsi que le volume pour la base de données. Vérifions tout cela

Les containers :

```
docker container ls
```

```
CONTAINER ID   IMAGE          COMMAND                  CREATED              STATUS              PORTS                    NAMES
4862dd6bf222   wordpress:5.4..  "docker-entrypoint.s..."  About a minute ago   Up About a minute   0.0.0.0:8085->80/tcp     wordpress
d3c8db8f470c   mariadb:10.4    "docker-entrypoint.s..."  About a minute ago   Up About a minute   3306/tcp                 mariadb
```

Notez le port **3306** qui est exposé sur le container mariadb. C'est ce port qui va permettre à notre container wordpress, d'accéder à la base données.

Les volumes:

```
docker volume ls
```

```
local               root_db-volume
```

Notez que le volume a été préfixé avec le compte utilisateur qui a lancé docker-compose. Dans mon cas, je l'ai lancé en tant que root. Si vous l'avez lancé avec votre compte utilisateur il sera préfixé avec le nom de votre utilisateur.

Si vous regardez dans le dossier **/var/lib/docker/volumes/root_db-volume/_data** vous y verrez un dossier demowp qui correspond à notre base de données. Pour l'instant ce dossier ne contient pas grand-chose, car nous n'avons pas encore créé les tables pour wordpress. Elles seront créées après l'installation de wordpress.

```
/var/lib/docker/volumes/root_db-volume/_data/demowp ; ls -l
```

Tout est en place, finalisons notre installation Wordpress.
Connectez-vous sur l'URL : **172.16.VM.ID:8085**
Et terminez l'installation de wordpress avec les options suivantes :
Base de données : **demowp**
Utilisateur : **demowp**
Mot de passe **123456**
Hôte de base de données : **mariadb**:**3306**

Ici, nous indiquons à Wordpress (qui fonctionne sur notre container web) qu'il doit aller chercher sa base données sur le **container mariadb** en utilisant le **port 3306**.

Si vous avez tout correctement saisi, votre site wordpress devrait fonctionner.
Si ce n'est pas le cas, relisez cette partie et vérifier bien ce que vous avez saisie comme information dans votre fichier docker-compose.yml , ainsi que dans le formulaire d'installation de wordpress.

Si nous retournons, dans le dossier du volume root_db-volume nous devrions y avoir nos tables wordpress.

```
ls var/lib/docker/volumes/root_db-volume/_data/demowp#
```

```
db.opt    wp_comments.frm  wp_links.ibd    wp_postmeta.frm  wp_posts.ibd     wp_term_relationships.frm  wp_terms.ibd
wp_usermeta.frm  wp_users.ibd wp_commentmeta.frm  wp_comments.ibd  wp_options.frm  wp_postmeta.ibd  wp_termmeta.frm
wp_term_relationships.ibd  wp_term_taxonomy.frm  wp_usermeta.ibd wp_commentmeta.ibd  wp_links.frm     wp_options.ibd
wp_posts.frm     wp_termmeta.ibd  wp_terms.frm  wp_term_taxonomy.ibd  wp_users.frm
```

Toutes les tables de wordpress sont là.
Mais où sont les fichiers wordpress ? wp_config, le dossier wp_content etc. .. ?

Inspectons le container wordpress pour voir s'il existe un volume

```
docker container inspect  -f  "{{ json .Mounts}}"  wordpress  |  jq
```

```
[
  {
    "Type": "volume",
    "Name": "f8b5a2d365c675cc0b28741a69107e401a21505d1a11c7fc5c3e3858924864e0",
    "Source": "/var/lib/docker/volumes/f8b5a2d365c675cc0b28741a69107e401a21505d1a11c7fc5c3e3858924864e0/_data",
    "Destination": "/var/www/html",
    "Driver": "local",
    "Mode": "",
    "RW": true,
    "Propagation": ""
  }
]
```

Effectivement , il existe bien un volume pour les données de wordpress. Ce volume est monté sur le dossier /var/www/html du container qui correspond comme vous le savez au répertoire par défaut des pages web pour Apache. Listez le contenu de ce dossier et vous y retrouverez tous les fichiers wordpress habituels.

```
cd /var/lib/docker/volumes/NAME_CONTAINER/_data
ls
```

```
index.php    wp-activate.php     wp-comments-post.php  wp-content   wp-links-opml.php  wp-mail.php      wp-trackback.php
license.txt  wp-admin            wp-config.php         wp-cron.php  wp-load.php        wp-settings.php  xmlrpc.php
readme.html  wp-blog-header.php  wp-config-sample.php  wp-includes  wp-login.php       wp-signup.php
```

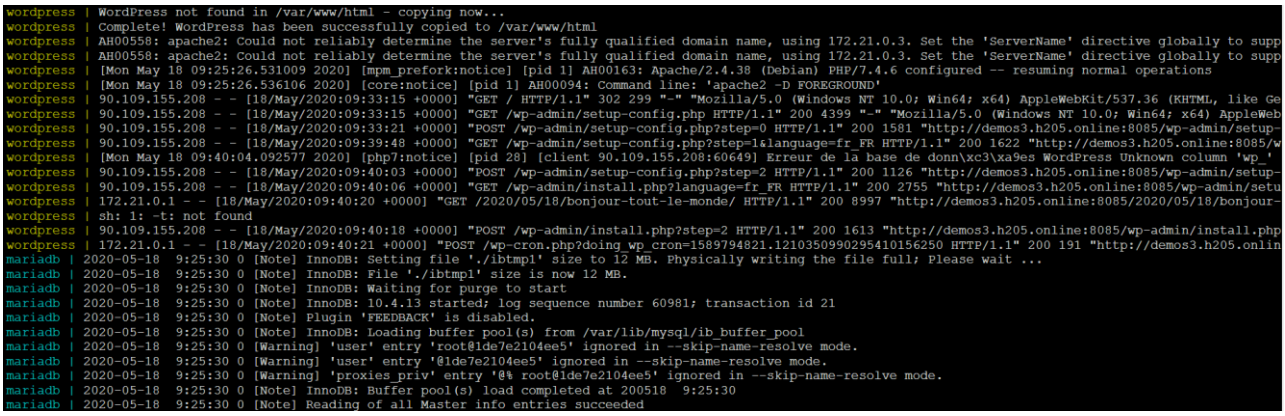
Pour terminer cette partie, voyons encore quelques commandes de docker-compose

docker-compose ps

```
Name        Command                        State        Ports
-------------------------------------------------------------------------
mariadb     docker-entrypoint.sh mysqld    Up           3306/tcp
wordpress   docker-entrypoint.sh apach ...  Up           0.0.0.0:8085->80/tcp
```

Qui nous liste l'état des containers de note stack, ainsi que les ports exposés sur nos containers

docker-compose logs



Qui nous affiche tous les logs écrits sur nos containers depuis leur création. On y voit donc aussi bien l'installation de wordpress, que la création des bases des données, mais aussi les accès fait sur votre site.

Et pour terminer le tout, une dernière commande pour stopper toute la stack d'un coup .

docker-compose down

```
Stopping wordpress ... done
Stopping mariadb   ... done
Removing wordpress ... done
Removing mariadb   ... done
Removing network root_default
```

Notez que docker-compose **arrête** les containers et les **supprime**. Il supprime aussi le réseau qui avait été créé entre les deux containers (**network_root_default)**. Nous reviendrons sur cet aspect dans le prochain TP. Il reste bien sur les images qui ont servi à créer nos containers

docker image ls

```
REPOSITORY   TAG                 IMAGE ID       CREATED       SIZE
mariadb      10.4                9c1f27148b1f   2 days ago    357MB
wordpress    5.4.1-php7.4-apache 58546b852cd9   2 days ago    544MB
```

et les volumes (et nos données persistantes)

docker volume ls

```
DRIVER    VOLUME NAME
local     f8b5a2d365c675cc0b28741a69107e401a21505d1a11c7fc5c3e3858924864e0
local     root_db-volume
```

mais pas nos containers

```
docker container ls -a
```

```
CONTAINER ID   IMAGE          COMMAND                  CREATED          STATUS                      PORTS                  NAMES
77117f0c62bf   httpd          "httpd-foreground"       4 hours ago      Up 4 hours                  0.0.0.0:8082->80/tcp   test8
7425d3a6a514   httpd          "httpd-foreground"       4 hours ago      Up 4 hours                  0.0.0.0:8081->80/tcp   test7
613c4ce1bf86   mmisite:2.0    "apachectl -D FOREGR..."   4 hours ago    Exited (137) 3 hours ago                           test6a
15985f9987ea   mmisite:2.0    "apachectl -D FOREGR..."   4 hours ago    Up 4 hours                  80/tcp                 test6
```

Vous ne verrez ici que les containers qui n'ont pas été supprimés. Peut-être n'avez plus rien.

Partie 3 : Exercice

Dans l'état actuel, si vous relancez votre docker-compose et reconstruisez votre stack que se passe-t-il du point de vue de wordpress ? Récupérez-vous votre site dans l'état ou faut-il relancer l'installation ?

Essayons :

```
docker-compose up -d
```

Revenez sur le site avec l'URL : **172.16.VM.ID:8085 (ou lynx localhost :8085)**
Vous devez relancer l'installation de wordpress !!!!

Seriez-vous dire pourquoi ?

EXERCICE :
Corrigez le problème pour faire en sorte que l'on puisse arrêtez
et relancer notre stack sans avoir à refaire l'installation.
Créez un dossier **/root/exo**
Faites une copie de votre fichier **docker-compose.yml** final dans **/root/exo**

Pour information : La correction sera faite comme suit :
UP du docker-compose à partir du dossier **/root/exo**
Vérification de la persistance de votre site Wordpress avec l'URL : **IP:8085**