TP_docker_basic_2

Docker - Approfondissement : optimiser et orchestrer sans compose

---

Objectif général

Approfondir les concepts Docker en construisant des images plus efficaces, en gérant plusieurs conteneurs à la main (sans `docker compose`), en configurant réseaux et volumes, et en introduisant des bonnes pratiques de production (healthcheck, utilisateurs non root, scripts d'entrée).

Prérequis

- Avoir réalisé le TP « Docker - Initiation » (ou maîtriser `docker build` / `docker run`)
- Docker fonctionnel, `python3`, `pip`, `curl`, `jq`
- Connaissances basiques en FastAPI ou équivalent (lecture/écriture d'un petit service web)

Compétences visées

- Écrire un Dockerfile multi-stage avec dépendances gelées
- Ajouter un healthcheck et un entrypoint personnalisé
- Créer et utiliser un réseau Docker personnalisé
- Faire communiquer deux conteneurs (API + base de données) sans composer
- Gérer la persistance via volumes et bind mounts
- Manipuler les tags, pousser sur un registre local et nettoyer proprement

---

Fil rouge du TP

Tu vas conteneuriser un service FastAPI (`todo_api`) qui gère une liste de tâches. L'API utilisera d'abord SQLite intégré, puis sera reliée à une base PostgreSQL exécutée dans un second conteneur. Tu automatiseras le démarrage via un script `entrypoint.sh` et tu optimiseras l'image finale.

Arborescence recommandée : `~/workspace/docker-basic-2`.

Étape 0 — Préparer le code de l'API

1. Crée le dossier et récupère le squelette :

```
mkdir -p ~/workspace/docker-basic-2 && cd ~/workspace/docker-basic-2
git init .
cat > app.py <<'PY'
from fastapi import FastAPI
from pydantic import BaseModel
from typing import Dict
import uuid
import os
import sqlite3
from contextlib import closing
from pathlib import Path


DB_PATH = Path(os.getenv("TODO_DB_PATH", "data/todo.db"))
DB_PATH.parent.mkdir(parents=True, exist_ok=True)


def init_db():
    with closing(sqlite3.connect(DB_PATH)) as conn:
        conn.execute(
            "CREATE TABLE IF NOT EXISTS todos (id TEXT PRIMARY KEY, title TEXT, done INTEGER)"
        )
        conn.commit()


def all_todos() -> Dict[str, Dict[str, str]]:
    with closing(sqlite3.connect(DB_PATH)) as conn:
        conn.row_factory = sqlite3.Row
        rows = conn.execute("SELECT * FROM todos").fetchall()
        return {row["id"]: {"title": row["title"], "done": bool(row["done"])} for row in rows}


def add_todo(title: str) -> str:
    todo_id = str(uuid.uuid4())
    with closing(sqlite3.connect(DB_PATH)) as conn:
        conn.execute(
            "INSERT INTO todos (id, title, done) VALUES (?, ?, ?)", (todo_id, title, 0)
        )
        conn.commit()
    return todo_id


def mark_done(todo_id: str):
    with closing(sqlite3.connect(DB_PATH)) as conn:
        conn.execute("UPDATE todos SET done = 1 WHERE id = ?", (todo_id,))
        conn.commit()


init_db()
app = FastAPI(title="Todo API")


class TodoIn(BaseModel):
    title: str


@app.get("/health")
def healthcheck():
    return {"status": "ok"}


@app.get("/todos")
def list_todos():
    return all_todos()


@app.post("/todos")
def create(todo: TodoIn):
    todo_id = add_todo(todo.title)
    return {"id": todo_id, "title": todo.title, "done": False}


@app.post("/todos/{todo_id}/done")
def complete(todo_id: str):
```

```
    mark_done(todo_id)
    return {"id": todo_id, "done": True}
PY

cat > requirements.txt <<'TXT'
fastapi==0.112.0
uvicorn[standard]==0.30.1
TXT
```

2. Teste localement (optionnel) :

```
python3 -m venv .venv && source .venv/bin/activate
pip install -r requirements.txt
uvicorn app:app --reload
```

➜ Vérifie `http://127.0.0.1:8000/health`. Arrête avec `Ctrl+C` et désactive le venv.

Étape 1 — Dockerfile multi-stage

1. Crée `Dockerfile` :

```
# syntax=docker/dockerfile:1
FROM python:3.12-slim AS base
ENV PYTHONDONTWRITEBYTECODE=1 PYTHONUNBUFFERED=1
WORKDIR /app

FROM base AS deps
COPY requirements.txt .
RUN pip install --upgrade pip \
 && pip install --no-cache-dir -r requirements.txt

FROM base AS runtime
COPY --from=deps /usr/local/lib/python3.12/site-packages /usr/local/lib/python3.12/site-packages
COPY --from=deps /usr/local/bin/uvicorn /usr/local/bin/uvicorn
COPY app.py .
RUN adduser --disabled-password --gecos "" todo && chown -R todo:todo /app
USER todo
EXPOSE 8000
ENV PORT=8000 TODO_DB_PATH=/app/data/todo.db
HEALTHCHECK --interval=30s --timeout=5s --start-period=10s CMD curl -fsS http://127.0.0.1:${PORT}/health || exit 1
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8000"]
```

2. Construit :

```
docker build -t todo-api:2.0.0 .
```

3. Observe la taille de l'image (`docker image ls todo-api`). Compare avec une version single-stage (à construire en sous-tâche) et note la différence.

Étape 2 — EntryPoint et logs structurés

1. Crée `entrypoint.sh` :

```
cat > entrypoint.sh <<'SH'
#!/usr/bin/env sh
set -eu
echo "[entrypoint] $(date -u +"%Y-%m-%dT%H:%M:%SZ") - Démarrage de todo-api"
exec "$@"
SH
chmod +x entrypoint.sh
```

2. Mets à jour le `Dockerfile` (stage runtime) :

```
COPY entrypoint.sh .
COPY log.ini .
ENTRYPOINT ["./entrypoint.sh"]
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8000", "--log-config", "log.ini"]
```

3. Ajoute un fichier `log.ini` pour forcer un format JSON minimal :

```
cat > log.ini <<'INI'
[loggers]
keys=root,uvicorn

[handlers]
keys=console

[formatters]
keys=json

[formatter_json]
format={"timestamp":"%(asctime)s","level":"%(levelname)s","logger":"%(name)s","message":"%(message)s"}

[handler_console]
class=StreamHandler
level=INFO
formatter=json
args=(sys.stdout,)

[logger_root]
level=INFO
handlers=console

[logger_uvicorn]
level=INFO
handlers=console
propagate=0
qualname=uvicorn
INI
```

4. Reconstruis et vérifie :

```
docker build -t todo-api:2.1.0 .
docker run --rm -p 8000:8000 todo-api:2.1.0
```

➜ Observe les logs JSON. Répertorie la différence entre `ENTRYPOINT` et `CMD`.

Étape 3 — Réseaux et persistance sans compose

1. Crée un réseau dédié :

```
docker network create todo-net
```

2. Lance la base PostgreSQL :

```
docker run -d \
  --name todo-db \
  --network todo-net \
  -e POSTGRES_DB=todo \
  -e POSTGRES_USER=todouser \
  -e POSTGRES_PASSWORD=supersecret \
  -v todo-db-data:/var/lib/postgresql/data \
  postgres:16-alpine
```

➜ Suis les logs (`docker logs -f todo-db`) jusqu'au message prêt.

3. Adapte l'API pour utiliser PostgreSQL quand `TODO_DB_URL` est défini :
   - Ajoute `psycopg[binary]==3.2.1` dans `requirements.txt` .
   - Modifie `app.py` :

```
import psycopg

DB_URL = os.getenv("TODO_DB_URL")
USE_POSTGRES = bool(DB_URL)
```

```
- Si `USE_POSTGRES`, utilise `psycopg.connect(DB_URL, autocommit=True)` et des tables SQL équivalentes.
- Sinon, conserve SQLite.
```

- Ajoute une fonction `init_pg()` qui crée la table si nécessaire.

➜ Documente clairement les changements dans le TP (pseudo-code accepté si manque de temps).

4. Reconstruis l'image (`todo-api:2.2.0`) et lance-la reliée au réseau :

```
docker run -d \
 --name todo-api \
 --network todo-net \
 -e TODO_DB_URL=postgresql://todouser:supersecret@todo-db:5432/todo \
 -p 8000:8000 \
 todo-api:2.2.0
```

5. Vérifie :

```
curl -s http://127.0.0.1:8000/health | jq
curl -s -X POST http://127.0.0.1:8000/todos -H "Content-Type: application/json" -d '{"title":"Task 1"}' | jq
curl -s http://127.0.0.1:8000/todos | jq
```

6. Liste les volumes et démontre que les données persistent après redémarrage :

```
docker stop todo-api todo-db
docker start todo-db
docker start todo-api
curl -s http://127.0.0.1:8000/todos | jq
```

➜ Explique le rôle des réseaux et volumes créés manuellement.

Étape 4 — Tags, push et rollback

1. Tague l'image actuelle :

```
docker tag todo-api:2.2.0 todo-api:latest
docker tag todo-api:2.2.0 localhost:5001/todo-api:2.2.0
```

2. Lance un registre local :

```
docker run -d -p 5001:5000 --name registry registry:2
```

3. Pousse et vérifie :

```
docker push localhost:5001/todo-api:2.2.0
curl -s http://localhost:5001/v2/_catalog | jq
```

4. Simule un rollback :

```
docker tag todo-api:2.1.0 todo-api:rollback
docker run --rm todo-api:rollback --help
```

➜ Rédige quelques lignes sur la stratégie de tagging (ex. `major.minor.patch` , `latest` , `rollback`).

Étape 5 — Diagnostic et nettoyage

1. Inspecte les conteneurs :

```
docker inspect todo-api --format '{{json .Config.Env}}' | jq
docker stats todo-api todo-db
docker events --since 5m
```

2. Sauvegarde un log :

```
mkdir -p logs
docker logs todo-api > logs/todo-api.log
```

3. Nettoie :

```
docker stop todo-api todo-db registry
docker rm todo-api todo-db registry
docker network rm todo-net
docker volume ls
docker volume rm todo-db-data
docker image prune
```

➜ Attention : ne supprime pas une ressource si elle est utilisée ailleurs, note-le dans le rapport.

Livrables attendus

- Repository avec `app.py` , `requirements.txt` , `entrypoint.sh` , `log.ini` , `Dockerfile`
- Script ou documentation décrivant les commandes de démarrage (`docker network` , `docker run` …)
- Rapport synthétique :

    - architecture finale (API + DB + registre local)
    - stratégie de tagging et rollback
    - diagnostics réalisés (logs, stats, inspect)
- Exports des requêtes de test (captures `curl` ou équivalent)

Aller plus loin (optionnel)

- Remplacer SQLite par PostgreSQL uniquement et ajouter des migrations (`alembic`).
- Construire une image « builder » spécifique aux tests (ex. `docker run todo-api:deps pytest`).
- Mettre en place un job GitHub Actions imaginé qui exécute build + tests + push.
- Ajouter un service de cache Redis et prouver l'interaction via `docker run --network` .
- Expérimenter avec `docker buildx` pour produire une image multi-architecture (amd64/arm64).