Conception d'une application conteneurisée générique

Contexte du projet

L'application développée dans ce projet est construite sous forme de services conteneurisés grâce à Docker. Elle contient une base de données, utilisée par une API, et une interface web qui récupère et affiche les informations récupérées depuis l'API.

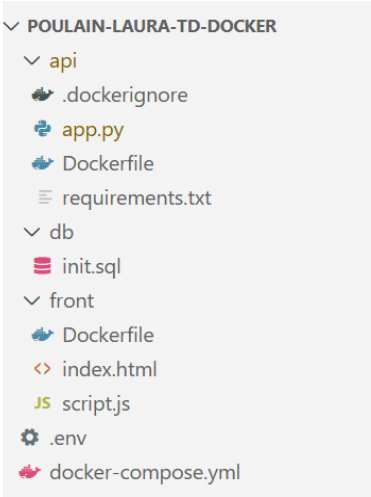
Le but est de mettre en place une architecture à la fois légère et complète, où les services peuvent être configurés de façon indépendante, tout en communiquant avec les autres, avec **Docker Compose**.

Architecture

L'application est composée de trois services principaux, gérés et liés grâce à Docker Compose:

- API (**Python Flask**) : ce service interroge la base de données et permet de l'exposer sur l'application web
- Base de données (**MySQL**) : stockage des objets, la base est initialisée au démarrage grâce au script init.sql.
- Front-end du site web (**Apache** et **HTML/JS**) : interface accessible à l'utilisateur, qui affiche la liste des objets contenus dans la base de données, et récupérés via l'API

Structure du projet



1. API

L'API expose 2 routes, /status et /items, et permet également l'accès à la base de données.

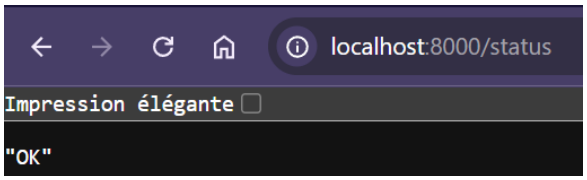
a) Fichier app.py

Le fichier principal **app.py** contient le code de l'API Flask.

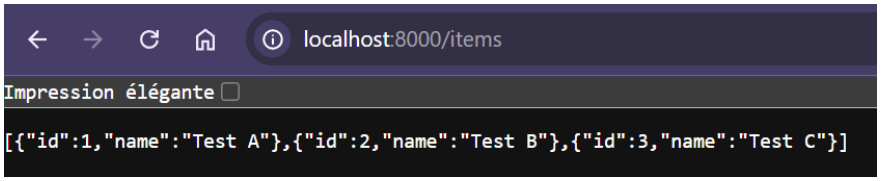
Ce fichier permet notamment de gérer la connexion de l'API à la base de données MySQL (avec le module connector.connect, et en utilisant les variables d'environnement définies dans le fichier **.env**).

Il permet également de gérer les routes :

- **/status**, qui, si l'API est disponible, renvoie le message **OK** (**ERREUR** sinon)



- **/items**, qui renvoie la liste des objets contenus dans la base MySQL, en se connectant à la base



L'API renvoie les informations au format JSON, ce qui permet au front de récupérer facilement les données grâce aux routes définies dans le fichier **app.py**.

b) Fichiers requirements.txt et .dockerignore

Le fichier **requirements.txt** contient la liste des dépendances à installer pour exécuter l'API, qui seront installées automatiquement avec Docker Compose.

Pour réduire la taille de l'image et optimiser le déploiement, le fichier **.dockerignore** est utile puisqu'il spécifie quels fichiers ne doivent pas être copiés dans l'image Docker (caches Python, dossier env par exemple).

c) Fichier Dockerfile

Le Dockerfile de l'API utilise un build **multi-étapes** : une première image installe les dépendances Python, puis une seconde, plus légère, sert d'image finale.

Les services fonctionnent par étapes : la base de données démarre en premier, puis l'API ne se lance qu'une fois MySQL déclaré comme disponible.

Enfin, l'interface web devient accessible seulement lorsque l'API est disponible à son tour.Cela garantit un ordre de démarrage correct et évite les erreurs liées à des services qui seraient encore indisponibles.

Le port de l'API est défini dans le fichier **.env** (port **8000**) et exposé au démarrage du conteneur pour rendre l'API accessible.

2. Base de données

La base de données MySQL permet quant à elle de stocker les objets. Celle-ci est initialisée automatiquement avec le script **init.sql**, qui crée la table stockant les objets et y insère des données de test.

Les données sont stockées dans deux volumes (un de données, un autre de montage de fichiers), définis dans le fichier **docker-compose.yml**, ce qui garantit leur persistance même en cas de conteneur supprimé ou stoppé.

3. Front-end

Le front-end est structuré avec deux fichiers, **index.html** (la page web statique) et **script.js** (qui permet d'appeler l'API, récupérant les données au format JSON, afin de les afficher dans la page web HTML).

La page s'affiche sur le port **8080**.

Un fichier **Dockerfile** lui est également associé. Le Dockerfile du front utilise l'image légère **httpd:2.4-alpine** et copie le contenu du dossier du front dans /usr/local/apache2/htdocs/, le répertoire utilisé par Apache. L'image obtenue héberge donc directement la page web.

Commandes clés utilisées

Construction des images

> On utilise la commande ***docker compose build,*** qui récupère les dépendances et génère les images à partir des Dockerfiles.

```
PS C:\Users\laura\OneDrive\Documents\POULAIN-Laura-TD-Docker> docker compose build
[+] Building 2.5s (22/22) FINISHED
 => [internal] load local bake definitions
 => => reading from stdin 1.14kB
 => [front internal] load build definition from Dockerfile
 => => transferring dockerfile: 315B
 => [api internal] load build definition from Dockerfile
 => => transferring dockerfile: 692B
 => [front internal] load metadata for docker.io/library/httpd:2.4-alpine
 => [api internal] load metadata for docker.io/library/python:3.11-slim
```

> Si les changements ne s'appliquent pas et semblent non détectés, on peut utiliser la commande ***docker compose build –no-cache***, pour reconstruire l'image sans utiliser le cache.

```
PS C:\Users\laura\OneDrive\Documents\POULAIN-Laura-TD-Docker> docker compose build --no-cache
[+] Building 10.5s (20/21)
 => [front internal] load .dockerignore
 => => transferring context: 2B
 => [front internal] load build context
 => => transferring context: 91B
 => CACHED [front 1/2] FROM docker.io/library/httpd:2.4-alpine@sha256:07b2fabb7029a0b8aeb2e0fd
 => => resolve docker.io/library/httpd:2.4-alpine@sha256:07b2fabb7029a0b8aeb2e0fd02651c28fe22c
```

Lancement des services

> On utilise la commande ***docker compose up -d***, permettant de lancer tous les services.

```
PS C:\Users\laura\OneDrive\Documents\POULAIN-Laura-TD-Docker> docker compose up -d
[+] Running 3/3
 ✔Container poulain-laura-td-docker-api-1    Started
 ✔Container poulain-laura-td-docker-front-1  Started
 ✔Container poulain-laura-td-docker-db-1     Healthy
```

Vérification du bon fonctionnement des conteneurs

> Listage des conteneurs avec la commande ***docker compose ps***, qui affiche la liste des conteneurs gérés par Docker Compose ainsi que leur état.

```
PS C:\Users\laura\OneDrive\Documents\POULAIN-Laura-TD-Docker> docker compose ps
NAME                            IMAGE                           COMMAND                  SERVICE
poulain-laura-td-docker-api-1   poulain-laura-td-docker-api     "python app.py"          api
000->8000/tcp, [::]:8000->8000/tcp
poulain-laura-td-docker-db-1    mysql:8.0                       "docker-entrypoint.s…"   db
 33060/tcp
poulain-laura-td-docker-front-1 poulain-laura-td-docker-front   "httpd-foreground"       front
```

```
SERVICE   CREATED          STATUS                   PORTS
api       49 seconds ago   Up 41 seconds            0.0.0.0:8000->8000/tcp, [::]:8000->8000/tcp
db        5 hours ago      Up 47 seconds (healthy)  3306/tcp, 33060/tcp
front     48 seconds ago   Up 40 seconds            0.0.0.0:8080->80/tcp, [::]:8080->80/tcp
```

Mesure de la taille des images

> *On peut voir la taille des images utilisées avec la commande* ***docker images,*** qui affiche toutes les images présentes localement, avec leur taille et leur date de création.

```
PS C:\Users\laura\OneDrive\Documents\POULAIN-Laura-TD-Docker> docker images
REPOSITORY                       TAG       IMAGE ID       CREATED          SIZE
poulain-laura-td-docker-api      latest    a79e5bc5bf63   9 minutes ago    328MB
poulain-laura-td-docker-front    latest    a2d2769698d7   9 minutes ago    93.2MB
```

Signatures des images (Docker Content Trust)

> On peut dans un premier temps, activer la signature pour toutes les images Docker qui seront push, avec la commande ***setx DOCKER_CONTENT_TRUST 1***

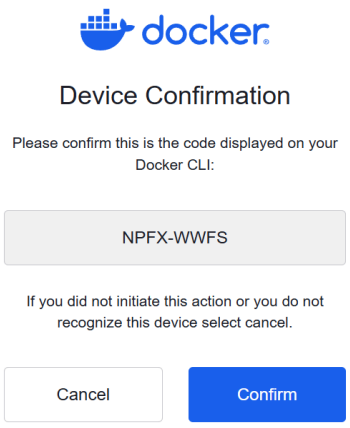
```
PS C:\Users\laura\OneDrive\Documents\POULAIN-Laura-TD-Docker> setx DOCKER_CONTENT_TRUST 1

RÉUSSITE : la valeur spécifiée a été enregistrée.
```

> Puis en faisant le push d'une image avec la commande ***docker push nom-image,*** Docker Content Trust créera une signature (clé privée locale), l'enregistrant dans le registre du Docker Hub, et vérifie à chaque pull que l'image est bien authentique.

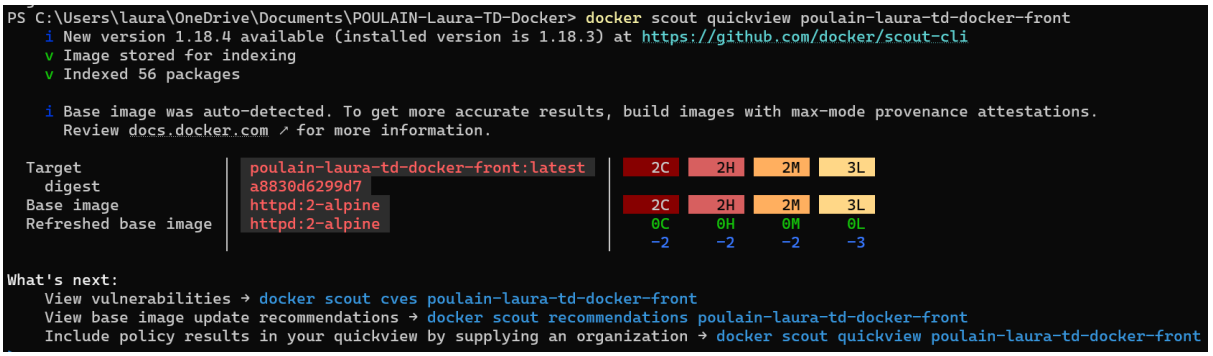
Scan des images

> Analyse l'image pour rechercher des vulnérabilités connues, avec la commande ***docker scout quickview nom-image***
Pour utiliser la commande, on s'authentifie d'abord avec la commande **docker login** :

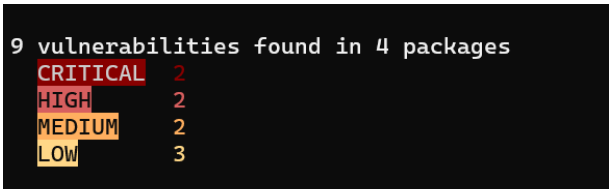
```
PS C:\Users\laura\OneDrive\Documents\POULAIN-Laura-TD-Docker> docker login

USING WEB-BASED LOGIN

i Info → To sign in with credentials on the command line, use 'docker login -u <username>'


Your one-time device confirmation code is: NPFX-WWFS
Press ENTER to open your browser or submit your device code here: https://login.docker.com/activate

Waiting for authentication in the browser…
```



On peut alors lancer la commande de scan sur l'image front : **docker scout quickview poulain-laura-td-docker-front.** Il détecte que l'image utilise httpd:2-alpine et montre qu'elle contient plusieurs vulnérabilités, critiques à modérées.

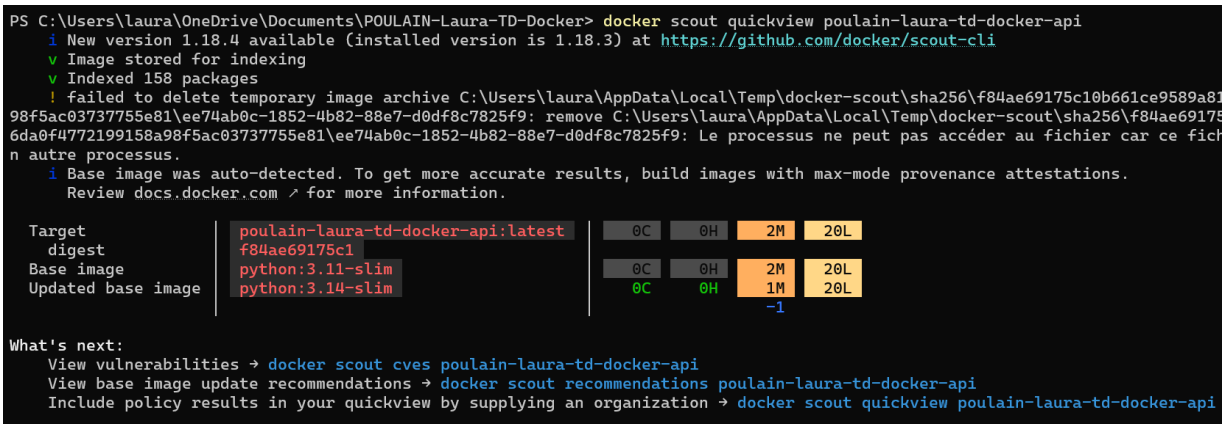


Les vulnérabilités peuvent être détaillées avec la commande **docker scout cves poulain-laura-td-docker-front**. Au vu des résultats, 2 sont critiques et 2 hautes, il pourrait donc être recommandé de mettre l'image à jour si possible.



En testant la commande sur l'image de l'API, Docker détecte plusieurs vulnérabilités provenant de l'image **python:3.11-slim**. Il indique également qu'une version plus récente (**python:3.14-slim**) est disponible.

Ici, il détecte des vulnérabilités moins conséquentes, allant du niveau moyen à un niveau bas de criticité.



Bonnes pratiques mises en place

- La construction **multi-étapes** dans les fichiers DockerFile permet une image finale plus légère, optimisée et sécurisée, intégrant uniquement les dépendances nécessaires.

- Les images choisies sont minimalistes, afin de réduire la taille totale de l'image et l'optimiser : **python:3.11-slim** pour l'API, **httpd:alpine.**

- Fichier **.dockerignore** : comme mentionné précédemment, il permet d'empêcher de récupérer des fichiers inutiles à Docker (caches python, environnements)

- Des variables d'environnement sont utilisées (fichier **.env**) pour configurer la base de données, et les informations de connexion à la base plus précisément (ports, identifiants, nom de la base). Cela permet d'éviter le stockage direct dans les fichiers source.

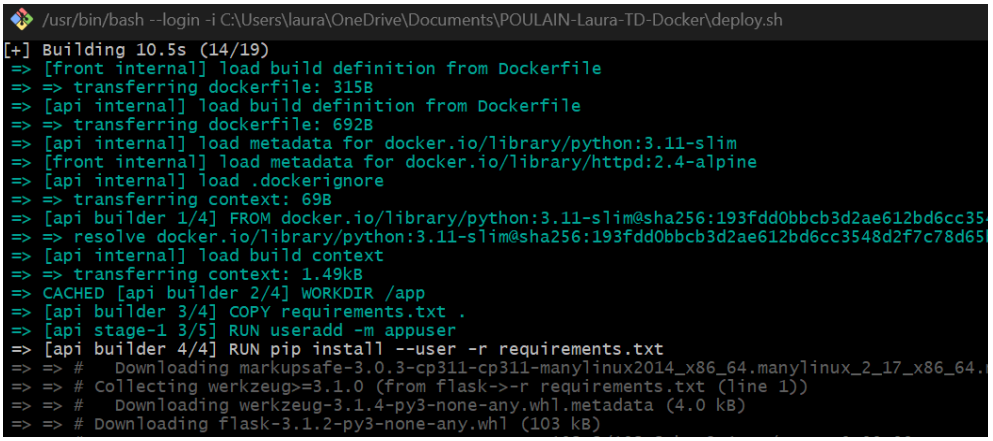
- Un utilisateur **non-root** a également été configuré dans l'API afin de renforcer la sécurité, évitant un accès administrateur en cas de compromission.

- Le fichier docker-compose.yml intègre également un **healthcheck**, qui permet de vérifier automatiquement que la base MySQL est active avant de pouvoir lancer l'API

- Comme mentionné précédemment, des volumes dédiés ont été spécifiés pour la base MySQL pour garantir la persistance des données (dans le fichier **docker-compose.yml**)

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

- Le fichier **deploy.sh** permet, à son exécution, d'automatiser tout le processus de déploiement : construction des images, connexion au registre Docker, push des images signées, puis lancement via Docker Compose.



Difficultés rencontrées

Cache Docker persistant

Même après modification et corrections sur des fichiers ou erreurs, Docker utilisait parfois d'anciennes versions des fichiers stockées en cache. Pour corriger le problème, la solution a été d'utiliser la commande "**docker compose build --no-cache**", ou si le problème persistait de supprimer l'image avec la commande "**docker rmi nom-image**"

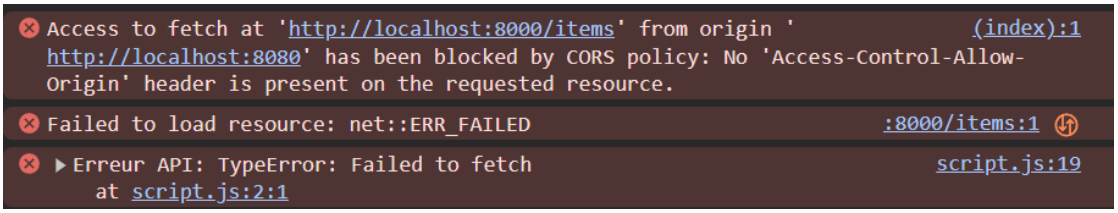
Problème de permission Apache

Après avoir indiqué l'utilisation d'un utilisateur non root pour ce service, cette utilisation provoquait des erreurs au démarrage du conteneur qui se stoppait immédiatement après son lancement.
Apache nécessite un utilisateur root pour créer certains fichiers, c'est pourquoi il a été conservé pour ce service (à la différence de l'API).

Problème lors de l'affichage sur l'interface web des objets

Le front n'arrivait pas à récupérer les données depuis l'API, car le navigateur bloquait les requêtes entre deux origines différentes, via le mécanisme **CORS** (voir ci-dessous). Le CORS est un mécanisme de sécurité des navigateurs qui bloque les appels vers une autre adresse ou un autre port si le serveur n'autorise pas explicitement ces requêtes.

Les objets ne s'affichaient donc pas sur la page web.



Ce problème a été résolu en modifiant la configuration de l'API pour autoriser les requêtes provenant du front-end (ajout de CORS dans **app.py** et dans **requirements.txt**).

Améliorations possibles

Pour améliorer le projet, il serait pertinent de pouvoir intégrer de la **supervision**, avec Prometheus et Grafana pour collecter et mesurer la performance (temps de réponse, utilisation de la mémoire / CPU).

Des **logs centralisés** pourraient également être mis en place pour faciliter l'analyse en cas d'incident et la détection d'anomalies.