Docker, cours pour débuter

SOMMAIRE

A) Définition de la conteneurisation ? 🚢

Qu'est-ce qu'un conteneur ?
Imaginons une boîte magique qui contient :
- Une application (ex: site web)
- Ses dépendances (ex: Python 3.10, bibliothèques)
- Sa configuration (ex: variables d'environnement)

Et bien… cette boîte fonctionne DE LA MÊME MANIÈRE sur n'importe quelle machine !
Plus jamais de : "Ça marche sur mon PC, oui, mais pas en production !" 😅
Et ce, quelle que soit la machine !! 😅😅😅

B) Pourquoi utiliser Docker ? 🐳

✅ Isolation : Chaque conteneur est indépendant (pas de conflits de versions).
✅ Portabilité : Fonctionne sur Windows, macOS, Linux, cloud.
✅ Efficacité: Les conteneurs partagent le noyau de l'OS → Léger et rapide.

Pourquoi Linux est-il souvent utilisé ?
🐠 La majorité des images Docker sont basées sur Linux (plus stable, open-source).
🐠 Meilleure gestion des conteneurs (Docker utilise des fonctionnalités natives de Linux).

C) Installation 💻

Linux (Ubuntu)
1. Met à jour la liste des paquets disponibles (comme un catalogue de boutiques)
sudo apt update

>> ne pas confondre avec la commande : sudo apt upgrade (qui permet de mettre à jour tous les paquets disponibles de tous les logiciels).
D'ailleurs pour faire une mise à jour complète, on écrira les deux commandes ensemble :

sudo apt update && sudo apt upgrade -y (&& permet de lier deux opérations en n'exécutant la seconde que si la première passe sans encombres, et l'option -y permet de valider la ligne de commande sans demande de confirmation) */

2. Installer Docker (le logiciel qui gère les conteneurs)
sudo apt install docker.io

3. Démarrer le service Docker :
sudo systemctl start docker

4. Activer Docker au démarrage du système (pour ne pas avoir à le relancer) :
sudo systemctl enable docker

Vérification
Afficher la version de Docker installée (ex: 24.0.7)
docker --version

Exécuter un conteneur de test "hello-world" (comme un "Hello, World!" pour Docker)
docker run hello-world

💡 Aide :
Explication de `docker run hello-world` :
- Docker télécharge l'image "hello-world" depuis Docker Hub (un magasin d'images).
- Crée un conteneur éphémère qui affiche un message de bienvenue.
- BUT : Vérifier que Docker fonctionne correctement.

docker run hello-world # Le conteneur s'arrête immédiatement après avoir affiché le message.

D) COMMANDES DOCKER UTILES 🛠️

Gestion des images

Télécharger l'image Ubuntu 22.04 LTS (version stable à long terme)
docker pull ubuntu:22.04

💡 Pourquoi une LTS (Long-Term Support) ?
→ Mises à jour de sécurité garanties pendant 5 ans.
→ Idéal pour la production !

Liste toutes les images sur votre machine
docker images

Supprimer l'image Ubuntu 22.04 (si vous n'en avez plus besoin)

docker rmi ubuntu:22.04

Gestion des conteneurs
Lancer un conteneur Ubuntu avec un terminal interactif (-it)
docker run -it ubuntu:22.04 /bin/bash

🔍 Aide :
-it = "Interactif + Terminal" → permet de taper des commandes dans le conteneur.
/bin/bash = Ouvre un shell Bash (l'interpréteur de commandes).

Donc pour résumer, la commande docker run -it ubuntu:22.04 /bin/bash permet de lancer un conteneur Ubuntu avec un terminal ET un interpréteur de commandes Bash, ce qui permet d'avoir un terminal opérationnel configuré en Bash, et donc capable d'exécuter des commandes Linux.

Obtenir la liste les conteneurs en cours d'exécution
docker ps

Obtenir la liste TOUS les conteneurs (même arrêtés)
docker ps -a

Redémarrer un conteneur arrêté (remplace <ID> par l'ID du conteneur)
docker start <ID>

💡 Aide :
Où trouver l'ID ? → Colonne "CONTAINER ID" dans `docker ps -a`.
Exemple : `docker start 5a3b8c7d`.

E) Créer une première image (node.js) 🚀

Structure du projet
```
my-node-app/
├── app.js          # Code de l'application
├── package.json    # Dépendances Node.js
└── Dockerfile      # Recette de construction de l'image
```

app.js
```
const http = require('http');
const server = http.createServer((req, res) => {
  res.end('Salut Docker ! 👋');
});
server.listen(3000);
```

Dockerfile

Étape 1 : Choisir l'image de base (Node.js 18 sur Alpine Linux)
FROM node:18-alpine

Étape 2 : Définir le dossier de travail dans le conteneur
WORKDIR /app

Étape 3 : Copier les fichiers locaux vers le conteneur
COPY package.json . # Copie le fichier package.json
COPY app.js . # Copie le fichier app.js
Étape 4 : Installer les dépendances Node.js
RUN npm install

Étape 5 : Exposer le port 3000 (celui utilisé par l'application)
EXPOSE 3000

Étape 6 : Commande à exécuter au démarrage du conteneur
CMD ["node", "app.js"]

Construction et exécution
Dans le dossier my-node-app/
docker build -t my-node-app . # Construit l'image (-t = nom de l'image)

Lance le conteneur en arrière-plan (-d) avec redirection de port
docker run -d -p 3000:3000 my-node-app

✅ Test : Ouvrir http://localhost:3000 → Résultat attendu : on voit normalement le message : "Salut Docker ! 👋"

Erreurs courrantes & solutions 🚑

Problème : "Permission denied" sur Linux ---
Solution :
sudo usermod -aG docker $USER # Ajoute votre utilisateur au groupe "docker"
newgrp docker # Active les changements sans redémarrage

Problème : "Port 3000 already in use" ---
Solution : Changez le port de l'hôte :
docker run -d -p 8080:3000 my-node-app # Accès via http://localhost:8080

Problème : "Image not found" ---
Vérifiez :
1. Avez-vous fait `docker build -t nom-image .` ?
2. L'orthographe du nom de l'image est correcte ?

F) Exercices

Exercice 1 : Nginx

1. Lancer un conteneur Nginx :
   docker run -d -p 8080:80
2. Modifier la page d'accueil :
   - Entrer dans le conteneur : docker exec -it <ID> /bin/bash

- Éditer docker run -d -p 8080:80 -v $(pwd)/html:/usr/share/nginx/html nginx
3. Rafraîchir http://localhost:8080 → le fichier HTML s'affiche !

4. Créer un dossier html avec un index.html personnalisé.

Exercice 2 : Python Flask

1. Créer un fichier app.py :

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def home(): return "Bonjour depuis Flask !"
```

2. Dockerfile :

```
FROM python:3.10
COPY . /app
WORKDIR /app
RUN pip install flask
CMD ["python", "app.py"]
```

Exercice expliqué :

1. Fichier app.py

```
from flask import Flask  # Importe le module Flask

app = Flask(__name__)  # Crée une instance de l'application Flask

@app.route('/')  # Définit la route pour la page d'accueil ("/")

def home():

    return "Bonjour depuis Flask !"  # Affiche ce message quand on visite la page
```

💡 Aide :
Flask est un micro-framework pour créer des applications web en Python.

@app.route('/') signifie : "Quand quelqu'un visite l'URL racine (ex: http://localhost:5000/), exécute la fonction home()".

C'est le strict minimum pour une application web fonctionnelle !

2. Dockerfile

Étape 1 : Image de base

```
FROM python:3.10  # Utilise Python 3.10 officiel
```

Étape 2 : Copie des fichiers locaux

```
COPY . /app  # Copie TOUS les fichiers du dossier courant vers /app dans le conteneur
```

Étape 3 : Définit le répertoire de travail

```
WORKDIR /app  # Toutes les commandes suivantes seront exécutées dans /app
```

Étape 4 : Installation des dépendances

```
RUN pip install flask  # Installe Flask via pip
```

```
# Étape 5 : Commande de démarrage
CMD ["python", "app.py"]  # Lance l'application avec le serveur de développement Flask
```

Pourquoi python:3.10 ? → Garantit que Python 3.10 est utilisé, évitant les conflits de versions.

COPY . /app copie **tous les fichiers** (y compris app.py, templates, etc.).

- En pratique, on utilise souvent un fichier requirements.txt pour gérer les dépendances (voir *Bonnes Pratiques* plus bas).

Construction et Exécution du Conteneur

```
# Depuis le dossier contenant app.py et Dockerfile
docker build -t my-flask-app .  # Construit l'image (-t = nom de l'image)

docker run -p 5000:5000 my-flask-app  # Lance le conteneur
# -p 5000:5000 : Redirige le port 5000 de l'hôte vers le port 5000 du conteneur
```

Résultat :

Ouvrir http://localhost:5000 dans le navigateur.

On doit clairement voir : "Bonjour depuis Flask !" 🎉

G) Bonnes Pratiques à utiliser :

a. Utiliser un fichier requirements.txt :

Créer un fichier requirements.txt avec dedans :

```
flask==3.0.2
```

Modifier le Dockerfile en y copiant :

```
COPY requirements.txt .    # Fichier léger → Cache Docker intact si dépendances inchangées

RUN pip install -r requirements.txt  # Installe toutes les dépendances listées
```

b. Mode Production :

Le serveur de développement Flask (python app.py) n'est **pas adapté pour la production**.

Solution : Utiliser un serveur WSGI comme **Gunicorn** :

```
RUN pip install gunicorn
CMD ["gunicorn", "--bind", "0.0.0.0:5000", "app:app"]  # "app:app" = module:instance_flask
```

c. Optimisation :

Ajoutez .dockerignore pour ignorer les fichiers inutiles :

```
# .dockerignore
__pycache__
*.pyc
.env
```

5. Erreurs Courantes et Solutions

"ModuleNotFoundError: No module named 'flask'"

Vérifier que RUN pip install flask est bien dans le Dockerfile.

L'application ne se lance pas

Vérifiez le port : Flask utilise le port 5000 par défaut.

Modifications non prises en compte
Reconstruire l'image après chaque changement (docker build -t ...).

Résumé :

PC→ Dockerfile → Image Docker → Conteneur Flask (port 5000) → Navigateur

Ce qu'on sait maintenant faire :

1) Créer une application web simple avec Flask

2) Dockeriser l'application pour la rendre portable.

3) Lancer le conteneur et accéder à l'application via le navigateur.

Annexe 1 : Quelques aides en plus :

Vérifier les versions d'Ubuntu disponibles

1. Aller sur Docker Hub : https://hub.docker.com/_/ubuntu
2. Rechercher les tags (ex: 22.04, 24.04, jammy, lunar)

Commandes à connaître

```
docker logs <ID>                # Affiche les logs d'un conteneur
docker exec -it <ID> bash       # Ouvre un shell dans un conteneur en cours
```

Checklist de départ

Docker installé (`docker --version`)
Conteneur hello-world fonctionnel
Première image personnalisée construite

Astuces pour Débutants 🚀

```
docker --help                   # Affiche l'aide générale
docker <commande> --help        # Aide spécifique à une commande (ex: `docker run --help`)
docker version                  # Affiche la version de Docker et du moteur
```

Annexe 2 : Petit Lexique des commandes Docker courantes et utiles :

Gestion des Images

```
docker pull <image:tag>          # Télécharge une image (ex: `docker pull nginx:latest`)
docker build -t <nom-image> .    # Construit une image depuis un Dockerfile (dans le dossier actuel `.`)
docker images                    # Liste toutes les images locales
docker rmi <image>               # Supprime une image (ex: `docker rmi nginx`)
docker image prune               # Nettoie les images inutilisées ou "dangling"
```

Gestion des Conteneurs

```
docker run -d -p <hôte:conteneur> <image>  # Lance un conteneur en arrière-plan (ex: `docker run -d -p 8080:80 nginx`)
docker ps -a                     # Affiche tous les conteneurs (actifs ou arrêtés)
docker start/stop/restart <ID>   # Contrôle l'état d'un conteneur
docker rm <ID>                   # Supprime un conteneur arrêté (ajoutez `-f` pour forcer)
docker logs <ID>                 # Affiche les logs d'un conteneur (ajoutez `-f` pour suivre en temps réel)
docker exec -it <ID> <commande>  # Exécute une commande dans un conteneur en cours (ex: `docker exec -it 5a3b bash`)
```

Docker Compose

```
docker-compose up -d             # Lance les services en arrière-plan (depuis un fichier docker-compose.yml)
docker-compose down              # Arrête et supprime les conteneurs, réseaux, volumes
docker-compose logs              # Affiche les logs des services
docker-compose build             # Reconstruit les images des services
```

Réseaux

```
docker network ls                # Liste les réseaux Docker
docker network create <nom>      # Crée un réseau personnalisé
docker network inspect <nom>     # Affiche les détails d'un réseau
```

Volumes

```
docker volume ls                 # Liste les volumes Docker
docker volume create <nom>       # Crée un volume
docker volume inspect <nom>      # Affiche les infos d'un volume
```

Nettoyage 🧹

```
docker system prune              # Supprime conteneurs arrêtés, réseaux inutilisés, images "dangling"
docker system df                 # Affiche l'espace disque utilisé par Docker
```

Inspecter/Déboguer 🔍

```
docker inspect <ID>              # Affiche toutes les infos d'un conteneur/image/volume
docker stats                     # Affiche l'utilisation CPU/RAM des conteneurs en temps réel
docker top <ID>                  # Affiche les processus en cours dans un conteneur
```

Utiliser Docker pour le déploiement d'une application Java.

Introduction : Pourquoi Docker pour Java ?

Problématiques Résolues

- **"Ça marche sur ma machine !"** : Différences de JDK, OS, versions de bibliothèques.
- **Dépendances complexes** : Configuration de Tomcat, Jetty, ou serveurs d'applications.
- **Scalabilité** : Déploiement rapide de multiples instances.

Avantages Clés

- **Portabilité** : Même environnement de dev à prod.
- **Isolation** : Pas de conflits entre applications.
- **Efficacité** : Les conteneurs démarrent en quelques secondes.

Installer Docker sur Linux, ou sur une machine virtuelle Linux (avec VirtualBox par exemple).

Pour des raisons de sécurité et de facilité de travail, il est vivement recommandé d'effectuer ce travail sous Linux. C'est pourquoi, tous mes exemples seront effectués sous Ubuntu Linux.

Mise à jour des paquets

sudo apt update && sudo apt upgrade -y
(Commande déjà expliquée précédemment)

Installation de Docker

sudo apt install docker.io
sudo systemctl start docker
sudo systemctl enable docker

Vérification

docker --version # Exemple de renvoi de prompt : Docker version 24.0.7

Dockeriser une Application Java

Structure du Projet

```
mon-application-java/
├── src/
├── pom.xml          # Fichier Maven
├── Dockerfile       # Recette de construction
└── docker-compose.yml  # Optionnel pour multi-conteneurs
```

Dockerfile pour une App Maven

Étape 1 : Image de build (Maven + JDK)

```
FROM maven:3.8.6-openjdk-17 AS build
WORKDIR /app
COPY pom.xml .
RUN mvn dependency:go-offline  # Cache les dépendances
COPY src/ ./src/
RUN mvn package -DskipTests    # Build le JAR
```

Étape 2 : Image finale (JRE seul)

```
FROM openjdk:17-jdk-slim
WORKDIR /app
COPY --from=build /app/target/*.jar /app/app.jar
EXPOSE 8080
CMD ["java", "-jar", "app.jar"]
```

Explications :

- **Multi-stage build** : Réduit la taille finale en ne gardant que le JAR.
- **Cache Maven** : Accélère les rebuilds avec dependency:go-offline.
- **JDK vs JRE** : Le JRE est suffisant en production (plus léger).

Lancer l'Application en Production

Construction de l'Image

docker build -t mon-app-java . # Construit l'image

Exécution Simple

docker run -d -p 8080:8080 --name java-app mon-app-java

Options :

- -d : Détaché (arrière-plan).
- --restart unless-stopped : Redémarrage automatique.
- -e "SPRING_PROFILES_ACTIVE=prod" : Passe une variable d'environnement.

Docker Compose pour les Dépendances

Avec Postgresql :

```
version: '3.8'

services:
  app:
    image: mon-app-java
    ports:
      - "8080:8080"
    environment:
      SPRING_DATASOURCE_URL: jdbc:postgresql://db:5432/mydb
      SPRING_DATASOURCE_USERNAME: admin
      SPRING_DATASOURCE_PASSWORD: secret
    depends_on:
      - db

  db:
```

```
  image: postgres:15
  volumes:
   - db_data:/var/lib/postgresql/data
  environment:
   POSTGRES_PASSWORD: secret

volumes:
 db_data:
```

Exécuter le lancement :

```
docker-compose up -d  # Démarre l'app et la base de données
```

Bonnes Pratiques de Production

Sécurité

Utilisateur non-root :

```
FROM openjdk:17-jdk-slim
RUN adduser --system --group appuser
USER appuser  # Exécute l'app avec un utilisateur non-privilégié
```

Analyser les images :

```
docker scan mon-app-java  # Détecte les vulnérabilités (via Snyk)
```

Logs et Monitoring

Journalisation centralisée :

Commande bash : docker run -d --log-driver=syslog --log-opt syslog-address=udp://mon-serveur:514 mon-app-java

Optionnel : Avec Prometheus :

management.endpoints.web.exposure.include=health,metrics,prometheus

Mise à jour sans temps d'arrêt :

```
docker service update --image mon-app-java:v2 my-java-service
```

Intégration Continue (CI/CD)

Pipeline Jenkins

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh 'docker build -t mon-app-java .'
            }
        }
        stage('Test') {
            steps {
                sh 'docker run mon-app-java mvn test'
            }
        }
        stage('Deploy') {
            steps {
                sh 'docker tag mon-app-java registry.mon-domaine.com/mon-app-java:latest'
                sh 'docker push registry.mon-domaine.com/mon-app-java:latest'
            }
        }
    }
}
```

La même chose sur Github Action

```
name: Java CI/CD
on:
  push:
```

```
    branches: [main]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Build and Push
        run: |
          docker build -t mon-app-java .
          docker login -u ${{ secrets.DOCKER_USER }} -p ${{ secrets.DOCKER_PASSWORD }}
          docker push mon-app-java:latest
```

Commandes utiles pour la surveillance

Vérifier les ressources :

```
docker stats  # CPU, mémoire en temps réel
```

Inspecter un conteneur :

```
docker inspect java-app | grep "IPAddress"
```

Outils à utiliser avec pour plus d'efficacité :

Analyse des performances : Cadvisor

Dashboarding : Prometheus + Grafana

Centralisation des logs : ELK Stack

Un exemple avec SpringBoot + Mysql :

Dockerfile :

```
FROM maven:3.8.6-openjdk-17 AS build
```

```
WORKDIR /app

COPY . .

RUN mvn package -DskipTests


FROM openjdk:17-jdk-slim

COPY --from=build /app/target/*.jar app.jar

ENTRYPOINT ["java", "-jar", "app.jar"]
```

Déploiement, avec bash :

```
docker build -t spring-app .

docker run -d -p 8080:8080 --link mysql-db:db spring-app
```

Commandes utiles pour le déploiement d'une app Java :

```
docker logs -f java-app      # Suivre les logs en temps réel

docker exec -it java-app sh  # Accéder au shell du conteneur
docker system prune          # Nettoyer les ressources inutilisées
```