Question 1

Pourquoi devrions-nous exécuter le conteneur avec un drapeau pour donner les variables d'environnement ? « -e »

Utiliser le drapeau -e lors de l'exécution d'un conteneur permet de définir des variables d'environnement, offrant une configuration flexible et dynamique de l'application sans modifier son code. Cela facilite la gestion des paramètres comme les connexions aux bases de données ou les clés d'API, tout en séparant la configuration du code source pour plus de sécurité et de modularité. Cela permet aussi d'adapter rapidement les configurations selon les environnements (développement, production), et s'intègre bien dans des workflows de déploiement ou d'intégration continue.

Question2

Pourquoi avons-nous besoin d'un volume à attacher à notre conteneur postgres ?

Nous avons besoin d'attacher un volume à un conteneur PostgreSQL pour persister les données au-delà de la durée de vie du conteneur. Par défaut, lorsque vous exécutez un conteneur, toutes les données créées à l'intérieur sont éphémères et sont perdues lorsque le conteneur est supprimé. En attachant un volume, les données de la base de données (comme les tables, les enregistrements, etc.) sont stockées sur l'hôte ou sur un stockage externe, ce qui garantit que même après la suppression ou le redémarrage du conteneur, les données restent intactes et accessibles. Cela permet également une séparation des préoccupations entre la gestion des données et l'exécution du conteneur, facilitant les mises à jour ou les migrations du conteneur sans risque de perte de données.

Question3

1-1 Documentez l'essentiel du conteneur de votre base de données : commandes et Dockerfile.

Explication des directives :

- FROM postgres:14.1-alpine : Utilise l'image officielle PostgreSQL version 14.1, optimisée pour un usage léger avec Alpine Linux.
- ENV POSTGRES_DB=db : Définit la base de données db comme base de données par défaut lors de la création du conteneur.
- ENV POSTGRES_USER=usr : Définit l'utilisateur usr qui aura les droits d'accès à la base de données.
- ENV POSTGRES_PASSWORD=pwd : Définit le mot de passe pwd pour l'utilisateur usr.

- COPY ./scripts /docker-entrypoint-initdb.d/ : Copie les scripts SQL (présents dans le répertoire scripts/) dans le dossier /docker-entrypoint-initdb.d/ du conteneur. Ces scripts seront exécutés lors de l'initialisation du conteneur.

Question 4

Pourquoi avons-nous besoin d'une construction en plusieurs étapes ? Et expliquez chaque étape de ce dockerfile. en 1 seul paragrafe

La construction en plusieurs étapes dans Docker est utilisée pour optimiser la taille de l'image finale et séparer les environnements de build et d'exécution. Dans le premier stage de ce Dockerfile, nous utilisons une image avec le **JDK** (Java Development Kit) pour compiler le code Java en un fichier JAR, ce qui nécessite des outils comme Maven, et peut inclure de nombreux fichiers inutiles pour l'exécution finale. Ensuite, dans le second stage, nous utilisons une image plus légère contenant uniquement le **JRE** (Java Runtime Environment), car c'est tout ce qui est nécessaire pour exécuter l'application. Cela permet de ne pas inclure les outils de build dans l'image finale, réduisant ainsi sa taille et améliorant l'efficacité. La commande COPY --from=0 copie uniquement le fichier JAR compilé du premier stage vers le second, et enfin, l'application est exécutée avec java -jar myapp.jar. Cette approche est plus propre et sécurisée, car elle isole les outils de développement de l'environnement de production.

Question 5

Pourquoi avons-nous besoin d'un proxy inverse ?

Un proxy inverse est essentiel car il agit comme un intermédiaire entre les clients et les serveurs d'application, offrant des avantages tels que la répartition de la charge pour gérer le trafic de manière efficace, l'amélioration de la sécurité en cachant les adresses IP. des serveurs internes et en filtrant les requêtes malveillantes, ainsi que la gestion de la terminaison SSL/TLS pour simplifier le chiffre. De plus, il peut mettre en cache les réponses pour réduire la latence, compresser les données pour optimiser les performances et faciliter l'accès à des services internes, ce qui en fait un outil précieux pour améliorer la performance, la sécurité et la scalabilité des candidatures.

Question 6

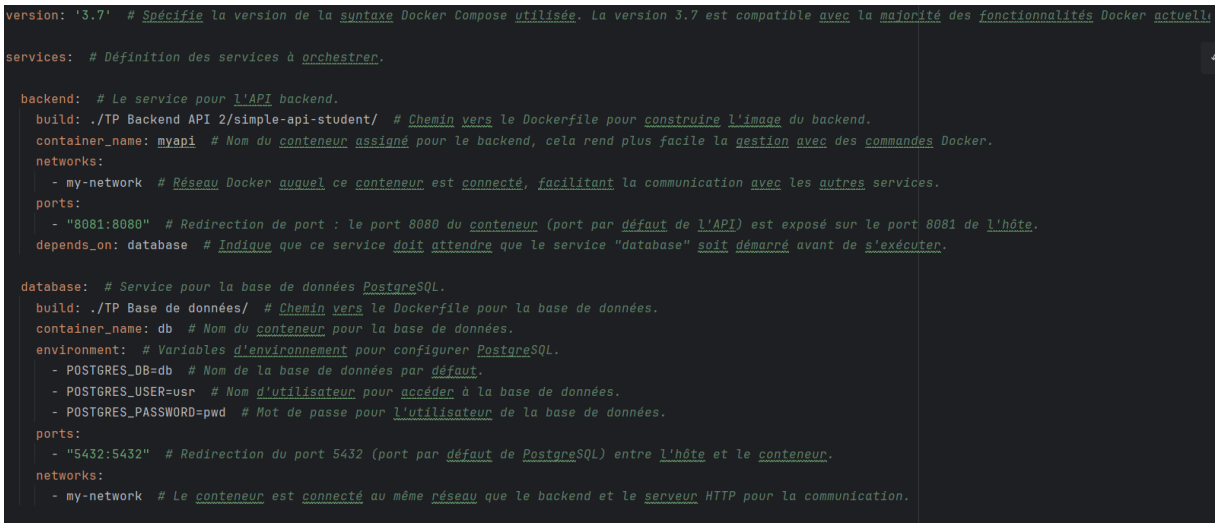
Pourquoi docker-compose est-il si important ?

Docker Compose est essentiel pour orchestrer et gérer facilement plusieurs conteneurs Docker, surtout dans des environnements complexes où plusieurs services interagissent. Il permet de définir tous les services dans un seul fichier docker-compose.yml, simplifiant ainsi la gestion de services tels qu'une API, une base de données, et un serveur web, sans avoir à démarrer chaque conteneur individuellement avec des commandes docker run complexes. Docker Compose assure également une orchestration fluide des dépendances entre services, garantissant qu'un service ne démarre pas avant que ceux dont il dépend ne soient prêts, grâce à la directive depends_on. Cet outil facilite aussi l'isolement des réseaux et des volumes, permettant à chaque service d'avoir son propre environnement, tout en communiquant avec les autres via des réseaux définis. Il prend en charge des scénarios comme le démarrage automatique des services au lancement du projet et la gestion des redémarrages en cas de panne. En outre, Docker Compose est idéal pour les environnements de développement, car il permet de créer des configurations reproductibles pour que l'application fonctionne de la même manière sur toutes les machines, quel que soit le système d'exploitation. Enfin, il permet de simplifier le déploiement et l'évolution des projets, en automatisant la construction, la mise à jour et la mise à l'échelle des services, rendant ainsi l'ensemble de l'écosystème plus cohérent et facile à maintenir.

Question7

Documentez les commandes les plus importantes de docker-compose.

Documentez votre fichier docker-compose.

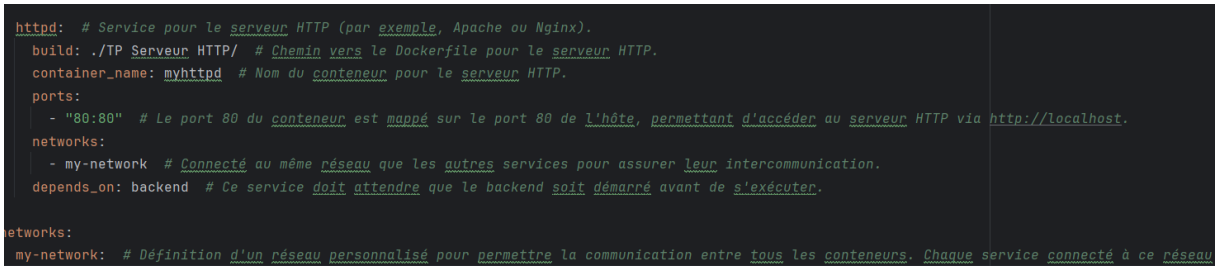


```yaml
version: '3.7'  # Spécifie la version de la syntaxe Docker Compose utilisée. La version 3.7 est compatible avec la majorité des fonctionnalités Docker actuelle

services:  # Définition des services à orchestrer.

  backend:  # Le service pour l'API backend.
    build: ./TP Backend API 2/simple-api-student/  # Chemin vers le Dockerfile pour construire l'image du backend.
    container_name: myapi  # Nom du conteneur assigné pour le backend, cela rend plus facile la gestion avec des commandes Docker.
    networks:
      - my-network  # Réseau Docker auquel ce conteneur est connecté, facilitant la communication avec les autres services.
    ports:
      - "8081:8080"  # Redirection de port : le port 8080 du conteneur (port par défaut de l'API) est exposé sur le port 8081 de l'hôte.
    depends_on: database  # Indique que ce service doit attendre que le service "database" soit démarré avant de s'exécuter.

  database:  # Service pour la base de données PostgreSQL.
    build: ./TP Base de données/  # Chemin vers le Dockerfile pour la base de données.
    container_name: db  # Nom du conteneur pour la base de données.
    environment:  # Variables d'environnement pour configurer PostgreSQL.
      - POSTGRES_DB=db  # Nom de la base de données par défaut.
      - POSTGRES_USER=usr  # Nom d'utilisateur pour accéder à la base de données.
      - POSTGRES_PASSWORD=pwd  # Mot de passe pour l'utilisateur de la base de données.
    ports:
      - "5432:5432"  # Redirection du port 5432 (port par défaut de PostgreSQL) entre l'hôte et le conteneur.
    networks:
      - my-network  # Le conteneur est connecté au même réseau que le backend et le serveur HTTP pour la communication.
```



```yaml
  httpd:  # Service pour le serveur HTTP (par exemple, Apache ou Nginx).
    build: ./TP Serveur HTTP/  # Chemin vers le Dockerfile pour le serveur HTTP.
    container_name: myhttpd  # Nom du conteneur pour le serveur HTTP.
    ports:
      - "80:80"  # Le port 80 du conteneur est mappé sur le port 80 de l'hôte, permettant d'accéder au serveur HTTP via http://localhost.
    networks:
      - my-network  # Connecté au même réseau que les autres services pour assurer leur intercommunication.
    depends_on: backend  # Ce service doit attendre que le backend soit démarré avant de s'exécuter.

networks:
  my-network:  # Définition d'un réseau personnalisé pour permettre la communication entre tous les conteneurs. Chaque service connecté à ce réseau
```

Question 8

Pourquoi mettons-nous nos images dans un dépôt en ligne ?

Mettre nos images Docker dans un dépôt en ligne, comme Docker Hub ou un registre privé, est essentiel pour faciliter le partage et la collaboration entre les équipes, car cela permet à chacun de tirer facilement les images, peu importe leur environnement local. Cela simplifie également le déploiement automatisé dans des pipelines CI/CD, où des outils comme Kubernetes ou Jenkins peuvent récupérer et déployer les dernières versions des images directement depuis le dépôt. De plus, les dépôts en ligne permettent de versionner les images, assurant une gestion claire des différentes itérations de l'application, ce qui est crucial pour les mises à jour et la maintenance. Enfin, cela améliore la portabilité, car les images peuvent être utilisées sur n'importe quel environnement cloud ou machine physique, garantissant une cohérence et une reproductibilité du comportement des applications.