PARIS PANTHÉON-ASSAS UNIVERSITÉ

DevOps & MLOps
TP1

YANG Dominique

BDML -2

1. Introduction

Ce rapport a pour objectif de documenter de manière exhaustive les décisions techniques, les configurations et les justifications relatives à la conteneurisation et au déploiement d'une application multi-services. Il s'appuie sur les réponses fournies concernant les meilleures pratiques Docker, la gestion des données, l'optimisation des images, l'architecture réseau avec reverse proxy, l'orchestration via Docker Compose, et la distribution des images. Ce document sert de référence technique pour le projet.

2. Gestion des Configurations : Variables d'Environnement

Il est préférable d'injecter les variables d'environnement lors de l'exécution d'un conteneur (docker run -e VAR=valeur) plutôt que de les intégrer statiquement dans le Dockerfile (ENV VAR valeur). Les raisons principales sont :

- **Sécurité :** Les variables d'environnement (notamment les secrets comme les mots de passe) définies dans un Dockerfile sont stockées en clair dans les couches de l'image. Si l'image est partagée, ces informations sensibles sont exposées. L'injection à l'exécution évite ce risque.
- **Flexibilité :** Cette approche permet de réutiliser la même image Docker dans différents environnements (développement, test, production) en fournissant des configurations spécifiques à chaque contexte, sans avoir à reconstruire l'image.
- **Séparation des Préoccupations :** Elle favorise une séparation nette entre le code applicatif (contenu dans l'image) et sa configuration (fournie à l'exécution), ce qui est une bonne pratique en ingénierie logicielle.
- **Gestion des Secrets :** L'injection à l'exécution s'intègre plus facilement avec des systèmes de gestion de secrets dédiés tels que Docker Secrets, Kubernetes Secrets, ou HashiCorp Vault.

3. Persistance des Données : Cas du Conteneur PostgreSQL

L'utilisation d'un volume Docker attaché à un conteneur PostgreSQL est essentielle pour les raisons suivantes :

- **Persistance des Données :** Les conteneurs Docker sont éphémères par nature. Sans volume, toutes les données écrites par PostgreSQL (bases de données, tables, etc.) seraient perdues à la suppression du conteneur. Les volumes persistent au-delà du cycle de vie du conteneur.

- **Consistance des Données :** Un volume garantit que les données de la base restent intactes et cohérentes lors des redémarrages, des mises à jour ou des remplacements du conteneur PostgreSQL.
- **Performance :** PostgreSQL est un SGBDR optimisé pour des opérations d'I/O sur des systèmes de fichiers persistants. Les volumes permettent d'utiliser les pilotes de stockage de l'hôte et de bénéficier de ses optimisations.
- **Sauvegarde et Restauration :** Les volumes facilitent grandement les opérations de sauvegarde et de restauration des données, car les fichiers de la base de données sont accessibles directement depuis le système de fichiers de l'hôte (ou le stockage réseau mappé).

4. Documentation des Essentiels du Conteneur de Base de Données

La mise en place du service de base de données PostgreSQL et de son environnement implique plusieurs étapes et commandes Docker. Voici une documentation de ces éléments essentiels :

1. Création du Réseau Docker :
Un réseau dédié est créé pour permettre la communication entre les conteneurs de l'application.

- docker network create app-network

2. Construction de l'Image PostgreSQL :
Si une image PostgreSQL personnalisée est nécessaire (par exemple, pour inclure des scripts d'initialisation), elle est construite avec :

- docker build -t postgres-db .

Un Dockerfile typique pour postgres-db pourrait se baser sur une image officielle postgres et copier des scripts .sh ou .sql dans /docker-entrypoint-initdb.d/ pour l'initialisation automatique de la base.

3. Exécution du Conteneur PostgreSQL :
Le conteneur est lancé avec les paramètres suivants :

```
docker run -d \
--name postgres-db \
--network app-network \
-v postgres-data:/var/lib/postgresql/data \
-e POSTGRES_DB=db \
-e POSTGRES_USER=postgres \
-e POSTGRES_PASSWORD=root \
postgres-db
```

- -d: Exécution en mode détaché.
- --name postgres-db: Nom du conteneur.
- --network app-network: Connexion au réseau applicatif.
- -v postgres-data:/var/lib/postgresql/data: Mappage du volume nommé postgres-data pour la persistance.
- -e POSTGRES_*: Variables d'environnement pour configurer l'instance PostgreSQL à sa création.

4. Autres Commandes :
Les commandes pour Adminer, le backend Java simple, et l'API Spring Boot font partie de la configuration globale de l'application et sont documentées dans leurs sections respectives ou dans la section Docker Compose.

5. Optimisation des Images : Utilité et Mécanisme des Builds Multistages

Les builds multistages sont une technique Docker cruciale pour optimiser les images.

1. Pourquoi sont-ils nécessaires ?

- **Réduction de la Taille d'Image :** L'image finale ne contient que les artefacts et l'environnement d'exécution stricts (ex: JRE, binaire applicatif), en excluant les dépendances de build (ex: JDK, Maven, code source, fichiers intermédiaires).
- **Sécurité Renforcée :** Une image plus petite avec moins de composants signifie une surface d'attaque réduite.
- **Meilleure Organisation :** Le Dockerfile est structuré plus clairement, séparant l'environnement de construction de l'environnement d'exécution.
- **Performance Accrue :** Des images plus petites se traduisent par des temps de pull/push et de déploiement plus rapides, ainsi qu'une consommation réduite des ressources disque et réseau.

2. Explication des Étapes d'un Dockerfile Multistage typique (pour une application Java/Maven) :

- **Étape 1 : Environnement de Build (AS builder)**
    1. FROM maven:tag ou FROM jdk-image AS builder: Utilise une image contenant le JDK complet et les outils de build (comme Maven).
    2. WORKDIR /app: Définit le répertoire de travail.
    3. COPY pom.xml .: Copie d'abord le pom.xml pour exploiter le cache Docker. Si le pom.xml ne change pas, les dépendances ne sont pas retéléchargées.
    4. RUN mvn dependency:go-offline (ou mvn verify clean --fail-never pour télécharger les dépendances).
    5. COPY src ./src: Copie le reste du code source.

6. RUN mvn package -DskipTests: Compile l'application et produit l'artefact exécutable (ex: un fichier JAR).

- **Étape 2 : Environnement d'Exécution (Image Finale)**
  1. FROM openjdk:tag-jre-slim ou FROM image-jre-minimale: Démarre à partir d'une image JRE minimale, beaucoup plus légère que l'image JDK.
  2. WORKDIR /app: Définit le répertoire de travail.
  3. COPY --from=builder /app/target/*.jar app.jar: Copie *uniquement* l'artefact JAR construit lors de l'étape précédente (builder) dans l'image finale.
  4. EXPOSE <port>: Documente le port sur lequel l'application écoute.
  5. ENTRYPOINT ["java", "-jar", "app.jar"]: Configure la commande pour démarrer l'application lors de l'exécution du conteneur.
     L'image finale ne contient ni Maven, ni le JDK, ni le code source, ni les fichiers de build intermédiaires.

6. Rôle et Mise en Place du Reverse Proxy

Un reverse proxy est un composant d'infrastructure essentiel dans les architectures web modernes.

1. Pourquoi est-il nécessaire ?

- **Amélioration de la Sécurité :** Il agit comme une barrière entre Internet et les serveurs backend, masquant leur topologie et leurs adresses IP directes, et peut filtrer le trafic malveillant.
- **Répartition de Charge (Load Balancing) :** Distribue les requêtes entrantes entre plusieurs instances de serveurs applicatifs, améliorant la performance, la disponibilité et la scalabilité.
- **Terminaison SSL/TLS :** Centralise la gestion du chiffrement/déchiffrement HTTPS, déchargeant les serveurs backend de cette tâche et simplifiant la gestion des certificats.
- **Mise en Cache de Contenu :** Peut mettre en cache le contenu statique (images, CSS, JS) ou même des réponses dynamiques, réduisant la charge sur les backends et accélérant les temps de réponse.
- **Simplification de l'Architecture et Point d'Entrée Unifié :** Fournit une URL unique pour accéder à divers services backend, même s'ils sont hébergés sur des serveurs ou des ports différents.
- **Optimisation des Ressources et des Requêtes :** Peut effectuer la compression/décompression des données, la réécriture d'URL, etc.
- **Surveillance et Journalisation Centralisées :** Offre un point unique pour collecter les logs de trafic et surveiller l'état des services.

2. Documentation des Commandes pour la Mise en Place d'un Reverse Proxy Apache :

1. **Exécution d'un conteneur Apache temporaire :**

   docker run -d --name httpd-temp httpd:2.4-alpine

2. **Copie du fichier de configuration httpd.conf depuis le conteneur vers l'hôte :**

   docker cp httpd-temp:/usr/local/apache2/conf/httpd.conf ./httpd/

   Ce fichier est ensuite modifié pour inclure les directives de proxy (ex: ProxyPass, ProxyPassReverse).

3. **Suppression du conteneur temporaire :**

   docker rm -f httpd-temp # Le nom du conteneur doit correspondre, celui-ci était httpd-proxy dans l'exemple, mais ici httpd-temp

4. **Construction de l'image du reverse proxy personnalisée :**

   docker build -t httpd-proxy ./httpd

5. **Exécution du conteneur reverse proxy final :**

   docker run -d --name httpd-proxy --network app-network -p 80:80 httpd-proxy

7. Importance Stratégique de Docker Compose pour l'Orchestration

Docker Compose est un outil crucial pour définir et exécuter des applications multi-conteneurs. Son importance réside dans :

- **Simplification de l'Orchestration :** Il permet de gérer des applications complexes composées de plusieurs services (conteneurs) interdépendants (ex: web, API, base de données) comme une seule unité.
- **Configuration Déclarative :** Toute l'infrastructure de l'application (services, réseaux, volumes, variables d'environnement, dépendances) est définie dans un unique fichier YAML (docker-compose.yml), ce qui rend la configuration lisible, versionnable et reproductible.
- **Automatisation des Réseaux :** Crée automatiquement un réseau par défaut pour l'application, permettant aux services de se découvrir et de communiquer par leurs noms de service.

- **Cohérence des Environnements :** Garantit que l'application s'exécute de manière identique dans différents environnements (développement, test, production) à partir du même fichier de configuration.
- **Gestion des Dépendances :** Permet de définir l'ordre de démarrage des services (ex: la base de données doit démarrer avant l'application).
- **Commandes Unifiées :** Offre des commandes simples (docker-compose up, docker-compose down) pour démarrer, arrêter, et gérer l'ensemble de l'application.
- **Gestion Optimisée des Volumes et de la Persistance des Données.**
- **Configuration Facilitée des Variables d'Environnement pour chaque service.**

8. Commandes Docker Compose Fondamentales

Voici une liste des commandes Docker Compose les plus importantes :

- docker-compose up: Crée et démarre les conteneurs définis dans le fichier docker-compose.yml. Les logs sont affichés dans le terminal.
- docker-compose up -d: Démarre les conteneurs en arrière-plan (mode détaché).
- docker-compose down: Arrête et supprime les conteneurs, les réseaux créés par up. Par défaut, les volumes nommés ne sont pas supprimés.
- docker-compose down -v: Arrête et supprime les conteneurs, les réseaux, ET les volumes nommés associés à l'application.
- docker-compose build [service_name]: Construit (ou reconstruit) les images pour les services qui ont une section build (ou pour un service spécifique).
- docker-compose logs [-f] [service_name]: Affiche les logs des services (ou d'un service spécifique). L'option -f suit les logs en continu.
- docker-compose ps: Liste les conteneurs en cours d'exécution pour le projet Compose.
- docker-compose restart [service_name]: Redémarre les services.
- docker-compose stop [service_name]: Arrête les services sans les supprimer. Ils peuvent être redémarrés avec docker-compose start.
- docker-compose exec <service_name> <commande>: Exécute une commande à l'intérieur d'un conteneur de service en cours d'exécution.
- docker-compose config: Valide et affiche la configuration Compose effective (après fusion des fichiers et résolution des variables).

9. Documentation du Fichier docker-compose.yml de l'Application

Le fichier docker-compose.yml de ce projet orchestre une application à trois niveaux principaux. Sa structure est la suivante :

- **Version :** Spécifie la version du format de fichier Docker Compose utilisée.
- **Services :**
    - **Service backend (ou backend-api) :**

        - build: Construit à partir d'un Dockerfile situé dans un sous-répertoire (ex: ./simple-api). Utilise un build multistage pour l'optimisation.
        - container_name: Un nom explicite pour le conteneur (ex: backend-api-container).
        - restart: Politique de redémarrage (ex: unless-stopped).
        - environment: Variables d'environnement pour la configuration de l'application, notamment les identifiants de connexion à la base de données (SPRING_DATASOURCE_URL, etc.).
        - depends_on: Spécifie que ce service dépend du service database et doit démarrer après lui.
        - networks: Connecté au réseau app-network.
        - volumes: Potentiellement, un volume pour des données persistantes spécifiques à l'application si nécessaire (non mentionné pour les données de base).
    - **Service database :**
        - build: Construit à partir d'un Dockerfile situé à la racine du projet ou dans un sous-répertoire dédié, basé sur une image PostgreSQL.
        - image: Alternativement, peut utiliser directement une image pré-construite (ex: postgres:latest ou lytwz1/devops-database:1.0).
        - container_name: Un nom explicite (ex: postgres-db-container).
        - restart: Politique de redémarrage.
        - environment: Variables d'environnement pour l'initialisation de PostgreSQL (POSTGRES_DB, POSTGRES_USER, POSTGRES_PASSWORD).
        - volumes: Un volume nommé (ex: postgres-data) est mappé sur /var/lib/postgresql/data pour garantir la persistance des données de la base.
        - networks: Connecté au réseau app-network.
    - **Service httpd (Reverse Proxy) :**
        - build: Construit à partir d'un Dockerfile situé dans un sous-répertoire (ex: ./httpd), qui copie un fichier httpd.conf personnalisé.
        - container_name: Un nom explicite (ex: httpd-proxy-container).
        - restart: Politique de redémarrage.
        - ports: Expose le port 80 du conteneur sur le port 80 de l'hôte. C'est le seul port directement exposé à l'extérieur.
        - depends_on: Spécifie que ce service dépend du service backend.
        - networks: Connecté au réseau app-network.
        - volumes: Un volume peut être utilisé pour monter le fichier httpd.conf (souvent en lecture seule) et potentiellement pour du contenu statique ou des certificats SSL.
- **Réseaux (networks) :**
    - app-network: Un réseau de type bridge est défini pour permettre la communication isolée et la résolution de noms entre les services.
- **Volumes (volumes) :**

- postgres-data: Un volume nommé est déclaré pour être utilisé par le service database pour la persistance des données. D'autres volumes peuvent être déclarés si nécessaire pour d'autres services.

Cette structure permet de définir, configurer et lier tous les composants de l'application de manière déclarative.

10. Publication des Images Docker : Procédures et Justifications

10.1. Commandes de Publication et Images Publiées

Le processus de publication des images Docker construites sur un registre distant comme Docker Hub comprend les étapes suivantes :

1. **Connexion à Docker Hub (ou autre registre) :**

   docker login -u lytwz1 # Remplacer lytwz1 par le nom d'utilisateur du registre

2. **Étiquetage (Tagging) des Images Locales :**
   Avant de pouvoir pousser une image, elle doit être étiquetée avec le nom d'utilisateur du registre, le nom de l'image et une version (tag).

   docker tag devops-mlops-database lytwz1/devops-database:1.0
   docker tag devops-mlops-backend lytwz1/devops-backend:1.0
   docker tag devops-mlops-httpd lytwz1/devops-httpd:1.0

3. **Publication (Push) des Images sur Docker Hub :**

   docker push lytwz1/devops-database:1.0
   docker push lytwz1/devops-backend:1.0
   docker push lytwz1/devops-httpd:1.0

4. **Images publiées sur Docker Hub (selon la documentation) :**

   lytwz1/devops-database:1.0 : Image de la base de données PostgreSQL, potentiellement avec un schéma initial.

   lytwz1/devops-backend:1.0 : Image de l'API Spring Boot pour la gestion des étudiants et départements.

   lytwz1/devops-httpd:1.0 : Image du serveur Apache configuré comme reverse proxy.

10.2. Pourquoi Mettre les Images dans un Dépôt en Ligne ?

Stocker les images Docker dans un dépôt en ligne (registre Docker) offre de multiples avantages :

- **Partage et Collaboration :** Facilite le partage des images standardisées entre les membres de l'équipe de développement et avec d'autres équipes (QA, Ops).
- **Déploiement Multi-Environnements :** Permet de déployer la même image testée et validée sur différents environnements (développement, staging, production) sans avoir à reconstruire l'image localement sur chaque serveur.
- **Intégration Continue / Déploiement Continu (CI/CD) :** Les registres d'images sont un composant essentiel des pipelines CI/CD. Les images construites et testées sont poussées vers le registre, d'où elles peuvent être tirées pour le déploiement.
- **Versionnement et Traçabilité :** Permet de gérer différentes versions (tags) des images, facilitant le suivi des modifications, les rollbacks vers des versions précédentes et l'audit.
- **Sauvegarde Centralisée :** Agit comme un dépôt centralisé et une sauvegarde pour les artefacts de build (images Docker).
- **Distribution Simplifiée :** Simplifie le déploiement sur de multiples serveurs, des clusters d'orchestration (comme Kubernetes) ou des plateformes cloud.
- **Mise à l'Échelle Horizontale :** Facilite le scaling des applications en permettant à de nouvelles instances de conteneurs de tirer rapidement l'image requise.
- **Reproductibilité des Déploiements :** Assure que les déploiements sont basés sur des images bien définies et immuables, garantissant la cohérence.

11. Conclusion

Ce rapport a documenté les aspects clés de la stratégie de conteneurisation adoptée, couvrant la configuration, la persistance, l'optimisation, l'architecture réseau, l'orchestration et la distribution. Les choix effectués visent à maximiser la sécurité, la flexibilité, la maintenabilité et l'efficacité opérationnelle de l'application. L'utilisation combinée de Docker, Docker Compose et d'un registre d'images constitue une fondation solide pour le cycle de vie de l'application.