TP – Déploiement automatisé d'une application dans une démarche DevOps

🎯 Objectifs opérationnels

L'objectif de ce TP est de **déployer une application de manière entièrement automatisée**, selon les principes de l'approche DevOps. Vous allez mettre en place une infrastructure légère mais complète, permettant :

- le développement et les tests dans un environnement isolé,
- l'intégration continue
- l'analyse de code automatisée (SAST),
- et le déploiement sur un environnement distant simulé.

Le tout en conteneurs, via une approche **Infrastructure as Code** (IaC), avec **Docker Compose** comme orchestrateur local.

⚙️ Ce que nous allons construire

Nous allons déployer une infrastructure composée des services suivants :

1. 🖥️ `desktop` – Environnement graphique de pilotage

Un conteneur Linux complet, doté d'un environnement graphique **LXDE** accessible via VNC dans un navigateur.

- Accessible depuis la machine hôte à l'adresse : `http://localhost:80`
- Permet de piloter les autres conteneurs sans se heurter aux problèmes de `localhost` ou de `hostname` côté hôte.
- Utilisé comme **point d'entrée universel** pour l'ensemble de l'infrastructure (dev, tests, déploiement, monitoring).

2. 🚢 `plateforme` – Environnement de déploiement

Conteneur basé sur **Alpine Linux**, extrêmement léger.

- Contient **OpenSSH**, **Docker**, **Docker-Compose**, **Git**.
- Joue le rôle de **serveur cible de déploiement** pour les pipelines Jenkins.
- Simule un serveur distant dans une architecture plus large.

3. 🔧 `jenkins` – Serveur d'intégration continue

Instance locale de **Jenkins LTS**.

- Fournit l'interface d'orchestration CI/CD.
- Pilote les builds, tests et déploiements à partir de pipelines Jenkinsfile.

- Utilise les dépôts Git (auto-hébergés ou distants) comme source de code et d'infrastructure (`docker-compose.yaml`, etc.).

4. 🚀 `jenkins-agent-docker` – Agent Docker Jenkins

Un **agent Jenkins rooté** exécuté dans un conteneur Docker, capable de lancer d'autres conteneurs Docker grâce à l'accès au socket Docker de l'hôte (`/var/run/docker.sock`).

- Permet de builder des images Docker, lancer des tests automatisés, et déployer sur les environnements cibles (ex. : le conteneur `plateforme`).
- Utilise le plugin "Docker Pipeline" pour exécuter dynamiquement des étapes Docker dans les pipelines.
- Fonctionne en mode **inbound agent**, connecté automatiquement au Jenkins master via un `secret`.
- Rôle clé dans l'infrastructure CI/CD : **centralise l'exécution des builds Docker dans un environnement isolé mais privilégié**, sans compromettre l'isolation du maître Jenkins.
- Ce setup assure une séparation claire entre l'orchestration (master Jenkins) et l'exécution (agents), tout en gardant une capacité complète d'exécution de conteneurs.

5. 🧪 `sonarqube` – Analyse statique de code (SAST)

Serveur **SonarQube** configuré pour recevoir et afficher les résultats d'analyse de code.

- Permet d'assurer la **qualité logicielle** dès la phase d'intégration.
- S'intègre dans le pipeline CI pour lancer automatiquement les analyses lors des pushs.

Le déploiement de ce composant n'est pas une priorité mais est une amélioration bienvenue.

6. 🗃️ `gitea` – Serveur Git auto-hébergé

Instance locale de **Gitea**, un gestionnaire de dépôts Git léger et rapide.

- Fournit une interface web pour la gestion du code source, des issues et des pull requests.
- Sert de **point central** pour les dépôts utilisés par Jenkins (via Webhooks ou polling Git).
- Remplace GitLab dans un setup plus minimaliste, tout en restant compatible avec les workflows Git classiques (CI/CD, forks, branches...).

🧭 Démarche pédagogique

Dans ce TP, vous allez :

- Définir l'infrastructure dans un fichier `docker-compose.yaml` unique (IaC),
- Générer une clé SSH et la configurer pour permettre les déploiements sans mot de passe,
- Écrire un pipeline CI/CD complet (build + analyse + déploiement),
- Valider que le déploiement vers le conteneur cible fonctionne sans intervention manuelle.

💬 En résumé

Une mini-infrastructure DevOps auto-hébergée, en pur Docker, pilotée graphiquement depuis un conteneur VNC, permettant un enchaînement CI/CD complet : du commit à la mise en production.

> 🧠 *Le tout, sans jamais rien installer sur l'hôte, hormis Docker lui-même.*

Disclaimer: La mise en forme de ce sujet a en partie été réalisée avec l'aide d'un agent conversationnel.

🔨 Travail à faire

Étape 0 - Prérequis

Avant de commencer, assurez-vous que les prérequis suivants sont satisfaits pour pouvoir exécuter les TP dans de bonnes conditions.

- 🐧 **Système Linux requis** : Les exercices doivent être réalisés sur un système Linux, avec un accès root ou `sudo`.
- 🐳 **Docker installé** : L'environnement doit disposer de Docker et Docker Compose installés.
  - Vérifiez l'installation avec :

```
docker --version
docker compose version
```

- 🧱 **Définition des conteneurs via Docker Compose** :
  - Tous les services seront définis dans un fichier `docker-compose.yaml`.
  - Cela permet de centraliser la configuration et de démarrer tous les services avec une seule commande :

```
docker compose up -d
```

- 🌐 **Nom du service = nom d'hôte** :
  - Docker Compose attribue automatiquement à chaque conteneur un nom d'hôte correspondant à son nom de service.
  - Exemple : un service nommé `jenkins` sera joignable depuis un autre conteneur avec l'URL `http://jenkins:8080`.
- 💾 **Volumes ou montages liés pour la persistance** :
  - Il est important de **conserver les données** (configuration Jenkins, jobs, plugins, logs…) entre les redémarrages ou suppressions de conteneurs.
  - Pour cela, utilisez :
    - des volumes Docker :

```
volumes:
  - jenkins_home:/var/jenkins_home
```

- ou des montages liés (bind mounts) :

```
volumes:
  - ./jenkins_data:/var/jenkins_home
```

  - Cela évite de perdre les données en cas de recréation du conteneur.

- ✅ Vérifiez que les ports nécessaires (par ex. `8080` pour Jenkins) sont libres sur votre machine.

Étape 1 — Déploiement du service `desktop`

Ce service correspond à une machine Linux graphique accessible via VNC dans un navigateur. Il servira de poste d'administration pour interagir avec les autres conteneurs de l'infrastructure.

Instructions

1. Dans votre fichier `docker-compose.yml`, définis un service nommé `desktop` basé sur l'image Docker publique **`dorowu/ubuntu-desktop-lxde-vnc`**.
2. Je vous conseille de créer votre conteneur à partir d'un Dockerfile afin d'y inclure les commandes suivantes:

```
rm -f /etc/apt/sources.list.d/google-chrome.list && \
apt-get update && \
apt-get install -y firefox unzip git nano
```

3. Configurer :
   - Un mappage de port pour rendre le bureau accessible via `http://localhost`.
   - Deux variables d'environnement obligatoires : `USER` et `PASSWORD`
   - Un redémarrage automatique du conteneur avec la directive `restart: unless-stopped`.
4. Lancer le service avec la commande suivante :

```
docker compose up -d --build --force-recreate
```

   - `--build` : force la reconstruction des images (utile en cas de modification).
   - `--force-recreate` : force la recréation des conteneurs même s'ils existent déjà.
5. Vérifier le bon démarrage du conteneur :

```
docker compose ps
```

6. Une fois lancé, accèder à l'interface graphique via ton navigateur à l'adresse suivante :

   ```
   http://localhost
   ```

   Vous devriez voir s'afficher un bureau LXDE directement dans le navigateur.

🔐 **Remarque sur la sécurité**

Les variables `USER` et `PASSWORD` sont visibles en clair dans le fichier `docker-compose.yml`.
Ceci est **acceptable ici**, dans un cadre **local, pédagogique**, sans enjeu de sécurité. En production, on privilégierait un stockage des secrets via des volumes chiffrés, des fichiers `.env` ignorés du VCS, ou des gestionnaires de secrets (Vault, AWS Secrets Manager...).

Étape 2 — Déploiement du service `plateforme`

Ce service représente un environnement de déploiement **"prod-like"**, à partir duquel nous pourrons simuler des actions DevOps (ex. : déploiement applicatif, exécution de conteneurs, etc.).

🔧 Image personnalisée à construire

Contrairement au service `desktop`, ici **aucune image préexistante** ne répond à nos besoins.
Nous allons donc **construire notre propre image Docker** à l'aide d'un `Dockerfile`.

Ce fichier permettra notamment de :

- Créer un utilisateur (ex. `jenkins`) pouvant se connecter en SSH,
- Installer les outils nécessaires (`openssh`, `docker-cli`, `git`, etc.),
- Donner les droits à cet utilisateur pour interagir avec le socket Docker de l'hôte.

⚠ Pas de mappage de port par défaut

Par souci de sécurité, **aucun port n'est exposé** vers la machine hôte.
Cependant, pour déboguer une connexion SSH (ex. via `ssh jenkins@localhost -p 2222`), vous pouvez temporairement ajouter un mappage de port dans le `docker-compose.yml` :

```
ports:
  - "2222:22"
```

🐳 DoD — Docker-outside-of-Docker

Nous souhaitons que le conteneur `plateforme` puisse exécuter des commandes Docker (ex. : `docker run`, `docker build`, etc.).

Pour cela, on **monte le socket Docker de l'hôte dans le conteneur** :

```
volumes:
  - /var/run/docker.sock:/var/run/docker.sock
```

C'est ce qu'on appelle le **Docker-outside-of-Docker (DoD)**.
Cela permet au conteneur de piloter le moteur Docker de l'hôte, **sans avoir Docker installé dans le conteneur lui-même**.

> 🛑 **Attention** : ce montage donne des **droits élevés** sur le démon Docker de l'hôte.
> À ne jamais faire en production sans précautions.

🔐 Authentification SSH sécurisée

La connexion SSH en tant que `root` est **désactivée par défaut** (bonne pratique de sécurité).

Nous devons donc créer un **utilisateur dédié** (dans l'exemple `jenkins`) pour se connecter via SSH.

Cet utilisateur devra **faire partie du groupe `docker`** pour accéder au socket monté.

Génération d'une paire de clés SSH pour authentification client/serveur

Pour sécuriser l'accès SSH entre votre pipeline (client) et le serveur distant, il est recommandé d'utiliser une paire de clés SSH publique/privée.

Exemple de commande pour générer la paire de clés

```
ssh-keygen -t ed25519 -C "jenkins" -f ~/.ssh/jenkins
```

Explications des paramètres

- `-t ed25519` : spécifie le type de clé à générer. Ed25519 est un algorithme moderne, plus sûr et plus performant que RSA, avec des tailles de clés plus petites et une meilleure résistance aux attaques.
- `-C "jenkins"` : ajoute un commentaire dans la clé publique, ici "jenkins", qui permet d'identifier facilement cette clé, utile dans les fichiers `authorized_keys`.
- `-f ~/.ssh/jenkins` : définit le chemin et le nom du fichier de la clé privée générée. La clé publique correspondante sera automatiquement créée avec le suffixe `.pub` (ici `~/.ssh/jenkins.pub`).

Déploiement des clés

- La clé publique (`jenkins.pub`) doit être copiée sur le serveur distant dans le fichier `~/.ssh/authorized_keys` de l'utilisateur qui sera utilisé pour la connexion SSH. Cet utilisateur doit exister sur le serveur et correspondre à l'identifiant que vous utilisez dans la connexion SSH.
- La clé privée (`jenkins`) doit rester secrète et sera injectée plus tard dans l'`agent jenkins docker` de votre pipeline pour procéder au déploiement. Cela garantit que seul le pipeline possède l'accès privé nécessaire pour s'authentifier auprès du serveur.

Cette méthode évite l'utilisation de mots de passe en clair et renforce la sécurité des connexions automatisées entre votre pipeline et vos serveurs.

🧪 Dockerfile (à compléter)

Vous trouverez ci-dessous un squelette de Dockerfile.

Complétez-le progressivement — chaque instruction est commentée pour vous guider :

```
FROM alpine:latest

# TODO: Définir ici une variable d'enrionnement "USER" avec le user ssh

# TODO: Installer les outils nécessaires
RUN apk add --no-cache ...TODO...

# Création de l'utilisateur ssh et configuration SSH de base
RUN adduser -D $USER \
    && echo "$USER:$USER" | chpasswd \
    && mkdir -p /home/$USER/.ssh \
    && chown -R $USER:$USER /home/$USER \
    && chmod 700 /home/$USER/.ssh \
    && addgroup -S docker && adduser $USER docker

# TODO: Copier ici la clé SSH public dans
/home/$USER/.ssh/authorized_keys. N.B: peut-être allez-vous devoir adapter
les permissions...

# Configuration minimale de SSHD
RUN ssh-keygen -A \
    && echo "PermitRootLogin no" >> /etc/ssh/sshd_config \
    && echo "PasswordAuthentication no" >> /etc/ssh/sshd_config \
    && echo "PubkeyAuthentication yes" >> /etc/ssh/sshd_config \
    && echo "AllowUsers $USER" >> /etc/ssh/sshd_config

# TODO: Ajouter métadonnée indiquant le port exposé

# TODO: Donner les droits d'accès au socket pour le groupe "docker" et
lancer le serveur SSH.
CMD sh -c "...a_completer... && /usr/sbin/sshd -D"
```

Ce Dockerfile est ensuite à intégrer dans votre `docker-compose.yaml`.

Etape 3 : Test du serveur SSH sur le service `plateforme`

N.B: Vous pouvez réaliser cette étape depuis votre machine hôte également. Dans ce cas pensez à réaliser un port forwarding.

1. Sur l'application desktop, ouvrir un terminal.

2. Installer un client SSH si ce n'est pas déjà fait :

```
sudo apt install ssh
```

3. Vérifier que l'agent SSH est bien lancé :

```
eval "$(ssh-agent -s)"
```

4. Copier le contenu de la clé privée créée à l'étape précédente sur le conteneur "desktop" dans un fichier, par exemple nommé `jenkins`.

5. Modifier les permissions de la clé privée pour sécuriser l'accès :

```
sudo chmod 600 jenkins
```

6. Importer la clé privée dans l'agent SSH :

```
ssh-add jenkins
```

7. Vous pouvez maintenant vous connecter sur votre serveur de déploiement :

```
ssh jenkins@plateforme
```
