TP_docker_basic

Docker - Prise en main : images et conteneurs

Objectif général

Mettre en place un environnement Docker fonctionnel, créer sa première image, lancer et diagnostiquer des conteneurs, puis comprendre les notions fondamentales (volumes, réseaux, bonnes pratiques de base).

Prérequis

- Poste de travail avec Docker Desktop, Docker Engine ou Colima / Rancher Desktop configuré
- Accès à un shell (bash/zsh) avec `curl`, `git`, `docker`, `docker compose`
- Notions de base en ligne de commande et en Python (script simple)

Compétences visées

- Vérifier et comprendre la configuration d'un moteur Docker
- Construire une image à partir d'un Dockerfile minimal
- Lancer, inspecter et supprimer des conteneurs
- Monter des volumes et publier des ports
- Diagnostiquer les erreurs courantes (build, run, permission)

Fil rouge du TP

Tu vas conteneuriser un mini-service Python (« quotegen ») qui renvoie une citation aléatoire via une API Flask. Chaque étape t'amènera à introduire un concept Docker différent. Le répertoire recommandé est `~/workspace/docker-basic`.

Étape 0 — Préparer l'arborescence

1. Crée le dossier de travail et initialise le dépôt :

```
mkdir -p ~/workspace/docker-basic && cd ~/workspace/docker-basic
git init .
```

2. Ajoute le squelette applicatif :

```
cat > app.py <<'PY'
import random
from flask import Flask, jsonify

QUOTES = [
    "Docker vous donne l'isolement sans la douleur des VM.",
    "Build once, run anywhere (du moins en théorie).",
    "Les couches de Docker sont des caches déguisés.",
]

app = Flask(__name__)


@app.get("/quote")
def quote():
    return jsonify({"quote": random.choice(QUOTES)})


if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
PY

cat > requirements.txt <<'TXT'
flask==3.0.3
gunicorn==22.0.0
TXT
```

```
3. Vérifie que le script fonctionne localement (facultatif mais recommandé) :

   ```bash
   python3 -m venv .venv && source .venv/bin/activate
   pip install -r requirements.txt
   python app.py
```

➜ Ouvre http://127.0.0.1:5000/quote pour t'assurer que l'API répond. Stoppe ensuite le serveur (Ctrl+C) et désactive le venv (`deactivate`).

Étape 1 — Découvrir son moteur Docker

1. Valide la version et le statut :

```
docker version
docker info
```

➜ Note les informations importantes : version du serveur, drivers, nombre d'images/containers, configuration réseau.

2. Explore les commandes de base :

```
docker ps               # conteneurs en cours
docker ps -a            # conteneurs arrêtés compris
docker image ls         # images disponibles
docker volume ls        # volumes existants
```

➜ Capture d'écran ou export texte des sorties pour ton compte-rendu.

Étape 2 — Écrire un Dockerfile minimal

1. Crée un fichier `Dockerfile` avec les instructions de base :

```
# syntax=docker/dockerfile:1
FROM python:3.12-slim

WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
```
```

```
COPY app.py .
EXPOSE 5000

CMD ["gunicorn", "--bind", "0.0.0.0:5000", "app:app"]
```

➜ Justifie chaque instruction dans ton rapport (base image, workdir, copy, etc.).

2. Construis l'image :

```
docker build -t quotegen:1.0.0 .
```

3. Valide la présence de l'image :

```
docker image ls quotegen
```

➜ Note la taille et réfléchis à l'impact des dépendances sur celle-ci.

Étape 3 — Lancer et tester le conteneur

1. Démarre le conteneur en avant-plan :

```
docker run --rm -p 5000:5000 --name quotegen quotegen:1.0.0
```

➜ Observe les logs dans la console. Dans un autre terminal, exécute :

```
curl http://localhost:5000/quote
```

2. Stoppe le conteneur avec `Ctrl+C`. Relance-le en arrière-plan :

```
docker run -d -p 5000:5000 --name quotegen quotegen:1.0.0
docker logs quotegen --tail 5
```

3. Inspecte et supprime :

```
docker inspect quotegen --format '{{json .NetworkSettings.Ports}}' | python -m json.tool
docker stop quotegen
docker container prune
```

➜ Explique la différence entre `stop`, `rm`, `prune`.

Étape 4 — Volumes et persistance

Objectif : extraire la liste des citations vers un volume monté pour pouvoir la modifier sans rebuild.

1. Déplace le contenu dans un dossier `data` :

```
mkdir data
cat > data/quotes.txt <<'TXT'
Docker vous donne l'isolement sans la douleur des VM.
Build once, run anywhere (du moins en théorie).
Les couches de Docker sont des caches déguisés.
TXT
```

2. Modifie `app.py` pour lire depuis `data/quotes.txt`. Exemple :

```
from pathlib import Path

def load_quotes():
    with Path("data/quotes.txt").open() as fh:
        return [line.strip() for line in fh if line.strip()]
@app.get("/quote")
def quote():
    quotes = load_quotes()
    return jsonify({"quote": random.choice(quotes)})
```

3. Mets à jour le `Dockerfile` :

```
COPY data ./data
```

4. Reconstruis l'image (`docker build -t quotegen:1.1.0 .`) et lance-la avec un volume :

```
docker run -d -p 5000:5000 \
  -v "$(pwd)/data/quotes.txt:/app/data/quotes.txt:ro" \
  --name quotegen quotegen:1.1.0
```

5. Édite `data/quotes.txt` localement, ajoute une citation, puis teste l'API.

➜ Documente la différence entre `COPY` (image immuable) et `-v` (montage dynamique).

Étape 5 — Gestion des tags et nettoyage

1. Liste les images et supprime les anciennes :

```
docker image ls quotegen
docker image rm quotegen:1.0.0
```

2. Crée des aliases de tag :

```
docker tag quotegen:1.1.0 quotegen:latest
docker tag quotegen:1.1.0 registry.local:5000/quotegen:1.1.0  # exemple
```

➜ Explique l'intérêt des tags multiples (ex. `:1.1.0`, `:latest`, `:staging`).

3. Inspecte les couches pour comprendre la taille :

```
docker history quotegen:1.1.0
```

➜ Identifie les instructions qui pèsent le plus.

4. Nettoie tout :

```
docker stop quotegen
docker container prune -f
docker image prune -f
docker volume ls
docker volume prune   # (⚠️ confirme bien ce que tu supprimes)
```

➜ Insiste sur la prudence avec `prune` et la différence entre volumes nommés et bind mounts.

Étape 6 — Mini-quiz de validation

Réponds (oralement ou par écrit) :

1. Quelle est la différence entre `ENTRYPOINT` et `CMD` ?
2. Comment inspecter les variables d'environnement d'un conteneur en cours d'exécution ?
3. Comment diagnostiquer un build qui échoue derrière un proxy d'entreprise ?
4. Quelle commande permet de voir l'espace disque occupé par Docker ?
5. Pourquoi `--rm` est-il utile en développement ?

Livrables attendus

- Repository Git ou archive contenant : `Dockerfile`, `app.py`, `requirements.txt`, `data/quotes.txt`
- Fichier `README.md` qui résume la procédure de build et d'exécution
- Rapport de quelques lignes expliquant :
  - les commandes clés utilisées
  - les difficultés rencontrées et leurs résolutions
  - une capture d'écran de l'API en fonctionnement

Aller plus loin (optionnel)

- Ajouter un `docker-compose.yml` pour monter un reverse proxy nginx en front.
- Utiliser un multi-stage build pour réduire la taille de l'image.
- Pousser l'image sur un registre local (`docker run -d -p 5000:5000 registry:2`) ou distant (Docker Hub).
- Écrire un script `Makefile` qui automatise `build/run/test/clean`.