TD – Conception d’une application conteneurisée générique

YOUSRA ZAABAT

21/12/2025

Répo Github: https://github.com/DO12CE/app-generique

SOMMAIRE

1/ Architecture : description de chaque service et des interactions.

L'application repose sur une architecture logicielle de type 3-tier, où chaque service est isolé dans son propre conteneur Docker afin de garantir la modularité et la sécurité.Les conteneurs sont: Flask Python pour l'API, Nginx pour le Frontend et PostgreSql pour la base de données

A. Service Database (db)

Pour respecter la consigne de construction responsable, j'ai sélectionné l'image **`postgres:16-alpine`** car la version Alpine Linux réduit considérablement la taille de l'image (environ 100 Mo contre plus de 400 Mo pour l'image standard) et diminue la surface d'attaque en limitant les outils pré-installés afin d'assurer le stockage et la persistance des données (tâches, messages, etc.).

```
image: postgres:16-alpine
```

Initialisation :

Le schéma de la base est créé automatiquement au premier démarrage grâce à un script SQL monté dans le répertoire /db-init/init.sql.

```
-- db-data:/var/lib/postgresql/data assure la persistance (2)
CREATE TABLE IF NOT EXISTS items (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    description TEXT
);


INSERT INTO items (name, description) VALUES
    ('Tâche 1: Conteneuriser API', 'Créer le Dockerfile multiétapes pour l API'),
    ('Tâche 2: Conteneuriser Front', 'Créer le Dockerfile multiétapes pour le Front')
ON CONFLICT (id) DO NOTHING;
```

Persistance :

Un volume Docker nommé est utilisé pour conserver les données même après la suppression ou le redémarrage du conteneur.

```
volumes:
  - db-data:/var/lib/postgresql/data
  - ./db-init:/docker-entrypoint-initdb.d
```

Le service de base de données est le socle de la persistance de notre application. Sa configuration repose sur l'utilisation d'une image officielle optimisée et de mécanismes d'automatisation au démarrage.

Initialisation Automatique du Schéma:

Pour que l'application soit prête à l'emploi dès le premier lancement, j'ai automatisé la création des tables en montant mon fichier `init.sql` dans le dossier spécifique `./db-init:/docker-entrypoint-initdb.d`, PostgreSQL exécute automatiquement les scripts SQL lors de la création du conteneur.

```sql
CREATE TABLE IF NOT EXISTS items (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    description TEXT
);
```

Persistance des Données via Volume Nommé:

Avec ce service, je m'assure que les données ne soient pas perdues si le conteneur est supprimé ou mis à jour. Contrairement à un simple montage local, le volume nommé est géré par le moteur Docker, offrant une meilleure isolation et garantissant que les données survivent à un `docker compose down`. Quant à l'orchestration, elle est rendue possible par l'inclusion d'un **healthcheck** dans la base de données.

Utilisation de `pg_isready` : Cette commande vérifie que le serveur PostgreSQL accepte bien les connexions avant que l'API ne tente de s'y connecter.

```yaml
healthcheck:
  test: ["CMD", "curl", "--fail", "http://localhost:${API_PORT}/status"]
  interval: 10s
  timeout: 3s
  retries: 5
```

B. Service Backend (api)

J'utilise Python avec un Dockerfile optimisé en multi-étapes pour agir comme couche de logique métier. Elle expose deux points d'entrée:

/status : Renvoie un message « OK » pour confirmer la disponibilité de l'API. La route est utilisée par le système d'orchestration pour vérifier la disponibilité du service.

/items : Récupère et renvoie la liste des objets stockés dans la base de données où l'on interroge cette dernière pour retourner les objets stockés au format JSON.

L'API interroge la base de données en utilisant des variables d'environnement (hôte, port, identifiants) pour établir la connexion.

Endpoints Fonctionnels (Route /status et /items) :

Conformément aux exigences, l'API expose deux routes principales.

Route /status :Route /items :

```python
@app.route('/status')
def status():
    try:
        conn = get_db_connection()
        conn.close()
        return jsonify({"status": "OK", "database_status": "Connected"}), 200
    except Exception as e:
        return jsonify({"status": "Error", "database_status": str(e)}), 500

@app.route('/items')
def get_items():
    items = []
    try:
        conn = get_db_connection()
        cur = conn.cursor()
        cur.execute("SELECT id, name, description FROM items;")
        items = [{'id': row[0], 'name': row[1], 'description': row[2]} for row in cur.fetchall()]
        cur.close()
        conn.close()
    except Exception as e:
        return jsonify({"error": "Failed to retrieve items", "details": str(e)}), 500
    return jsonify(items), 200
```

Externalisation de la Configuration :

Pour assurer la portabilité et la sécurité, aucune information de connexion n'est inscrite "en dur" dans le code Python.

Utilisation de os.environ : Le code Python récupère les paramètres (DB_HOST, DB_NAME, ...) directement depuis l'environnement du conteneur. **Le fichier .env** centralise les secrets et les ports, facilitant la maintenance sans modifier l'image Docker.

```python
DB_HOST = os.getenv('DB_HOST', 'localhost')
```

Construction Multi-étapes et Sécurité :

Le Dockerfile de l'API a été optimisé pour répondre aux critères de performance et de sécurité. Seuls les packages nécessaires à l'exécution sont conservés dans l'image finale. Par mesure de sécurité, le processus ne s'exécute pas avec les privilèges administrateur (root), limitant ainsi les risques en cas d'intrusion.

```dockerfile
# Builder
#
FROM python:3.11-slim AS builder

WORKDIR /app
COPY requirements.txt .
# cache layer
RUN pip install --no-cache-dir -r requirements.txt
COPY app.py .

# Runtime
FROM python:3.11-slim

RUN apt-get update && apt-get install -y curl

RUN adduser --system --uid 1000 appuser

WORKDIR /app

COPY --from=builder /usr/local/lib/python3.11/site-packages/ /usr/local/lib/python3.11/site-packages/
COPY --from=builder /app /app

EXPOSE 8080

USER appuser

# lancement
CMD ["python", "app.py"]
```

Intégration des Healthchecks :

Pour que **Docker Compose** puisse gérer intelligemment les dépendances (depends_on), l'API inclut l'outil curl. Docker interroge périodiquement http://localhost:8080/status. Si l'API répond "OK", elle est marquée comme healthy.

```yaml
healthcheck:
  test: ["CMD", "curl", "--fail", "http://localhost:${API_PORT}/status"]
  interval: 10s
  timeout: 3s
  retries: 5
```

C. Service Frontend (frontend)

Le service Frontend est la vitrine de l'application. j'y sépare la couche de présentation et la couche de données par l'usage de Nginx en tant que serveur web et proxy.

Interface Web Dynamique :

L'interface est composée de fichiers statiques (HTML/JavaScript) servis par une image **Nginx Alpine**, choisie pour sa légèreté extrême (25 Mo environ). Le JavaScript interroge l'API via des appels asynchrones (fetch) pour récupérer les données de la base et les afficher dynamiquement à l'utilisateur sans recharger la page.

```python
def get_items():
    items = []
    try:
        conn = get_db_connection()
        cur = conn.cursor()
        cur.execute("SELECT id, name, description FROM items;")
        items = [{'id': row[0], 'name': row[1], 'description': row[2]} for row in cur.fetchall()]
        cur.close()
        conn.close()
    except Exception as e:
        return jsonify({"error": "Failed to retrieve items", "details": str(e)}), 500
    return jsonify(items), 200
```

Configuration du Reverse-Proxy :

Pour résoudre les problèmes de ports et de sécurité, Nginx a été configuré comme un **reverse-proxy**. Nginx intercepte les requêtes destinées à l'API et les redirige vers le conteneur api sur le port 8080 au sein du réseau Docker. Cela permet au navigateur de ne communiquer qu'avec un seul port (le port 80 du Front), renforçant ainsi la sécurité et simplifiant l'architecture réseau.

```
server {
    listen 80;
    server_name localhost;

    root /usr/share/nginx/html;
    index index.html;

    location /status {
        proxy_pass http://api:8080/status;
    }
    location /items {
        proxy_pass http://api:8080/items;
    }

    location / {
        try_files $uri $uri/ =404;
    }
}
```

Construction Multi-étapes et Sécurité :

Le Dockerfile du frontend suit les mêmes standards de rigueur que l'API. Une étape de "build" prépare les fichiers, et une étape de "runtime" (Nginx) les sert, garantissant que les outils de développement ne sont pas présents dans l'image

finale. Le processus Nginx a été configuré pour s'exécuter avec un utilisateur appuser, limitant les privilèges au sein du conteneur.

```
# Dockerfile multi étapes pour le front et configuration Nginx avec reverse proxy (3b)
FROM alpine AS builder

WORKDIR /app
COPY index.html .
COPY nginx.conf /etc/nginx/nginx.conf

FROM nginx:alpine

# utilisateur non root (5a)
RUN adduser -D -u 1000 appuser

# On donne les droits à l'utilisateur sur les dossiers Nginx
RUN touch /var/run/nginx.pid && \
chown -R appuser:appuser /usr/share/nginx/html /var/cache/nginx /var/run /var/log/nginx

WORKDIR /usr/share/nginx/html

COPY --from=builder /app/index.html .
COPY --from=builder /etc/nginx/nginx.conf /etc/nginx/conf.d/default.conf


# pour ne pas être en root (1c/5a)
USER appuser

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

Orchestration et Dépendances :

Dans le fichier docker-compose.yml, le frontend est le dernier maillon de la chaîne.

depends_on : Le frontend ne démarre que lorsque l'API est signalée comme "saine" (service_healthy), évitant ainsi que l'utilisateur n'arrive sur une page vide ou en erreur.

```
frontend:        DO12CE, 2 weeks ago • fi
  build:
    context: ./frontend
  ports:
    - "${HOST_FRONT_PORT}:${FRONT_PORT}
  environment:
    API_BASE_URL: ${API_BASE_URL}
  depends_on:
    api:
      condition: service_healthy
```

D. Interactions et Orchestration

L'ensemble de ces services est orchestré par **Docker Compose**. La communication entre les conteneurs se fait via un réseau virtuel interne où chaque service est joignable par son nom. L'ordre de démarrage est contrôlé par des **healthchecks** : le frontend attend que l'API soit opérationnelle, et l'API attend que la base de données soit prête à recevoir des connexions.

L'ensemble des services est piloté par Docker Compose, qui agit comme le chef d'orchestre de l'infrastructure. Cette approche garantit que la stack est reproductible en une seule commande.

Réseau Virtuel et Résolution DNS Interne :

Tous les services sont isolés au sein d'un réseau bridge virtuel privé. Docker fournit un serveur DNS interne permettant aux conteneurs de communiquer entre eux via leurs noms respectifs au lieu d'adresses IP instables. Par exemple, l'API contacte la base de données en utilisant l'hôte db.

```
# BDD
POSTGRES_USER=app_user_db
POSTGRES_PASSWORD=password
POSTGRES_DB=generic_app_db
DB_HOST=db
DB_PORT=5432
```

```
api:
  build:
    context: ./api
  ports:
    - "${HOST_API_PORT}:${API_PORT}"
  environment:
    DB_HOST: ${DB_HOST}
    DB_USER: ${POSTGRES_USER}
    DB_PASSWORD: ${POSTGRES_PASSWORD}
    API_PORT: ${API_PORT}
```

Gestion de l'Ordre de Démarrage via Healthchecks :

Démarrer les conteneurs ne suffit pas ; ils doivent être opérationnels avant que leurs dépendants ne les sollicitent. L'utilisation de depends_on avec la condition service_healthy force un ordre logique. Le **Frontend** attend que l'**API** soit healthy, et l'**API** attend que la **Base de données** soit healthy.

```yaml
api:
  build:
    context: ./api
  ports:
    - "${HOST_API_PORT}:${API_PORT}"
  environment:
    DB_HOST: ${DB_HOST}
    DB_USER: ${POSTGRES_USER}
    DB_PASSWORD: ${POSTGRES_PASSWORD}
    API_PORT: ${API_PORT}
  depends_on:
    db:
      condition: service_healthy
```

```
$ docker compose ps
NAME                  IMAGE                COMMAND                  SERVICE   CREATED       STATUS
app-generique-api-1   app-generique-api    "python app.py"          api       5 hours ago   Up 5 hours (healthy)
generic-db            postgres:16-alpine   "docker-entrypoint.s…"   db        2 weeks ago   Up 6 hours (healthy)
```

Automatisation du Déploiement :

Pour valider le critère d'automatisation, un script Bash (build_and_deploy.sh) a été conçu pour exécuter l'intégralité du cycle de vie sans intervention manuelle.

Le script commence par un docker compose config pour valider l'intégrité du fichier avant toute action. Il intègre les scans de vulnérabilités et la signature des images avant le déploiement final.

```
#!/bin/bash
set -e # Arrêt du script si une commande échoue

echo "--- étape 1 - Vérification de la configuration ---"
docker compose config

echo "--- étape 2 - Construction des images optimisées ---"
docker compose build --no-cache

echo "--- étape 3 - Scan de sécurité (Docker Scout) ---"
docker scout quickview app-generique-api
docker scout quickview app-generique-frontend

echo "--- étape 4 - Signature et Publication ---"
# empêche Docker de pousser ou de tirer des images non signées (5b)
export DOCKER_CONTENT_TRUST=1


echo "--- 5. Déploiement opérationnel ---"
docker compose up -d

echo "Application déployée avec succès !"
```

2/ Commandes clés utilisées pour construire, configurer et déployer la stack.

L'utilisation de Docker Compose centralise la gestion des services, j'ai ajouté plusieurs commandes dans le script pour répondre aux exigences de sécurité, de configuration et d'automatisation.

A. Construction et Validation :

Avant tout déploiement, je m'assure que les images soient correctement construites et que le fichier d'orchestration est valide.

docker compose config :

Utilisée pour valider la syntaxe du fichier docker-compose.yml et vérifier que les variables d'environnement du fichier .env sont correctement injectées.

docker compose build --no-cache :

Force la reconstruction complète des images en utilisant les Dockerfiles multi-étapes, garantissant que les dernières optimisations et correctifs de sécurité sont inclus.

B. Sécurité et Signature :

Conformément aux contraintes de sécurité du projet, chaque image est analysée avant d'être déployée.

docker scout quickview :

Pour d'analyser les vulnérabilités des images construites (API et Frontend) et interpréter les résultats pour appliquer des correctifs au besoin.

export DOCKER_CONTENT_TRUST=1 :

Cette commande active la signature des images, garantissant que seules les images authentifiées peuvent être poussées vers le registre ou déployées.

C. Déploiement et Supervision :

Une fois les images validées, la stack est lancée et surveillée pour s'assurer de sa haute disponibilité.

docker compose up -d :

Déploie l'ensemble des services (Base de données, API, Frontend) en arrière-plan. Cette commande crée également les réseaux virtuels et les volumes de persistance.

docker compose ps : Commande indispensable pour vérifier le statut de santé (healthy) des conteneurs grâce aux *healthchecks* mis en place.

```
NAME                IMAGE                COMMAND                  SERVICE   CREATED       STATUS
app-generique-api-1 app-generique-api    "python app.py"          api       5 hours ago   Up 5 hours (healthy)
generic-db          postgres:16-alpine   "docker-entrypoint.s…"   db        2 weeks ago   Up 6 hours (healthy)
```

docker compose logs -f [service] :

Utilisée pour superviser en temps réel le comportement d'un service spécifique, facilitant le débogage et la vérification des interactions entre l'API et la base de données.

3/ Bonnes pratiques suivies (multi-étapes, images légères, .dockerignore…).

A. Multi-stage builds :

C'est l'une des pratiques les plus structurantes de ce projet. Pour l'API et le Frontend, j'ai séparé la phase de construction (compilation, installation des dépendances) de la phase d'exécution. Le premier bloc FROM (étage "builder") prépare les fichiers, tandis que le second bloc FROM ne récupère que les exécutables finaux via l'instruction COPY --from=builder. Cela permet de ne pas embarquer de compilateurs ou de caches inutiles dans l'image finale, garantissant une image de production minimaliste.

B. Choix d'images légères :

Pour chaque service, j'ai privilégié des images basées sur **Alpine Linux** ou des versions **Slim**. Cette pratique réduit drastiquement l'empreinte disque et la surface d'attaque du conteneur.

```
FROM alpine AS builder

FROM nginx:alpine

FROM python:3.11-slim AS builder
```

C. Optimisation via le fichier .dockerignore :

Un fichier .dockerignore a été ajouté à la racine des répertoires api/ et frontend/. Il empêche Docker d'inclure des fichiers inutiles dans le contexte de construction. Les builds sont plus rapides et l'image finale ne contient que le code source nécessaire.

D. Sécurité : Utilisateur non-root et capacités limitées :

La sécurité a été renforcée en évitant l'utilisation du compte root à l'intérieur des conteneurs. J'ai créé un utilisateur système dédié avec des privilèges restreints. L'instruction USER appuser en fin de Dockerfile garantit que si une faille est exploitée dans l'application et donc l'attaquant n'aura pas les droits d'administration sur le reste de l'infrastructure.

```
"
FROM python:3.11-slim AS builder

WORKDIR /app
COPY requirements.txt .
# cache layer
RUN pip install --no-cache-dir -r requirements.txt
COPY app.py .

# Runtime
FROM python:3.11-slim

RUN apt-get update && apt-get install -y curl

RUN adduser --system --uid 1000 appuser

WORKDIR /app

COPY --from=builder /usr/local/lib/python3.11/site-
COPY --from=builder /app /app

EXPOSE 8080

USER appuser
```

4/ Difficultés rencontrées et améliorations possibles

A. Difficultés rencontrées :

Gestion des Healthchecks et dépendances d'outils :

L'une des difficultés majeures a été la mise en place des tests de santé (healthchecks) pour l'API. L'image de base python-slim étant dépourvue de l'outil curl, les tests échouaient systématiquement, empêchant le service d'être marqué

comme "healthy". La solution a consisté à modifier le Dockerfile pour installer manuellement curl dans l'étape finale.

Privilèges et utilisateurs non-root :

L'exécution du service Frontend sous un utilisateur non-root a initialement provoqué des erreurs de démarrage (permission refusée sur le fichier .pid et les logs de Nginx). Il a fallu ajuster précisément les droits (chown) dans le Dockerfile pour permettre à l'utilisateur appuser de manipuler les fichiers nécessaires sans être administrateur.

Communication Frontend-API :

La configuration du reverse-proxy a nécessité une attention particulière sur la résolution DNS interne de Docker. L'utilisation de chemins relatifs dans le code JavaScript a permis de s'assurer que les requêtes transitent correctement par Nginx avant d'atteindre l'API via le réseau virtuel.

B. Améliorations possibles :

Mise en place d'un pipeline CI/CD complet :

Bien que le script build_and_deploy.sh automatise les étapes locales, une amélioration logique serait l'intégration de GitHub Actions ou GitLab CI. Cela permettrait d'exécuter des tests unitaires et des scans de sécurité automatiquement à chaque push.

Scaling et Haute Disponibilité :

Pour supporter une charge utilisateur importante, la stack pourrait être migrée vers un orchestrateur comme Docker Swarm ou Kubernetes. Cela permettrait de multiplier les instances du service API et d'assurer une disponibilité continue même en cas de panne d'un conteneur.

Sécurisation avancée :

Actuellement, les mots de passe sont stockés dans un fichier .env. Une amélioration cruciale pour la production serait d'utiliser des outils de gestion de secrets pour éviter que des informations sensibles ne soient présentes sur le disque ou dans l'environnement du conteneur.

Supervision et Métriques :

L'ajout d'une pile de monitoring (comme Grafana) permettrait de suivre en temps réel la consommation de ressources et le temps de réponse de l'API.

Conclusion

Réaliser cette architecture conteneurisée a été un challenge très formateur. J'ai réussi à livrer une application complète (Base de données, API, Front) qui respecte les standards actuels : sécurité, légèreté et automatisation.

Ce que je retiens de ce projet, c'est qu'un bon Dockerfile ne s'improvise pas : le choix des images de base et la gestion des utilisateurs sont cruciaux pour la qualité finale. Malgré quelques obstacles techniques résolus en cours de route, le script de déploiement fonctionne désormais parfaitement.

Ce TP valide mes compétences sur Docker et me donne envie d'aller plus loin, notamment vers la mise en place d'un pipeline d'intégration continue.