TD Gestion de projet - 03

1. Intégration continue

Introduction

Ce document décrit les étapes nécessaires pour configurer un pipeline CI/CD (Continuous Integration/Continuous Deployment) en utilisant GitLab CI/CD. Le pipeline est composé de plusieurs étapes : build, test, documentation, et déploiement. Chaque étape est définie dans un job spécifique, et des dépendances sont établies entre les jobs pour assurer un flux de travail cohérent.

Objectif

1. Comprendre comment installer et configurer les dépendances d'un projet Python en utilisant Poetry.
2. Apprendre à exécuter des tests unitaires de manière automatisée pour s'assurer que le code fonctionne correctement.
3. Découvrir comment générer automatiquement la documentation d'un projet en utilisant Sphinx et Make.
4. Comprendre les bases de la construction d'une image Docker pour le déploiement d'une application.

Construction du pipeline

1. Créer un fichier .gitlab-ci.yml à la racine du projet
2. Configuration générale

- Définir les différentes étapes du pipeline
    - build
    - test
    - doc
    - deploy
- Mettre en cache les environnements virtuels
    - .venv

3. Job de Build

Le job de build devra posséder les informations suivantes:

- image
    - python:3.11-alpine
- stage:

    - build
- variables:
    - POETRY_VIRTUALENVS_PATH: $CI_PROJECT_DIR/.venv
- ajouter une initialisation de l'image pour pouvoir build des projets poetry

Installer un paquet sur une distribution alpine se fait en utilisant apk par exemple:

```
apk add git
```

- Les paquets suivants doivent être disponibles dans l'image:
    - git
    - pipx
- Installer poetry à l'aide de pipx (pipx install poetry)
    - Il faudra aussi ajouter pipx au PATH de l'image ainsi que de configurer poetry pour inclure le virtualenv dans le projet

```
export PATH="/root/.local/bin:$PATH"
poetry config virtualenvs.in-project true
```

- Ajouter le script permettant d'installer un projet poetry

4. Job de Test

Configurer un job de test qui utilisera les informations suivantes:

- image
    - python:3.11-alpine
- stage
    - test
- variables:
    - POETRY_VIRTUALENVS_PATH: $CI_PROJECT_DIR/.venv
- Les paquets suivants doivent être disponibles dans l'image:
    - git
    - pipx
- Installer poetry à l'aide de pipx (pipx install poetry)
    - Il faudra aussi ajouter pipx au PATH de l'image ainsi que de configurer poetry pour inclure le virtualenv dans le projet
- Activer le virtual env mit en cache à l'étape de build

```
source $CI_PROJECT_DIR/.venv/bin/activate
```

- Le job devra avoir une dépendence au job de build

```
 dependencies:
   - build-job
```

- Le job disposera d'un artefact de build
    - Le script devra produire cet artefact sous forme de rapport xml d'exécution des tests

```
poetry run pytest --junitxml=report.xml
```

- Et ce rapport sera toujours sauvegardé.

```
artifacts:
   when: always
   reports:
     junit: report.xml  # Sauvegarder le rapport JUnit
   expire_in: 1 day
```

5. Job de Documentation

Configurer un nouveau job doc-job qui publiera la documentation:

- stage:
  - doc
- dépendance
  - build-job
- paquet apk
  - git
  - pipx
  - make
- Configurer aussi poetry pour qu'il fonctionne dans ce paquet
- Utiliser le script suivant ainsi que le déplacement des fichier pour que le projet puisse être publié sur gitlab pages

```
 script:
   - cd docs                # Changer de répertoire vers le dossier
de documentation
   - /usr/bin/make html     # Générer la documentation HTML
   - /bin/mv _build/html ../public  # Déplacer la documentation
générée vers le répertoire public
 artifacts:
   paths:
     - public               # Sauvegarder le répertoire public
   expire_in: 1 day
```

- Ce job sera limité à la branche main

```
 only:
   - main                   # Exécuter ce job uniquement sur la
branche principale
```

- On indique également que l'on publiera cette documentation sur gitlab pages

```
pages: true
```

6. Job de Déploiement

Effectuer un job de déploiement qui n'aura lieu que lorsque l'on aura créé un tag dans notre projet.

- stage:
  - deploy

- dependance
    - test-job
- image

```
   name: gcr.io/kaniko-project/executor:v1.23.2-debug  # Image
Docker utilisée
   entrypoint: [""]        # Entrypoint vide pour exécuter des
commandes personnalisées
```

- script

```
script:
   - /kaniko/executor
     --context "${CI_PROJECT_DIR}"
     --dockerfile "${CI_PROJECT_DIR}/Dockerfile"
     --no-push
```

- règles

```
 rules:
   - if: $CI_COMMIT_TAG     # Exécuter ce job uniquement si un tag
de commit est présent
```

Dans le cadre d'un déploiement d'une application dans un vrai environnement, l'image docker précédemment créée serait poussée dans un registry docker, et une autre étape de déploiement remplacerait et démarrerait le container docker sur le serveur.