TP1 : Docker

**Database :**

On crée le fichier « Dockerfile » suivant sans mettre le mot de passe dans le fichier:

Une image contenant texte, capture d’écran, Police, logiciel

Le contenu généré par l’IA peut être incorrect.

On crée également les fichier d’initiation :

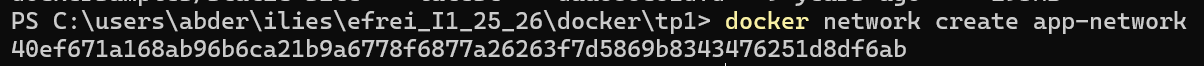
Une image contenant texte, logiciel, Logiciel multimédia, Logiciel de graphisme

Le contenu généré par l’IA peut être incorrect.

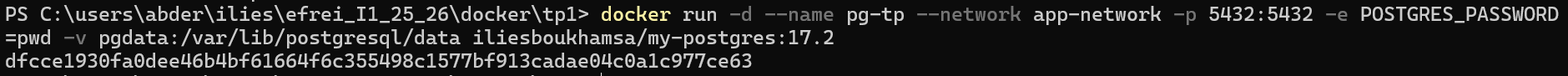
Puis on crée l’image :



Ensuite on crée le réseau :



Après ça on run le conteneur et au moment du run on passe en argument le mots de passe grâce au flag ‘’-e’’ on utilise également le flag ‘’-v’’ pour créer un volume pour conserver les données sur le disque hôte:



Ensuite on lace Adminer



Puis je me connecte à la base de donnée sur le naviguateur :

Une image contenant texte, capture d’écran, logiciel, nombre

Le contenu généré par l’IA peut être incorrect.

On peut voire que la base de donné à bien été initialisé :

Une image contenant texte, capture d’écran, logiciel, nombre

Le contenu généré par l’IA peut être incorrect.

**Q1-1 :** Il est préférable d’utiliser l’option -e pour passer les variables d’environnement plutôt que de les écrire dans le Dockerfile, car cela évite de stocker des informations sensibles (comme les mots de passe) dans l’image et dans le dépôt Git. Cette méthode est plus sécurisée, plus flexible (on peut changer les valeurs sans reconstruire l’image) et permet d’utiliser la même image sur plusieurs environnements avec des paramètres différents.

**Q1-2 :** On attache un volume à PostgreSQL pour garantir la persistance des données, car sans volume les données seraient perdues à la suppression du conteneur. Le volume monte le dossier /var/lib/postgresql/data sur la machine hôte, ce qui permet de conserver les fichiers de la base, de faciliter les sauvegardes et les restaurations, et d’assurer de meilleures performances.

**Backend API :**

**Basics :**

1.on compile le code Main.java :



Ou si problème de version de java



Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

2. on crée le dockerfile suivant :

Une image contenant texte, capture d’écran, Police, logiciel

Le contenu généré par l’IA peut être incorrect.

3. On lance l’app :

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Le contenu généré par l’IA peut être incorrect.

**Multisage Build, Backend simple API :**

On génère l’application springboot :

Une image contenant texte, capture d’écran, nombre, Police

Le contenu généré par l’IA peut être incorrect.

On ajoute le fichier greeting controller au springBoot:

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Le contenu généré par l’IA peut être incorrect.

On ajoute le Dockerfile suivant :

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Le contenu généré par l’IA peut être incorrect.

On build :



Puis on lance :

Une image contenant texte, capture d’écran, Police, menu

Le contenu généré par l’IA peut être incorrect.

Une image contenant texte, capture d’écran, Police, nombre

Le contenu généré par l’IA peut être incorrect.

Q1-5 : A multistage build is used to separate the build and runtime environments, making the final image smaller, faster, and more secure. The first stage uses a full JDK (with Maven) to compile the Java project, while the second stage only keeps the compiled JAR and runs it with a lightweight JRE.

Step-by-step explanation of the Dockerfile:

1- FROM eclipse-temurin:21-jdk-alpine AS myapp-build → creates the build stage using a JDK to compile the app.

2- WORKDIR /opt/myapp and RUN apk add maven → sets the working directory and installs Maven to build the project.

3- COPY pom.xml . and COPY src ./src → copies the project files into the container.

4- RUN mvn package -DskipTests → compiles the project and generates a JAR.

5- FROM eclipse-temurin:21-jre-alpine → starts a new lightweight stage for runtime only.

6- COPY --from=myapp-build /opt/myapp/target/\*.jar /opt/myapp/myapp.jar → copies the compiled JAR from the build stage.

7- ENTRYPOINT ["java", "-jar", "myapp.jar"] → runs the Spring Boot application with the JRE.

**Backend API :**

Dans le code source api, on place le docker file et on modifier le fichier application.yml :

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Le contenu généré par l’IA peut être incorrect.

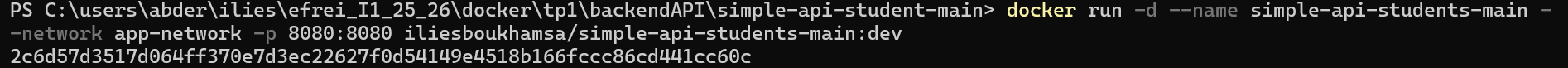
Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Le contenu généré par l’IA peut être incorrect.

On build :



Puis on run :



On obtient :

Une image contenant texte, capture d’écran, Police, ligne

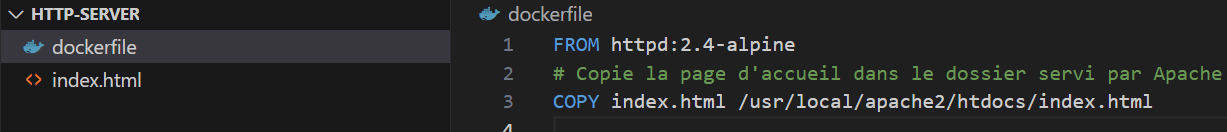
Le contenu généré par l’IA peut être incorrect.

**Http server**

**Basics**

Une image contenant texte, logiciel, Logiciel multimédia, capture d’écran

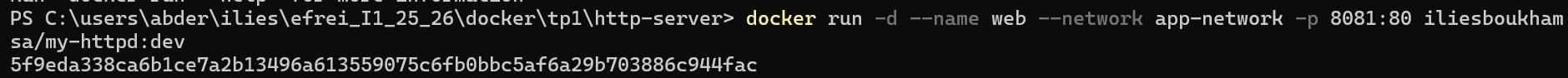
Le contenu généré par l’IA peut être incorrect.



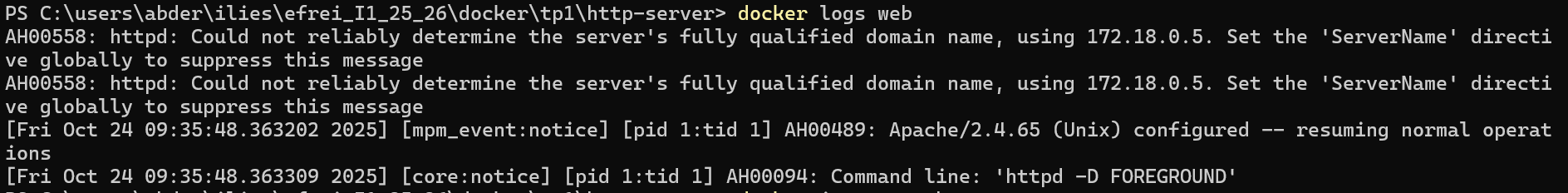
On build :



On run :



On teste les commande :



**Configuration**

Puis on récupère la configuration par defaut de apache :



On peut aussi taper :

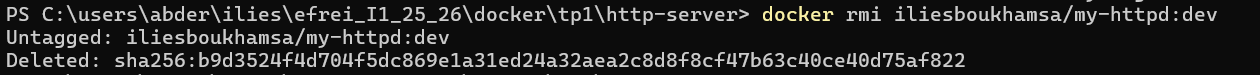


**Reverse proxy**

On supprime le conteneur et l’image :





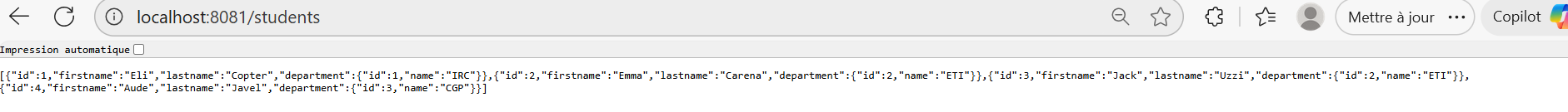


Maintenant on rebuild et on relance :

Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.

Voici le résultat :



Q1.5) Un reverse proxy agit comme un intermédiaire entre les clients et les serveurs backend.

Il permet de rediriger et gérer les requêtes entrantes afin que les utilisateurs accèdent à un point d’entrée unique (par exemple Apache sur le port 80), tandis que le reverse proxy transmet ces requêtes à l’application backend (comme ton API Spring Boot sur le port 8080).

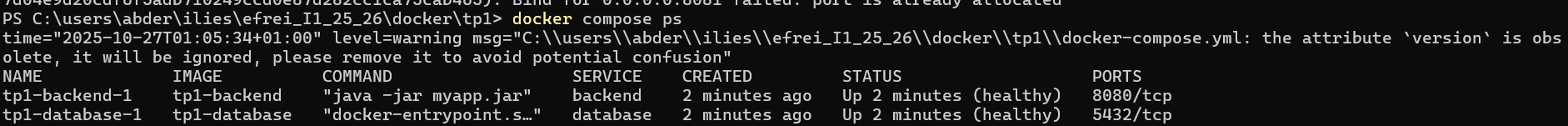
Il améliore la sécurité (en masquant les services internes), la flexibilité (en gérant le routage, le cache, le SSL et la répartition de charge) et la maintenabilité (en permettant à plusieurs services de fonctionner derrière un seul serveur).

Dans ce TP, il te permet d’accéder à ton API backend via le serveur Apache sans exposer directement le backend.

**Link application**

Docker-compose





Résultat :

Une image contenant texte, capture d’écran, Police, ligne

Le contenu généré par l’IA peut être incorrect.

Q1.6) Orchestre plusieurs conteneurs facilement. Déclare l’infra comme code. Démarre/arrête/reconstruit tout d’un coup. Gère réseaux, volumes, variables, dépendances. Rend les environnements reproductibles.

Q1.7)

docker compose up -d : build + démarre en arrière-plan.

docker compose up --build -d : rebuild puis démarre.

docker compose down : arrête et supprime conteneurs/réseaux.

docker compose down -v : idem + supprime volumes.

docker compose ps : liste l’état des services.

docker compose logs -f <svc> : suit les logs.

docker compose restart <svc> : redémarre un service.

docker compose exec <svc> <cmd> : exécute une commande dans un conteneur.

docker compose build : (re)construit les images.

docker compose pull / push : tire/publie les images.

Q1.8)

services: définit database, backend, httpd.

Chaque service a build.context (dossier Dockerfile).

environment passe les variables (URL DB, user, mdp).

depends\_on fixe l’ordre et la santé requise.

expose rend un port dispo au réseau interne.

ports publie le port public (ici httpd: 8081:80).

volumes persiste les données (pgdata pour Postgres).

networks met tous les services sur app-network pour qu’ils se résolvent par nom.

healthcheck vérifie la disponibilité (DB: pg\_isready, API: /actuator/health).

restart: unless-stopped relance automatiquement en cas d’échec.