DOCKER

Guillaume Rodrigues

[NOM DE LA SOCIETE] [Adresse de la société]

Table des matières

Introduction

Docker est une plateforme qui permet de déployer des applications dans des conteneurs.

Un conteneur est une sorte de "mini-machine virtuelle" qui contient tout ce dont un service a besoin pour fonctionner : le code, les bibliothèques, les dépendances, etc.

Cela permet de rendre les applications portables et facilement déployables, indépendamment de l'environnement dans lequel elles se trouvent.

De plus, Docker nous permet de choisir finement les services et leur version. C'est extrêmement puissant pour nous les développeurs car ça nous permet de monter très simplement des environnements de développement identiques à l'environnement présent en production (on utilise le terme « iso prod » pour parler d'un environnement identique à celui en prod).

Les Objets et concepts clé de Docker

- **Image** : Une image est un modèle immuable (non-modifiable) utilisé pour créer des conteneurs. Elle contient tout ce qu'il faut pour exécuter une application : le code, les runtimes, les bibliothèques, etc.
- **Conteneur** : Un conteneur est une instance d'une image. C'est l'environnement d'exécution isolé où tourne l'application.
- **Dockerfile** : Le Dockerfile est un fichier texte contenant des instructions pour assembler une image Docker.
- **Docker compose** : Un outil permettant de définir et de gérer des applications multi-conteneurs à l'aide d'un fichier YAML (docker-compose.yaml). C'est comme ça que l'on peut définir une « stack technique », c'est-à-dire l'environnement technique composé de plusieurs services/conteneurs (par exemple, un serveur de base de données, un serveur mail, un serveur apache, un serveur FTP, un service de gestion de queue, un service de gestion de cache, ...

Une stack LAMP, c’est quoi ?

La stack LAMP est un ensemble de logiciels open-source utilisés pour développer et déployer des applications web dynamiques. Le terme "LAMP" est un acronyme qui représente les quatre composants principaux de cette stack :

- **L**inux : Le système d'exploitation.
- **A**pache : Le serveur web.
- **M**ySQL/MariaDB : Le système de gestion de base de données.
- **P**HP/Perl/Python : Le langage de script côté serveur.

Composants d’une stack LAMP

Linux

Linux est le système d'exploitation qui sert de fondation à la stack LAMP. Il s'agit d'un système d'exploitation open-source, stable, et largement utilisé pour l'hébergement de serveurs web. Toute la stack repose sur Linux pour l'exécution des autres composants.

Apache

Apache HTTP Server est un serveur web populaire qui gère les requêtes HTTP et sert le contenu web aux utilisateurs. Apache est flexible, puissant et extensible, grâce à ses nombreux modules qui permettent d'ajouter des fonctionnalités supplémentaires (comme le support pour SSL, les redirections, etc.).

Apache écoute les requêtes entrantes (HTTP/HTTPS) sur un serveur et renvoie les pages web correspondantes au client (navigateur).

MySQL / MariaDB

MySQL (ou son fork MariaDB) est un système de gestion de base de données relationnelle (SGBDR) qui permet de stocker et de gérer les données de l'application. Les données sont stockées dans des tables, qui peuvent être interrogées via SQL (Structured Query Language).

Les applications web utilisent MySQL/MariaDB pour stocker des informations comme les comptes utilisateurs, les articles de blog, les produits, etc.

PHP / Perl / Python

PHP est le langage de script le plus couramment utilisé dans la stack LAMP. Il permet de générer dynamiquement le contenu des pages web en interagissant avec la base de données MySQL/MariaDB et en traitant les requêtes des utilisateurs.

PHP est largement utilisé pour créer des sites web dynamiques et interactifs. Alternativement, Perl ou Python peuvent être utilisés, mais PHP est de loin le plus courant dans le contexte LAMP.

Fonctionnement d'une stack LAMP

Lorsqu'un utilisateur visite un site web hébergé sur une stack LAMP, voici ce qui se passe typiquement :

1. **Requête utilisateur** : L'utilisateur entre une URL dans son navigateur, ce qui envoie une requête HTTP au serveur.
2. **Traitement par Apache** : Le serveur Apache reçoit la requête. Si la page demandée est statique (comme une image ou un fichier HTML), Apache la sert directement.
3. **Script PHP** : Si la page demandée est dynamique (comme un script PHP), Apache passe la requête à l'interpréteur PHP.
4. **Interaction avec la base de données** : PHP exécute le script, qui peut inclure des requêtes SQL à MySQL/MariaDB pour récupérer ou manipuler des données.
5. **Génération de la réponse** : PHP utilise les données récupérées pour générer du contenu HTML dynamique.
6. **Retour au client** : Apache renvoie la page HTML générée au navigateur de l'utilisateur.

Docker HUB

Le Docker HUB est une base de données recensant les images mises à disposition et utilisables directement sur Docker.

Bien qu'on ait la possibilité de créer nos propres images « from scratch », on partira le plus souvent d'une image existante sur le docker hub.

https://hub.docker.com/

Recherches (45mn)

- A l'aide d'un terminal (git bash), lancez un conteneur apache via docker (docker run)
- Confirmez que le conteneur fonctionne bien en accédant à votre adresse de loopback (127.0.0.1) ou localhost.
- Obtenez un accès bash (terminal) à l'intérieur de votre conteneur (docker exec). A l'aide d'un éditeur de texte (vim ou nano selon ce qui est installé dans le conteneur), créez un fichier /var/www/html/index.html contenant simplement un squelette HTML et un titre H1 disant « Hello Docker ».
- Accédez à votre adresse de loopback (127.0.0.1) ou localhost et appréciez votre nouvelle page d'accueil.
- Qu'est-ce qu'un volume dans Docker ?

Lancer un conteneur : docker run

La commande « docker run » permet de créer et démarrer un conteneur à partir d'une image.

Exemple

```
# Exécuter un conteneur en mode interactif avec une session bash
docker run -it ubuntu bash

# Exécuter un serveur web NGINX sur le port 80
docker run -d -p 80:80 nginx
```

- **« -it »** : lance le conteneur en mode interactif avec un terminal attaché.
- **« -d »** : lance le conteneur en arrière-plan (détaché).
- **« -p 80 :80 »** : mappe le port 80 du conteneur sur le port 80 de la machine hôte.

Exécuter une commande dans un conteneur : docker exec

La commande « docker exec » permet d'exécuter une commande dans un conteneur qui est déjà en cours d'exécution. Le plus souvent on l'utilise pour exécuter la commande « bash » ce qui nous donne un accès au terminal du conteneur.

Exemple

```
# Ouvrir une session bash dans un conteneur en cours d'exécution
docker exec -it <container_id> bash

# Exécuter une commande à l'intérieur d'un conteneur
docker exec <container_id> ls /app
```

- **« -it »** : ouvre un terminal interactif.
- **« <container_id> »** : l'identifiant ou le nom du conteneur.

Gestion du réseau

Docker permet aux conteneurs de communiquer entre eux et avec l'extérieur via différentes options de réseau.

- **Bridge** : Réseau par défaut où les conteneurs connectés peuvent communiquer entre eux via leurs IPs.
- **Host** : Le conteneur partage directement le réseau de la machine hôte (équivalent du NAT sur Virtualbox).
- **Overlay** : Permet de connecter des conteneurs sur plusieurs hôtes Docker (utile pour les clusters).

Exemple

```
# Lister les réseaux existants
docker network ls

# Créer un réseau bridge
docker network create my-bridge-network

# Lancer un conteneur en l'attachant à un réseau spécifique
docker run -d --network=my-bridge-network --name=myapp nginx
```

Gestion des volumes

Rôle des volumes

Les volumes dans Docker sont utilisés pour persister les données des conteneurs. Par défaut, les données créées à l'intérieur d'un conteneur sont éphémères : lorsque le conteneur est supprimé, les données le sont aussi. Les volumes permettent de contourner cette limitation en stockant les données de manière durable sur le système de fichiers de l'hôte ou un autre système de stockage persistant.

Utilités principales des volumes :

- **Persistance des données** : Conserver les données indépendamment du cycle de vie du conteneur.
- **Partage de données entre conteneurs** : Plusieurs conteneurs peuvent partager et accéder aux mêmes données.
- **Séparation des données et des applications** : Permet de gérer les données de manière indépendante par rapport à l'application qui les utilise.

Fonctionnement des volumes

Les volumes sont gérés par Docker et peuvent être montés dans un ou plusieurs conteneurs. Voici les différentes façons de gérer les volumes dans Docker :

Volumes nommés

Docker gère leur emplacement physique. Ils sont créés avec la commande « docker volume create » et montés dans un conteneur avec l'option « -v » ou « --mount ».

Exemple

```
# Créer un volume nommé
docker volume create mydata

# Lancer un conteneur en y attachant un volume
docker run -d -v mydata:/app/data myimage
```

Bind Mounts

Permettent de mapper un répertoire ou fichier de l'hôte directement dans le conteneur. Ils sont spécifiés avec un chemin absolu ou relatif de l'hôte.

Exemple

```
# Lancer un conteneur en utilisant un bind mount
docker run -d -v /path/on/host:/app/data myimage
```

Volumes anonymes

Créés sans nom explicite, ils sont généralement utilisés pour des cas temporaires ou spécifiques.

Commandes utiles

- **Lister les volumes** : « docker volume ls »
- **Inspecter un volume** : « docker volume inspect mydata »
- **Supprimer un volume** : « docker volume rm mydata »

Mapping de ports

Rôle du mapping de ports dans Docker

Le mapping de ports dans Docker permet de rediriger un port de l'hôte vers un port du conteneur. Cela permet aux services exécutés à l'intérieur du conteneur d'être accessibles depuis l'extérieur (par exemple, depuis un navigateur ou une API).

Utilités principales du mapping de ports :

- **Accéder à des services conteneurisés** : Rendre un service comme un serveur web accessible via l'IP de l'hôte.
- **Séparation des services** : Utiliser différents ports pour différents services sur le même hôte.
- **Développement et test** : Permet de tester des services locaux sans changer les configurations de l’hôte.

Fonctionnement du mapping de ports

Lorsque vous démarrez un conteneur, vous pouvez spécifier quel(s) port(s) du conteneur doit être accessible via un port de l'hôte.

Syntaxe du mapping de ports

« -p <port_hote>:<port_conteneur> » : Mappe le port <port_conteneur> du conteneur vers le port <port_hote> de l'hôte.

Exemples

```
# Démarrer un serveur web sur le port 8080 de l'hôte
docker run -d -p 8080:80 nginx

# Accéder à une application Node.js sur le port 3000
docker run -d -p 3000:3000 node-app
```

- Dans le premier exemple, le port 80 du conteneur (utilisé par Nginx) est accessible via le port 8080 de l'hôte.
- Dans le second exemple, le port 3000 du conteneur (utilisé par une application Node.js) est accessible via le port 3000 de l'hôte.

Commande utile pour vérifier le mapping de ports

« docker port <container_id> » : Affiche les ports mappés d'un conteneur spécifique.

Exercice

Vous utiliserez exclusivement la commande "docker run" pour cet exercice. L'idée générale est de déployer 2 conteneurs et les faire communiquer ensemble. Pour qu'ils puissent communiquer ensemble, ils doivent être **sur le même réseau**.

- La 1ère étape sera donc de créer un réseau (network)
- A l'aide de la doc du docker hub, déployer sur le réseau précédemment créé un container "phpmyadmin" (outil en ligne d'administration de base de données).
- On y accédera par le port 9003 => Gérez le mapping de ports. La page du hub : https://hub.docker.com/_/phpmyadmin
- Sur le même network, déployez un container mariaDB (un Système de Gestion de Base de Données Relationnelles : SGBDr) : https://hub.docker.com/_/mariadb
- Avec un navigateur et après une éventuelle configuration, accédez à l'interface de phpmyadmin (http://localhost:9003/) et constatez que vous pouvez bien administrer votre base MariaDB

Fichier Dockerfile

Un **Dockerfile** est un script textuel contenant une série d'instructions pour construire une image Docker. Il définit les étapes nécessaires pour créer une image, qui pourra ensuite être utilisée pour lancer un ou plusieurs conteneurs. En d'autres termes, un Dockerfile est une recette pour construire une image Docker de manière automatisée.

Rôle du Dockerfile

Le rôle principal d'un Dockerfile est de permettre la création d'images Docker reproductibles, personnalisées et optimisées. Avec un Dockerfile, vous pouvez spécifier :

- Le système d'exploitation de base (comme Ubuntu ou Alpine).
- Les logiciels à installer (comme Apache, Node.js, etc.).
- La configuration des services et des applications.
- Les fichiers à inclure dans l'image.
- Les commandes à exécuter lors du démarrage du conteneur.

Structure et fonctionnement

Un Dockerfile est composé de plusieurs instructions, chacune jouant un rôle spécifique dans la création de l'image Docker.

Instructions de base

Voici les instructions les plus courantes et leur rôle :

- **FROM**
  - **Rôle** : Définit l'image de base à partir de laquelle l'image Docker sera construite. C'est généralement la première instruction d'un Dockerfile.
  - **Exemple** : « FROM ubuntu:20.04 »
- **LABEL**
  - **Rôle** : Ajoute des métadonnées à l'image, comme des informations sur l'auteur ou la version.
  - **Exemple** : « LABEL maintainer= "youremail@example.com" »
- **RUN**
  - **Rôle** : Exécute une commande dans l'image pendant la construction. Utilisé pour installer des paquets, configurer le système, etc.
  - **Exemple** : « RUN apt-get update && apt-get install -y nginx »
- **COPY ou ADD**
  - **Rôle** : Copie des fichiers ou des répertoires du système hôte vers l'image Docker. « ADD » peut aussi extraire des archives compressées et télécharger des fichiers depuis une URL.

    - **Exemple** : « COPY ./myapp /app »
- **WORKDIR**
    - **Rôle** : Définit le répertoire de travail pour les instructions suivantes (RUN, CMD, ENTRYPOINT, etc.).
    - **Exemple : «** WORKDIR /app »
- **CMD**
    - **Rôle** : Spécifie la commande par défaut à exécuter lorsque le conteneur démarre. Cette commande peut être remplacée par des arguments passés lors de l'exécution du conteneur.
    - **Exemple** : « CMD ["nginx", "-g", "daemon off;"] »
- **ENTRYPOINT**
    - **Rôle** : Définit une commande qui sera toujours exécutée lors du démarrage du conteneur, même si d'autres arguments sont passés.
    - **Exemple** : « ENTRYPOINT ["python3", "app.py"] »
- **EXPOSE**
    - **Rôle** : Indique quel port du conteneur doit être exposé à l'extérieur, bien que cette instruction ne fasse pas de mapping de ports par elle-même.
    - **Exemple** : « EXPOSE 80 »
- **ENV**
    - **Rôle** : Définit des variables d'environnement qui seront utilisées dans le conteneur.
    - **Exemple** : « ENV APP_ENV=production »
- **VOLUME**
    - **Rôle** : Crée un point de montage de volume dans le conteneur, qui peut être utilisé pour stocker des données persistantes.
    - **Exemple** : VOLUME ["/data"]
- **ARG**
    - **Rôle** : Définit une variable qui peut être passée lors de la construction de l'image (« docker build --build-arg »).
    - **Exemple** : ARG BUILD_VERSION=1.0
- **USER**
    - **Rôle** : Définit l'utilisateur sous lequel les commandes suivantes seront exécutées.
    - **Exemple** : « USER nonrootuser »

Exemple de fichier Dockerfile

```
# 1. Spécification de l'image de base
FROM node:14

# 2. Création du répertoire de travail
WORKDIR /app

# 3. Copie des fichiers nécessaires
COPY package*.json ./

# 4. Installation des dépendances
RUN npm install --production

# 5. Copie du code de l'application
COPY . .

# 6. Exposition du port de l'application
EXPOSE 8080

# 7. Commande pour démarrer l'application
CMD ["node", "server.js"]
```

Explications

- **« FROM node:14 »** : Utilise l'image officielle Node.js version 14 comme base.
- **« WORKDIR /app »** : Définit le répertoire /app comme le répertoire de travail.
- **« COPY package*.json ./ »** : Copie les fichiers package.json et package-lock.json dans le conteneur.
- **« RUN npm install --production »** : Installe uniquement les dépendances nécessaires pour l'exécution en production.
- **« COPY . . »** : Copie tous les fichiers de l'application dans le répertoire de travail du conteneur.
- **« EXPOSE 8080 »** : Indique que l'application écoute sur le port 8080.
- **« CMD ["node", "server.js"] »** : Démarre l'application Node.js.

Construire et Exécuter une Image à partir d'un Dockerfile

Construction de l'image

Pour construire une image à partir d'un Dockerfile, vous utilisez la commande « docker build ». Par exemple :

```
docker build -t mynodeapp .
```

- **« -t mynodeapp »** : Tag de l'image, nommé ici « mynodeapp ».
- **« . »** : Spécifie le contexte de construction, généralement le répertoire contenant le Dockerfile si ce fichier respecte la convention (nom : Dockerfile).

Exécution de l'image

Une fois l'image construite, vous pouvez lancer un conteneur basé sur cette image :

```
docker run -d -p 8080:8080 mynodeapp
```

- **-d** : Démarre le conteneur en arrière-plan (mode détaché).
- **-p 8080:8080** : Mappe le port 8080 de l'hôte au port 8080 du conteneur.
- **mynodeapp** : Nom de l'image à exécuter.

Best practices pour les fichiers Dockerfile

- **Minimiser la taille des images** : Utiliser des images de base légères (comme Alpine), nettoyer les fichiers temporaires après installation.
- **Gérer les couches efficacement** : Chaque instruction crée une nouvelle couche. Regrouper les instructions liées pour réduire le nombre de couches.
- **Sécurité** : Ne pas inclure de données sensibles (comme des mots de passe) directement dans le Dockerfile.
- **Utiliser .dockerignore** : Pour exclure certains fichiers du contexte de construction, similaire à .gitignore.

Création d'une Stack avec Docker Compose

Docker Compose est un outil qui permet de définir et gérer des applications multi-conteneurs. Vous pouvez spécifier les services, réseaux, volumes, etc., dans un fichier YAML (**« docker-compose.yaml »**), et les démarrer tous ensemble avec une simple commande.

Le principal avantage de monter sa stack via docker compose est que tous les conteneurs créés seront définis sur le même réseau et pourront donc très facilement communiquer les uns avec les autres, ce qui est indispensable lorsque l'on souhaite monter son environnement de développement avec des services bien compartimentés.

Remarque sur les fichiers YAML

Le format yaml est très utilisé en développement web.

Il possède quelques particularités. Une des plus importantes est que c'est un langage basé sur l'indentation (comme Python). Il faudra donc veiller à respecter l'indentation de façon stricte (2 ou 4 espaces d'indentation).

Pour plus d'informations sur le format yaml :
https://symfony.com/legacy/doc/reference/1_3/fr/02-yaml

Structure de base d'un fichier « docker-compose.yaml »

```
version: '3'  # Version de Docker Compose

services:  # Définition des services (conteneurs)
  web:
    image: nginx
    ports:
      - "8080:80"  # Mapping de ports
    volumes:
      - ./html:/usr/share/nginx/html  # Volume bind mount pour servir les
fichiers HTML
    networks:
      - mynetwork

  db:
    image: mysql:5.7
    environment:
      MYSQL_ROOT_PASSWORD: example  # Variable d'environnement pour MySQL
    volumes:
      - dbdata:/var/lib/mysql  # Volume nommé pour les données MySQL
    networks:
      - mynetwork

volumes:  # Définition des volumes
  dbdata:

networks:  # Définition des réseaux
  mynetwork:
```

Explications

- **Services** : Chaque service correspond à un conteneur ou à un ensemble de conteneurs. Dans l'exemple, il y a un service web pour Nginx et un service db pour MySQL.
- **Volumes** : Le service web utilise un bind mount pour servir les fichiers HTML locaux, tandis que db utilise un volume nommé dbdata pour persister les données de la base de données.
- **Networks** : Les services web et db sont connectés au réseau mynetwork, ce qui leur permet de communiquer entre eux.

Commandes docker compose

- **Démarrer les services** : docker-compose up
    - Cette commande démarre tous les services définis dans le fichier docker-compose.yaml.
- **Démarrer en arrière-plan** : docker-compose up -d
    - Cette commande démarre les services en arrière-plan.
- **Arrêter les services** : docker-compose down
    - Cette commande arrête et supprime tous les conteneurs, réseaux, et volumes définis par le fichier docker-compose.yaml.
- **Afficher les logs** : docker-compose logs
    - Affiche les logs de tous les services gérés par Docker Compose.
- **Lister les services** : docker-compose ps
    - Liste tous les conteneurs créés par Docker Compose.

Commandes docker couramment utilisées

Commandes de base

- **docker run**
  - **Rôle** : Crée et démarre un conteneur à partir d'une image spécifiée.
  - **Exemple** : « docker run -it ubuntu bash »
- **docker ps**
  - **Rôle** : Affiche la liste des conteneurs en cours d'exécution.
  - **Exemple** : « docker ps »
- **docker ps -a**
  - **Rôle** : Affiche la liste de tous les conteneurs, y compris ceux qui sont arrêtés.
  - **Exemple** : « docker ps -a »
- **docker container ls**
  - **Rôle** : Liste les conteneurs en cours d'exécution (équivalent à docker ps).
  - **Exemple** : » docker container ls »
- **docker images**
  - **Rôle** : Affiche la liste des images Docker disponibles localement.
  - **Exemple** : « docker images »
- **docker pull**
  - **Rôle** : Télécharge une image depuis un registre Docker (comme Docker Hub).
  - **Exemple** : « docker pull nginx »
- **docker build**
  - **Rôle** : Construit une image Docker à partir d'un Dockerfile.
  - **Exemple** : « docker build -t myapp . »
- **docker exec**
  - **Rôle** : Exécute une commande dans un conteneur en cours d'exécution.
  - **Exemple** : « docker exec -it <container_id> bash »

- **docker start**
  - **Rôle** : Démarre un conteneur arrêté.
  - **Exemple** : « docker start <container_id> »
- **docker stop**
  - **Rôle** : Arrête un conteneur en cours d'exécution.
  - **Exemple** : « docker stop <container_id> »
- **docker restart**
  - **Rôle** : Redémarre un conteneur.
  - **Exemple** : « docker restart <container_id> »
- **docker rm**
  - **Rôle** : Supprime un conteneur arrêté.
  - **Exemple** : « docker rm <container_id> »
- **docker rmi**
  - **Rôle** : Supprime une image Docker.
  - **Exemple** : « docker rmi <image_id> »
- **docker logs**
  - **Rôle** : Affiche les logs d'un conteneur.
  - **Exemple** : « docker logs <container_id> »
- **docker push**
  - **Rôle** : Envoie une image à un registre Docker (comme Docker Hub).
  - **Exemple** : « docker push myrepo/myimage »
- **docker tag**
  - **Rôle** : Associe un tag à une image Docker pour un dépôt spécifique.
  - **Exemple** : « docker tag <image_id> myrepo/myimage:v1.0 »
- **docker inspect**
  - **Rôle** : Retourne les détails d'un conteneur ou d'une image sous forme de JSON.
  - **Exemple** : « docker inspect <container_id> »

- **docker commit**
  - **Rôle** : Crée une nouvelle image à partir d'un conteneur modifié.
  - **Exemple** : « docker commit <container_id> myrepo/myimage:v2 »
- **docker cp**
  - **Rôle** : Copie des fichiers ou dossiers entre le conteneur et le système hôte.
  - **Exemple** : « docker cp <container_id>:/path/in/container /path/on/host »

Commandes réseau

- **docker network ls**
  - **Rôle** : Liste tous les réseaux Docker.
  - **Exemple** : « docker network ls »
- **docker network create**
  - **Rôle** : Crée un nouveau réseau Docker.
  - **Exemple** : « docker network create mynetwork »
- **docker network inspect**
  - **Rôle** : Retourne les détails d'un réseau sous forme de JSON.
  - **Exemple** : « docker network inspect mynetwork »
- **docker network connect**
  - **Rôle** : Connecte un conteneur à un réseau existant.
  - **Exemple** : « docker network connect mynetwork <container_id> »
- **docker network disconnect**
  - **Rôle** : Déconnecte un conteneur d'un réseau.
  - **Exemple** : « docker network disconnect mynetwork <container_id> »

Commandes de volume

- **docker volume ls**
  - **Rôle** : Liste tous les volumes Docker.
  - **Exemple** : « docker volume ls »
- **docker volume create**
  - **Rôle** : Crée un nouveau volume Docker.
  - **Exemple** : « docker volume create myvolume »
- **docker volume rm**
  - **Rôle** : Supprime un volume Docker.
  - **Exemple** : « docker volume rm myvolume »
- **docker volume inspect**
  - **Rôle** : Retourne les détails d'un volume sous forme de JSON.
  - **Exemple** : « docker volume inspect myvolume »

Commandes Docker Compose

- **docker-compose up**
  - **Rôle** : Démarre les services définis dans le fichier docker-compose.yaml.
  - **Exemple** : « docker-compose up »
- **docker-compose down**
  - **Rôle** : Arrête et supprime les conteneurs, réseaux, volumes créés par docker-compose up.
  - **Exemple** : « docker-compose down »
- **docker-compose build**
  - **Rôle** : Construit ou reconstruit les services définis dans le fichier docker-compose.yaml.
  - **Exemple** : « docker-compose build »
- **docker-compose ps**
  - **Rôle** : Liste les conteneurs gérés par Docker Compose.
  - **Exemple** : « docker-compose ps »

- **docker-compose logs**
    - **Rôle** : Affiche les logs des services gérés par Docker Compose.
    - **Exemple** : « docker-compose logs »

Commandes avancées

- **docker save**
    - **Rôle** : Exporte une image Docker vers un fichier tarball.
    - **Exemple** : « docker save -o myimage.tar myrepo/myimage:v1 »
- **docker load**
    - **Rôle** : Importe une image Docker à partir d'un fichier tarball.
    - **Exemple** : « docker load -i myimage.tar »
- **docker export**
    - **Rôle** : Exporte le système de fichiers d'un conteneur vers un fichier tarball.
    - **Exemple** : « docker export -o mycontainer.tar <container_id> »
- **docker import**
    - **Rôle** : Crée une image Docker à partir d'un fichier tarball du système de fichiers.
    - **Exemple** : « docker import mycontainer.tar myrepo/myimage:v1 »
- **docker system prune**
    - **Rôle** : Supprime les conteneurs arrêtés, les images inutilisées, les réseaux et les volumes non référencés pour libérer de l'espace.
    - **Exemple** : « docker system prune »
- **docker stats**
    - **Rôle** : Affiche en temps réel les statistiques d'utilisation des ressources des conteneurs (CPU, mémoire, réseau, etc.).
    - **Exemple** : « docker stats »