Exercice : Mise en place d'une pipline CI avec Docker et GitHub Actions

Partie 1 : Création et Dockerisation de l'Application Flask en Local

1. **Installation de l'environnement de développement :**
   - Installer Python et PIP (si ce n'est pas déjà fait) sur votre machine.
   - Créer un dossier de travail pour votre projet localement, par exemple `FlaskApp`.

2. **Création de l'application Flask :** Créer une application Flask simple qui renverra un message "Hello, World !" lorsqu'on accède à la racine du site. Pour s'y faire, il faut créer trois fichiers dans le dossier `FlaskApp` :
   - `app.py` : contient le code de l'application.

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello_world():
    return 'Hello, World!'
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

   - `requirements.txt` : liste des dépendances nécessaires pour l'application.

```
Flask==3.0.3
pytest==7.4.2
```

   - `test_app.py` : fichier de tests automatisés pour vérifier que l'application fonctionne correctement.

```
import pytest
from app import app

def test_index():
    tester = app.test_client()
    response = tester.get('/')
    assert response.status_code == 200
    assert b"Hello, World!" in response.data
```

3. **Exécution de l'application localement :**
   - Dans votre terminal, naviguer vers le dossier de votre projet.
   - Installer les dépendances de Flask en exécutant la commande : pip install -r requirements.txt
   - Exécuter l'application Flask en local avec : python app.py
   - Vérifiez que l'application fonctionne correctement en accédant à `http://localhost:5000` dans votre navigateur, et assurez-vous que le message "Hello, World !" s'affiche.

4. **Création du dépôt local Git :**

- Initialiser un dépôt Git dans votre projet, ajouter les fichiers du projet à votre dépôt local et valider ces actions en mettant le message suivant : "Initial commit - Flask app and tests".

Partie 2 : Containerisation de l'application avec Docker (en local)

1. **Création du Dockerfile :** Créer un fichier `Dockerfile` dans votre projet pour containeriser l'application Flask. Ce fichier doit inclure :
   - L'installation des dépendances Flask.
   - La copie du code dans le conteneur.
   - La spécification du port sur lequel l'application Flask écoute : 5000
   - L'exécution de l'application Flask.

2. **Test Local avec Docker :**

- Construisez et exécutez l'image Docker localement.
- Vérifiez à nouveau que l'application fonctionne en accédant à `http://localhost:5000`.

Partie 3 : Dépôt GitHub et CI avec GitHub Actions

1. Créer un dépôt vide sur GitHub nommé `FlaskApp`.
2. Associez votre dépôt local à GitHub.
3. Ajout d'un pipeline CI avec GitHub Actions :
   - Créez un dossier `.github/workflows` dans votre projet local, puis ajoutez un fichier `ci.yml` pour configurer le pipeline CI.

Guide

GitHub Actions est un service d'intégration continue (CI) et de déploiement continu (CD) intégré à GitHub. Il permet d'automatiser divers workflows de développement, tels que la construction, le test et le déploiement d'applications, directement depuis un dépôt GitHub. Voici un aperçu des concepts clés d'un pipeline CI avec GitHub Actions et comment le configurer.

Concepts de base de GitHub Actions :

- Workflow : Un workflow est une série d'actions définies dans un fichier YAML. Il décrit les étapes à suivre lorsqu'un événement se produit (par exemple, un push dans le dépôt).
- Événements : Les workflows s'exécutent en réponse à des événements, comme : push, pull request, etc.
- Jobs : Un workflow peut contenir plusieurs jobs, chacun exécutant une série d'étapes. Les jobs peuvent s'exécuter en parallèle ou en séquence.
- Steps : Ce sont les actions individuelles à l'intérieur d'un job. Chaque étape peut exécuter une commande ou appeler une action (une tâche prédéfinie) disponible sur GitHub.
- Actions : Ce sont des scripts réutilisables qui peuvent être appelés dans les étapes d'un job. GitHub Actions offre une bibliothèque d'actions prédéfinies, et vous pouvez également créer vos propres actions.

Exemple d'un pipeline CI avec GitHub Actions : Voici comment configurer un pipeline CI simple pour une application Python Flask :

```yaml
name: CI Pipeline
  on:
    push:
      branches:
        - main
    pull_request:
      branches:
        - main
  jobs:
    build:
      runs-on: ubuntu-latest
      steps:
      - name: Checkout code
        uses: actions/checkout@v3
      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.13'  # spécifiez la version de Python

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt  # assurez-vous d'avoir ce fichier

      - name: Run tests
        run: |
          pytest  # ou toute autre commande pour exécuter vos tests
```

- name : Nom du workflow.
- on : Indique quand le workflow doit être exécuté (ici, lors des pushes ou des pull requests sur la branche `main`).
- jobs : Définit les tâches à exécuter.
- runs-on : Spécifie l'environnement sur lequel le job s'exécute, ici `ubuntu-latest`.
- Steps : Définit chaque étape du job, comme :
  * actions/checkout : Récupère le code du dépôt.
  * actions/setup-python : Configure l'environnement Python.
  * Installation des dépendances et exécution des tests.

- Après avoir ajouté et enregistré le fichier `ci.yml`, effectuez un commit et poussez-le vers votre dépôt GitHub.
- Vérifier l'exécution : Allez dans l'onglet "Actions" de votre dépôt sur GitHub. Vous devriez voir le workflow s'exécuter. Si tout est configuré correctement, il va passer les tests définis. GitHub Actions s'exécutera automatiquement lors du push, en testant votre application et en construisant l'image Docker.
- Modifier le fichier ci.yml pour inclure une étape qui pousse l'image sur Docker Hub après avoir été testée. Ceci vous permet de sauvegarder et réutiliser les images Docker.
- Ajoutez vos informations d'authentification Docker Hub dans GitHub pour que GitHub Actions puisse y pousser les images : Allez dans votre dépôt GitHub > Settings > Secrets and variables > Actions > "New repository secret".
    - Nommez le secret DOCKER_USERNAME avec votre nom d'utilisateur Docker.
    - Nommez le secret DOCKER_PASSWORD avec votre mot de passe Docker.

- Pourquoi devons-nous utiliser les **secrets** GitHub (DOCKER_USERNAME, DOCKER_PASSWORD) dans le fichier ci.yml pour l'authentification à Docker Hub ?
- Quels sont les **risques** associés au stockage des informations sensibles dans des pipelines CI/CD ? Quelles sont les meilleures pratiques pour limiter ces risques ?
- Pousser les modifications de votre fichier ci.yml sur Github et vérifier l'exécution du workflow dans l'onglet Actions sur Github.
- Si votre organisation utilise plusieurs registres Docker (par exemple Docker Hub et un registre privé), comment pourriez-vous **pousser l'image Docker vers plusieurs registres** dans une même action GitHub ? Modifier le pipeline pour inclure une étape qui pousse l'image Docker vers deux registres différents (sans tester).
- Dans le fichier ci.yml, le workflow est déclenché lors d'un push sur la branche main. Modifier le pour déclencher le workflow également lors de l'ouverture d'une **Pull Request**.
- Ajouter un job supplémentaire au pipeline ci.yml pour exécuter des tests unitaires en parallèle de la construction Docker.
- Ajouter une condition dans ci.yml pour n'exécuter l'étape de push de l'image Docker que si les tests sont passés avec succès.
- Ajouter une étape dans ci.yml qui envoie une **notification** par email si un test échoue en utilisant une action GitHub spécifique (par exemple actions/send-mail).