UNIVERSITE DE YAOUNDE I

****

ECOLE NATIONALE SUPERIEURE POLYTECHNIQUE DE YAOUNDE

****

DEPARTEMENT DE GENIE INFORMATQUE

****

ARTS ET HUMANITES NUMERIQUES

UNIVERSITY OF YAOUNDE I

****

NATIONAL ADVANCED SCHOOL OF ENGENEERING OF YAOUNDE

****

DEPARTMENT OF COMPUTER SCIENCE

****

ARTS AND DIGITAL HUMANITIES

OS ET VIRTUALISATION

RAPPORT PROJET : PIPELINE CI/CD

NOMS DES EXPOSANTS : FILIERE CYBERSECURITE ET INVESTIGATION NUMERIQUE ET MANAGEMENT DES SYSTEMES D'INFORMATION 5

NOMS ET PRENOMS	MATRICULE
ZOO ZAME NELLY JESSICA	20P039
TOMI OLAMA GABRIELLE SOLANGE	21P643
MINKOULOU ABE ALEXANDRE PATRICK	21P598
DJEUMI NOUBET STEVE THIERRY	20P023

Coordonnateur : M. DJOB MVONDO THYSTERE BARBE

Année académique : 2025-2026

SOMMAIRE

INTRODUCTION

Dans le développement logiciel moderne, la rapidité de livraison et la qualité du code sont devenues des exigences majeures. Les approches traditionnelles, reposant sur des tests manuels et des déploiements ponctuels, sont aujourd'hui insuffisantes face à la complexité croissante des systèmes informatiques.

Le CI/CD (Continuous Integration / Continuous Deployment) est une approche DevOps qui vise à automatiser l'intégration du code, les tests et le déploiement.
Ce projet s'inscrit dans cette dynamique et a pour objectif de concevoir une chaîne CI/CD autour d'une petite API développée en Python.

L'objectif principal est de permettre à toute modification du code source de déclencher automatiquement :

- L'installation de l'environnement,
- L'exécution des tests,
- Et la préparation au déploiement.

I- PRÉSENTATION DES OUTILS

Les outils utilisés sont :

- **Python** : langage de programmation principal.
- **Flask** : micro-framework pour le développement de l'API.
- **Pytest** : framework de tests unitaires.
- **Git** : gestion de versions.
- **GitHub** : hébergement du dépôt.
- **GitHub Actions** : automatisation CI.
- **Docker** : conteneurisation de l'application.
- **Windows** : système d'exploitation de développement.

II- MÉTHODOLOGIE GÉNÉRALE

La démarche suivie repose sur les étapes suivantes :

1. Mise en place de l'environnement de travail
2. Création d'une API simple
3. Implémentation de tests automatisés
4. Mise en place d'un pipeline CI
5. Gestion des erreurs d'environnement
6. Conteneurisation de l'application
7. Validation du processus CI/CD

Chaque étape a été validée par des tests locaux puis intégrée au pipeline.

III- MISE EN ŒUVRE

***Étape 1* : Installation et vérification des outils**

- Allez sur le site de chaque outils pour télécharger :
- Python : https://www.python.org/
- Git : https://git-scm.com/ (Nous avons utilisé celui de notre camarade, car ayany déjà un compte pour le cas d'espèce)
- Vscode : https://code.visualstudio.com/
- Docker Desktop : https://www.docker.com/products/docker-desktop/

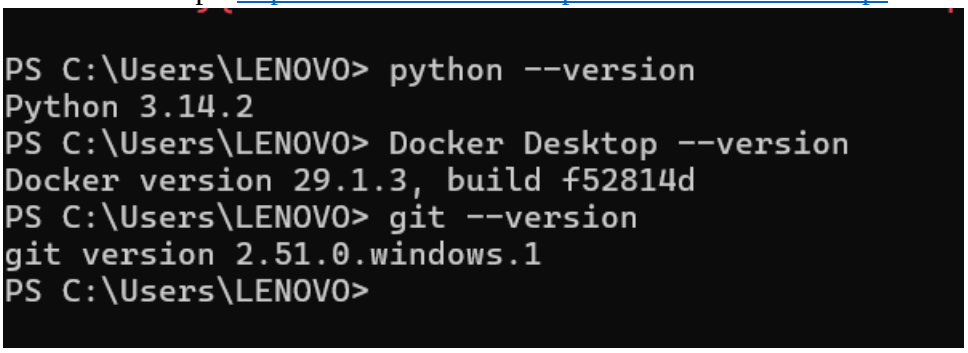


Figure 1: Outils utilisés

<u>Étape 2</u> : Préparation du dossier de travail

Sur le Bureau :

- Créer un dossier : ci-cd-projet

Figure 2: Dossier ci-cd-ptojet

- Ouvrir VS Code
- File → Open Folder → ci-cd-projet

<u>Étape 3</u> : créer le dépôt pour notre projet CI/CD

- Sur GitHub (site web)

1. Connectons-nous au compte GitHub
2. En haut à droite, cliquons sur ✚ → **New repository**
3. Remplissons :

- **Repository name** : **ci-cd-api**
- Description (facultatif) : Projet CI/CD mini API
- Public

4. Cliquons sur **Create repository**

Nous arrivons sur une page avec des commandes.

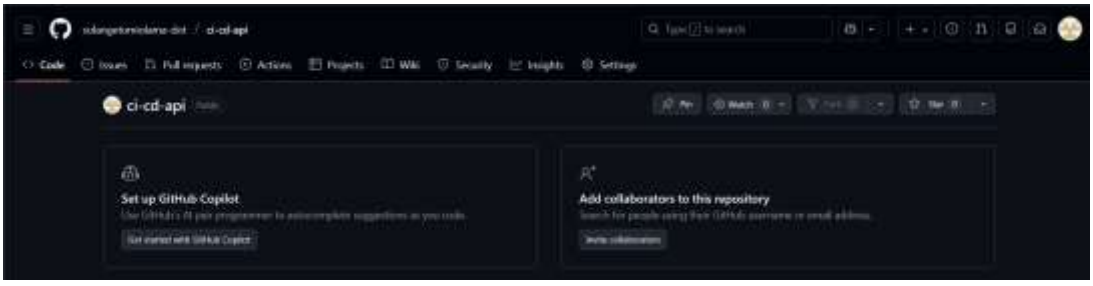


Figure 3:dépôt pour notre projet CI/CD

❖ Laisser cette page ouverte.

❖ Sur notre ordinateur (PowerShell)

a) Créons le dossier du projet (autre méthode si on ne veut pas le faire manuellement)

❖ Par exemple sur le Bureau :

cd Desktop
mkdir ci-cd-api
cd ci-cd-api

b) Initialisons Git

git init

c) Lier notre dossier à GitHub

❖ Sur la page GitHub, copions la ligne qui ressemble à :

git remote add origin https://github.com/solangetomiolama/ci-cd-projet.git

❖ Collons-la dans PowerShell et validons.

Puis :

git branch -M main

<u>Étape 4</u> : Test rapide (important)

Dans PowerShell, tapons :

git status

❖ Nous devons voir un message qui parle de la branche **main**.

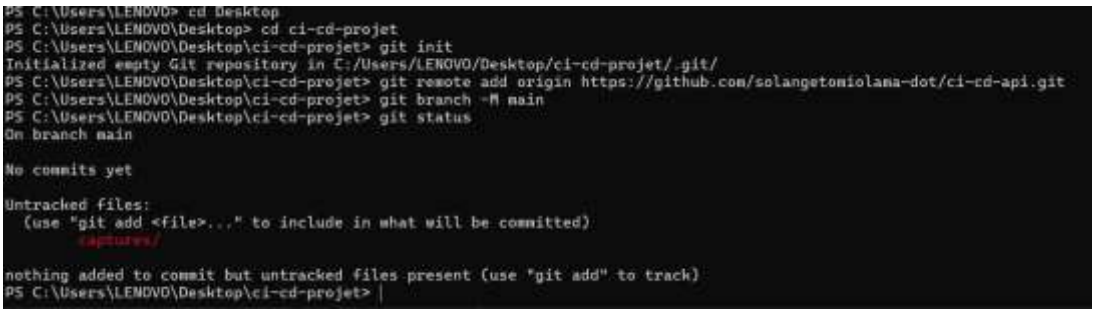


Figure 4:Liaison de notre dossier à GitHub

À ce stade, nous devons avoir :

❖ Un compte GitHub accessible
❖ Un dépôt ci-cd-api créé
❖ Un dossier ci-cd-api sur notre PC
❖ Git initialisé dedans

<u>Étape 5</u> : Créer la mini API

- On va créer une **petite API en Python avec Flask**

NB *: **Flask** est un **micro-framework Python** qui sert à créer des **applications web** et surtout des **API**.*

- ❖ Création de la mini API :

Nous sommes déjà dans ton dossier *ci-cd-projet*.

- Créer un environnement virtuel

- ❖ Dans PowerShell :

python -m venv venv
venv\Scripts\activate

```
PS C:\Users\LENOVO\Desktop\ci-cd-projet> Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned
PS C:\Users\LENOVO\Desktop\ci-cd-projet> venv\Scripts\activate
(venv) PS C:\Users\LENOVO\Desktop\ci-cd-projet> pip install flask pytest
```

Figure 5: Création d' un environnement virtuel

- Ça crée un environnement isolé pour le projet. Nous devons voir ***(venv)*** au début de la ligne.

NB : Il peut y avoir une erreur lors de la création, cette erreur est **normale sous Windows** : PowerShell bloque l'exécution des scripts, donc il empêche activate.ps1 de s'exécuter.

Figure 6: problème survenu lors de la création de l'environnement virtuel

- ❖ Pour le résoudre il faut taper la commande : ***Set-ExecutionPolicy -Scope CurrentUser -ExecutionPolicy RemoteSigned***
- ❖ Installation des bibliothèques : ***pip install flask pytest***
- Flask : pour créer l'API
- Pytest : pour les tests automatiques (CI)

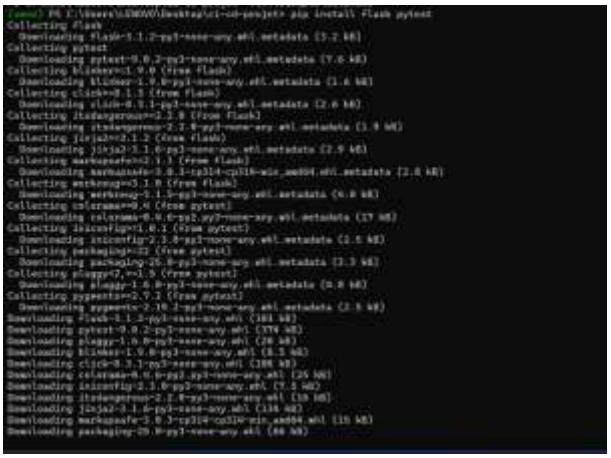


Figure 7: Installation des bibliothèques

- ❖ Création de l'API

❖ Dans ton dossier ***ci-cd-projet***, créons un fichier ***app.py***

- Mettons-y ce bout de code :

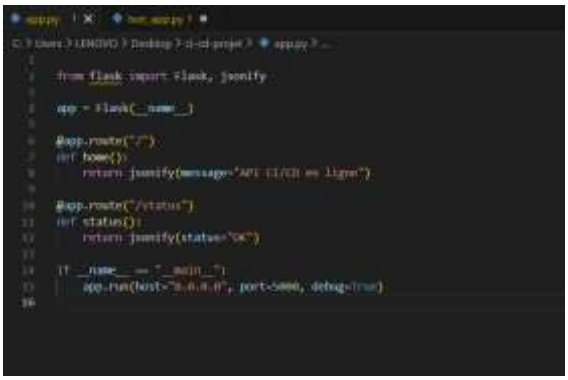


Figure 8: code du fichier app.py

- Explication simple du code app.py :

- from flask import Flask, jsonify : On importe Flask et l'outil pour renvoyer du JSON.
- app = Flask(__name__) : On crée l'application Flask.
- @app.route("/") : On définit une route. Quand on va sur /, Flask exécute la fonction home() et renvoie un message JSON.
- @app.route("/status") : Deuxième route. Elle permet de vérifier que l'API fonctionne.
- if __name__ == "__main__": Le serveur se lance seulement si on exécute ce fichier directement.
- app.run(host="0.0.0.0", port=5000, debug=True) : On démarre le serveur sur le port 5000, accessible sur le réseau, avec le mode debug activé.

Ce fichier crée une petite API avec Flask. On définit deux routes : une pour l'accueil et une pour le statut. Puis on lance le serveur pour rendre l'API accessible.

- *Que fait notre api ?*

*Notre API est une **petite application web Flask** qui fournit **deux réponses simples** :*

1. *Route Accueil /*

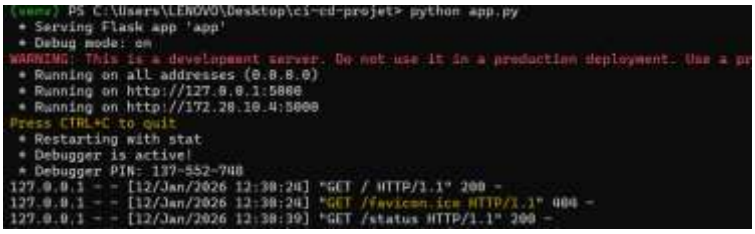
- *Quand on accède à :*
- *http://localhost:5000/, L'API renvoie : {"message": "API CI/CD en ligne"}*
- *Elle sert à montrer que l'API est accessible.*

2. *Route Statut /status*

- *Quand on accède à :*
- *http://localhost:5000/status, L'API renvoie : {"status": "OK"}*
- *Elle sert à vérifier que l'API fonctionne correctement.*

Notre API Flask expose deux routes : une pour l'accueil et une pour le statut. Elles renvoient des messages JSON pour confirmer que l'API est en ligne et opérationnelle.

❖ Lançons l'API : ***python app.py***



- Puis ouvre ton navigateur et teste :

http://localhost:5000 Si nous voyons le message, parfait

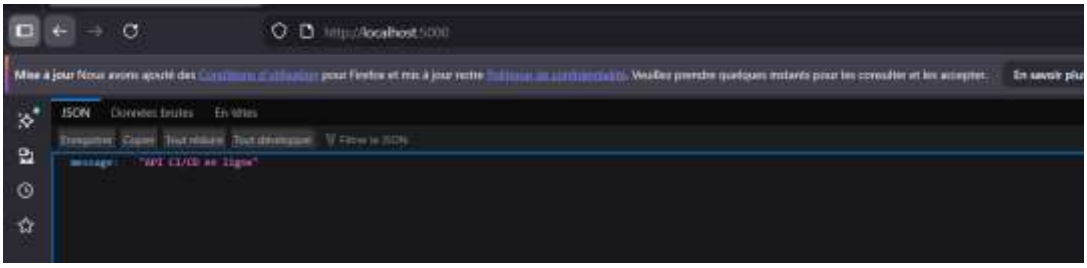


Figure 9: http://localhost:5000:

http://localhost:5000/status

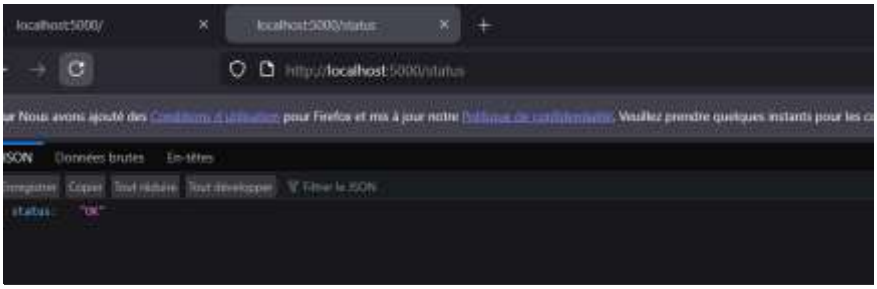


Figure 10: http://localhost:5000:

- Sauvegarder sur GitHub

- ***git add***
- ***git commit -m "Ajout de la mini API Flask"***
- ***git push origin main***

Nb : pour que l'API s'ouvre, il faut la laisser tourner dans powershell et pour arrêter, faire Ctrl+c

- ***Sauvegarde des dépendances***

```
(venv) PS C:\Users\LENOVO\Desktop\ci-cd-projet> pip freeze > requirements.txt
```

Figure 11: Sauvegarde des dépendances

Étape 6 : Ajout des tests automatiques (CI)

Le but : quand nous pushons (*envoyer le code de l'ordinateur sur GitHub*) sur GitHub, des **tests se lancent automatiquement**.

❖ Créons le fichier de tests

- Dans le dossier ci-cd-projet, créons un fichier test_app.py :

- Installation des dépendances

```
(venv) PS C:\Users\LENOVO\Desktop\ci-cd-projet> pip install flask pytest
Collecting flask
  Downloading flask-3.1.2-py3-none-any.whl.metadata (3.2 kB)
Collecting pytest
  Downloading pytest-9.0.2-py3-none-any.whl.metadata (7.6 kB)
Collecting blinker>=1.9.0 (from flask)
  Downloading blinker-1.9.0-py3-none-any.whl.metadata (1.6 kB)
Collecting click>=8.1.3 (from flask)
  Downloading click-8.3.1-py3-none-any.whl.metadata (2.6 kB)
Collecting itsdangerous>=2.2.0 (from flask)
  Downloading itsdangerous-2.2.0-py3-none-any.whl.metadata (1.9 kB)
Collecting jinja2>=3.1.2 (from flask)
  Downloading jinja2-3.1.6-py3-none-any.whl.metadata (2.9 kB)
Collecting markupsafe>=2.1.1 (from flask)
  Downloading markupsafe-3.0.3-cp314-cp314-win_amd64.whl.metadata (2.8 kB)
Collecting werkzeug>=3.1.0 (from flask)
  Downloading werkzeug-3.1.5-py3-none-any.whl.metadata (4.0 kB)
Collecting colorama>=0.4 (from pytest)
  Downloading colorama-0.4.6-py2.py3-none-any.whl.metadata (17 kB)
Collecting iniconfig>=1.0.1 (from pytest)
  Downloading iniconfig-2.3.0-py3-none-any.whl.metadata (2.5 kB)
Collecting packaging>=22 (from pytest)
  Downloading packaging-25.0-py3-none-any.whl.metadata (3.3 kB)
Collecting pluggy<2,>=1.5 (from pytest)
  Downloading pluggy-1.6.0-py3-none-any.whl.metadata (4.8 kB)
Collecting pygments>=2.7.2 (from pytest)
  Downloading pygments-2.19.2-py3-none-any.whl.metadata (2.5 kB)
Downloading flask-3.1.2-py3-none-any.whl (103 kB)
Downloading pytest-9.0.2-py3-none-any.whl (374 kB)
Downloading pluggy-1.6.0-py3-none-any.whl (20 kB)
Downloading blinker-1.9.0-py3-none-any.whl (8.5 kB)
Downloading click-8.3.1-py3-none-any.whl (108 kB)
Downloading colorama-0.4.6-py2.py3-none-any.whl (25 kB)
Downloading iniconfig-2.3.0-py3-none-any.whl (7.5 kB)
Downloading itsdangerous-2.2.0-py3-none-any.whl (16 kB)
Downloading jinja2-3.1.6-py3-none-any.whl (134 kB)
Downloading markupsafe-3.0.3-cp314-cp314-win_amd64.whl (15 kB)
Downloading packaging-25.0-py3-none-any.whl (66 kB)
```

- Ajoutons le code :

```python
import pytest
from app import app

@pytest.fixture
def client():
    """Crée un client de test pour l'application"""
    with app.test_client() as client:
        yield client

def test_home_route(client):
    """Teste la route principale /"""
    response = client.get('/')

    # Vérifie que la réponse est 200 OK
    assert response.status_code == 200

    # Vérifie le contenu JSON
    data = response.get_json()
    assert data['message'] == "API CI/CD en ligne"

def test_status_route(client):
    """Teste la route /status"""
    response = client.get('/status')

    # Vérifie que la réponse est 200 OK
    assert response.status_code == 200

    # Vérifie le contenu JSON
    data = response.get_json()
    assert data['status'] == "OK"

def test_404_route(client):
    """Teste qu'une route inexistante renvoie 404"""
    response = client.get('/inexistante')
    assert response.status_code == 404
```

Figure 12: test_app.py

❖ Tester manuellement les tests

- Dans PowerShell (toujours avec `(venv)` actif) : ***pytest test_app.py –v***

l'on devrait voir :

- test_app.py::test_home_route PASSED
- test_app.py::test_status_route PASSED
- test_app.py::test_404_route PASSED

Figure 13: Test manuel test_app.py

❖ Créer le fichier de configuration GitHub Actions

- Créons un dossier et un fichier :
- Crée le dossier .github/workflows : ***mkdir -p .github\workflows***
- Crée le fichier de pipeline CI

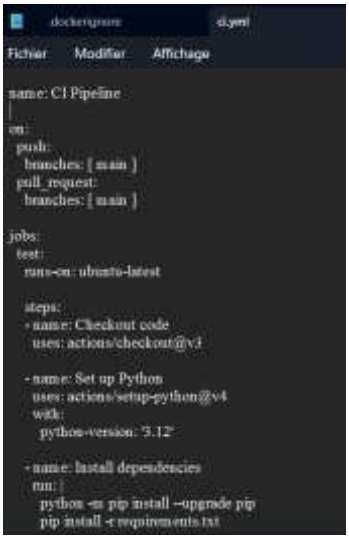


Figure 14: fichier de pipeline CI

- Ajouter un fichier .gitignore
- Ouvrir le Bloc-notes (Notepad)
- Mettre ce contenu :

- Enregistrer sous
- **Nom :** .gitignore (attention au point devant !)
- **Emplacement :** C:\Users\LENOVO\Desktop\ci-cd-projet

- **Type :** Tous les fichiers
- **Encoder en :** UTF-8
- **Enregistrer**

❖ Commit et push

- Vérifier les changements
- git status
- Ajouter tout
- git add
- Commit
- git commit -m "Étape 3 : ajout des tests automatiques et configuration CI"
- Push sur GitHub
- git push origin main

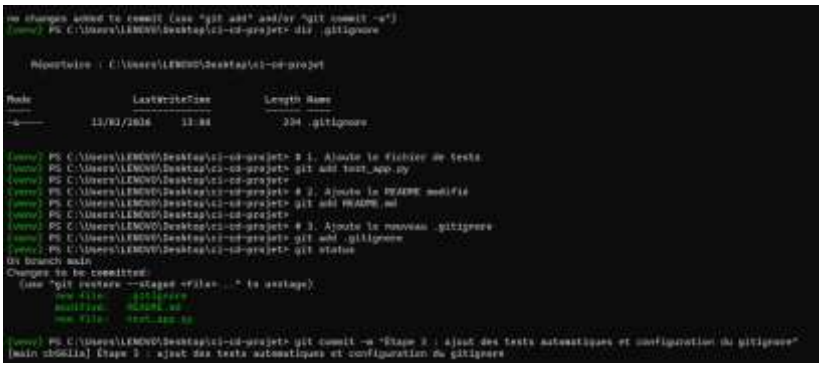


Figure 15: Commit et push

- Envoie sur github (push)
- git push origin main

Figure 16: Envoie sur github (push)

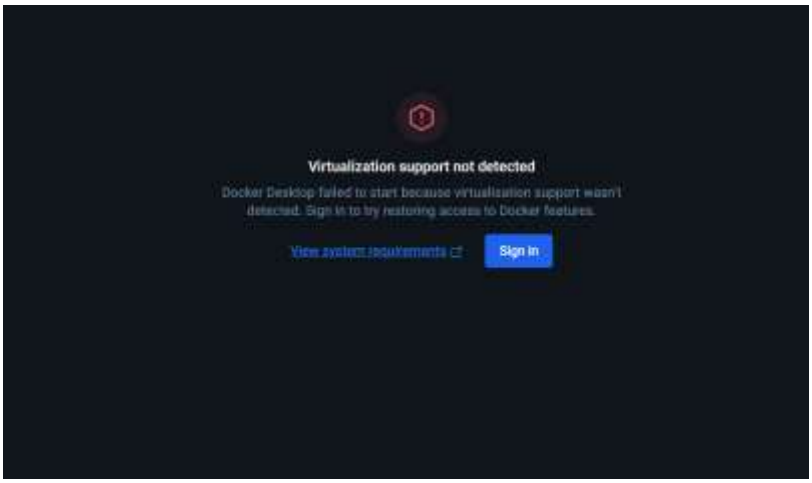
Étape7 : Containerisation avec docker

Objectif : *Empaqueter l'application dans un conteneur Docker pour garantir sa portabilité et sa reproductibilité entre différents environnements.*

- ***Docker*** = une boîte qui contient toute ton application
- Pourquoi Docker ?

- Ton application marche partout (PC, cloud, autre ordi)
- Même environnement en dev et en prod
- Pas de "ça marche chez moi mais pas chez toi"

NB : *La configuration Docker a été réalisée avec la création du Dockerfile et l'intégration au pipeline CI. L'activation de la virtualisation dans le BIOS, nécessaire pour l'exécution locale des conteneurs, n'a pas été effectuée afin de se concentrer sur les aspects essentiels du CI/CD. Le pipeline GitHub Actions assure pleinement la construction et le test des images Docker, validant ainsi la configuration.*



❖ Créer le Dockerfile

- Ouvrir le Bloc-notes (Windows + R, taper notepad, Enter)
- Mettre ce contenu

```
# Utilise une image Python officielle légère
FROM python:3.9-slim

# Définit le répertoire de travail dans le conteneur
WORKDIR /app

# Copie le fichier de dépendances
COPY requirements.txt .

# Installe les dépendances
RUN pip install --no-cache-dir -r requirements.txt

# Copie le reste du code
COPY . .

# Expose le port utilisé par Flask
EXPOSE 5000

# Commande pour lancer l'application
CMD ["python", "app.py"]
```

Figure 17: Dockerfile

- Fichier → Enregistrer sous
- Nom : Dockerfile (attention, PAS .txt)
- Type : Tous les fichiers
- Emplacement : C:\Users\LENOVO\Desktop\ci-cd-projet
- Enregistrer

❖ Vérifier que c'est bon :

- Dans PowerShell :
- # Vérifie que le fichier existe
- *dir Dockerfile*
- # Voir le contenu
- *cat Dockerfile*

L’on devrait voir notre fichier Dockerfile.

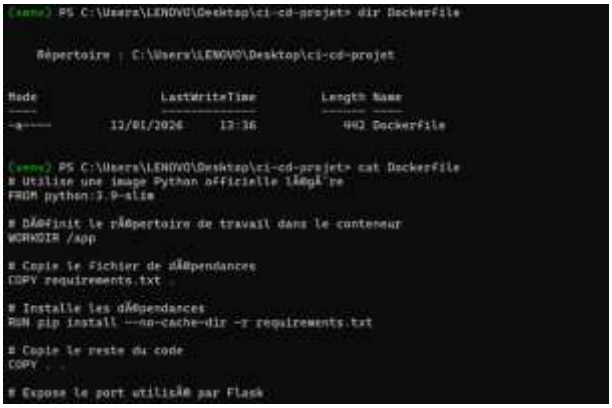


Figure 18: vérification du fichier Dockerfile

- ❖ Créer le fichier .dockerignore : *C'est la liste des choses à ne PAS mettre dans la boîte.*

- Comment créer :

- Même méthode que Dockerfile
- Nom : .dockerignore

- ❖ Mettre à jour le pipeline ci :

- Maintenant, on va dire à GitHub Actions de tester Docker pour nous :

- Ouvre ton fichier ci.yml
- Dans PowerShell
- *cd C:\Users\LENOVO\Desktop\ci-cd-projet\.github\workflows*
- *notepad ci.yml*

- Remplacer tout par CE code :

Figure 19: nouveau code ci.yml

Figure 20: nouveau code ci.yml

- Sauvegarder et fermer

❖ Tester le pipeline

- Sauvegarder sur GitHub

- Retourner au dossier principal : *cd C:\Users\LENOVO\Desktop\ci-cd-projet*
- Ajouter les fichiers Docker
- *git add Dockerfile .dockerignore .github/workflows/ci.yml*
- Commit *: git commit -m "Étape 4 : Ajout Dockerfile et configuration Docker dans le pipeline"*
- Push *: git push origin main*

- Vérifier sur GitHub

  - Ouvrir *https://github.com*
  - Aller dans le dépôt `***ci-cd-projet***`
  - Cliquer sur l'onglet "***Actions***"
  - Attendre 1-2 minutes

- L'on devrait voir :

  - test (Python tests) - 30s
  - build-docker (Docker build) - 2.75s ← **C'EST ÇA LE SUCCÈS !**

❖ Ce que github a fait pour nous :

- Quand nous avons fait `git push`, GitHub a fait ***tout ça automatiquement*** :

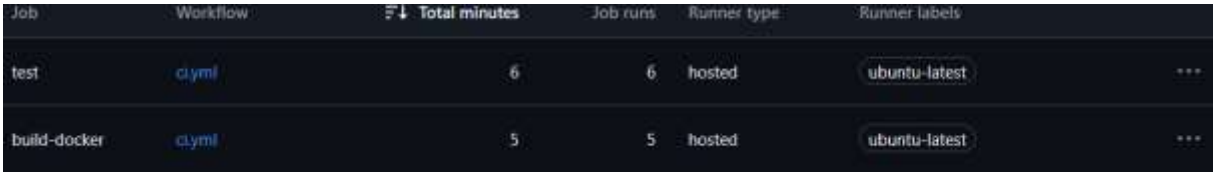
  - Récupère notre code
  - Lance une machine Ubuntu (avec Docker déjà installé)
  - Lit notre Dockerfile
  - Construit l'image "ci-cd-api"

- Lance un conteneur avec cette image
- Teste que l'API fonctionne dans le conteneur
- Affiche ✓ si tout est bon

*Le plus important : GitHub a **validé** que* notre *Dockerfile fonctionne, même si nous ne pouvons pas le tester localement.*

❖ Preuves que docker fonctionne :

- Logs de build sur GitHub



Étape8 : Déploiement continu (cd) - déploiement sur railway

On va déployer ton API automatiquement sur Railway.

- Pourquoi Railway ?

  - Gratuit pour les petits projets
  - Facile à configurer
  - Intégration GitHub automatique
  - Pas besoin de carte bancaire
  - URL publique immédiate

❖ Créer un compte Railway

- Ouvrir Railway : *https://railway.app*
- S'inscrire
- Cliquer sur "Start a New Project
- Choisir "Deploy from GitHub repo"
- Se connecter avec notre compte GitHub
- Autoriser l'accès à Railway

  - Sélectionner notre dépôt :

- Chercher ***ci-cd-projet***
- Cliquer sur "*Deploy*"

❖ Configuration automatique :

- Railway détectera automatiquement :

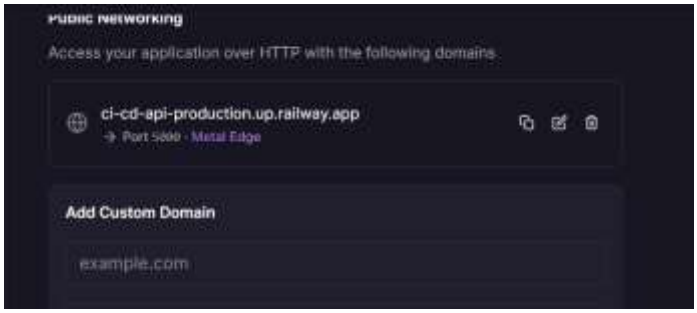
  - ✓ Dockerfile → utilisera Docker pour le déploiement
  - ✓ Port 5000 → exposera notre API

- Attendre 2-3 minutes que le déploiement se termine.

Figure 21: Déployement API

❖ Trouver notre url publique

- Dans Railway, allons sur ton projet
- Cliquons sur l'onglet "Settings"
- Cherchons "Public URL" ou "Domain"
- Copions l'URL (ex: `https://ci-cd-projet.up.railway.app`) : *https://ci-cd-api-production.up.railway.app/*



❖ Testons l'URL :

- Ouvrons *https://ci-cd-api-production.up.railway.app/* → ***Doit afficher*** `{"message":"API CI/CD en ligne"}`

Figure 22: test https://ci-cd-api-production.up.railway.app/

- Ouvrons *https://ci-cd-api-production.up.railway.app/status* → **Doit afficher** `{"status":"OK"}`

Figure 23: https://ci-cd-api-production.up.railway.app/status

CONCLUSION DU RAPPORT

CONCLUSION GÉNÉRALE

Ce projet avait pour objectif de mettre en pratique les principes du DevOps à travers la conception et la mise en œuvre d'un pipeline complet d'Intégration et Déploiement Continus (CI/CD). L'ambition était de démontrer comment automatiser le cycle de vie d'une application web, depuis le développement local jusqu'à sa mise en production accessible sur Internet, en garantissant qualité et reproductibilité à chaque étape.

L'objectif a été pleinement atteint avec la mise en ligne réussie d'une API Flask fonctionnelle à l'adresse ***https://ci-cd-api-production.up.railway.app.*** , validant ainsi l'ensemble de la chaîne d'outils et des processus mis en place. L'intégralité du code source, de l'historique des commits et de la configuration du pipeline est disponible et consultable publiquement sur le dépôt GitHub du projet : ***https://github.com/solangetomiolama-dot/ci-cd-api.***

RAPPEL DES ÉTAPES CLÉS ET RÉALISATIONS

1) Mise en place de l'environnement et initialisation

- Configuration d'un environnement de développement sous Windows avec Python, Git et Docker.
- Création et gestion d'un dépôt Git versionné sur GitHub (***https://github.com/solangetomiolama-dot/ci-cd-api*)**, établissant les bases de la collaboration et du suivi des modifications.

2) Développement de l'application cœur

- Conception d'une API web minimale mais robuste avec le framework Flask, offrant deux endpoints (`/` et `/status`) au format JSON.
- Gestion des dépendances via un environnement virtuel Python et un fichier `requirements.txt`, assurant l'isolation et la reproductibilité.

3) Assurance qualité par l'automatisation des tests

- Implémentation de tests unitaires ciblés avec **pytest** pour vérifier automatiquement le comportement de l'API.
- Configuration d'un pipeline d'intégration continue avec **GitHub Actions,** déclenché à chaque modification du code pour exécuter ces tests et valider la stabilité de l'application. L'historique des exécutions est visible dans l'onglet **Actions** du dépôt

4) Containerisation pour la portabilité

- Création d'un Dockerfile définissant l'environnement d'exécution complet de l'application.
- Intégration de la construction et du test de l'image Docker au sein du pipeline CI, garantissant que l'application peut s'exécuter de manière identique et isolée dans tout environnement compatible Docker.

5) Déploiement continu en production

- Connexion du dépôt GitHub à la plateforme cloud **Railway**.
- Obtention d'une URL publique https://ci-cd-api-production.up.railway.app permettant un accès global à l'API déployée.
- Validation du processus de déploiement automatique, où toute validation du pipeline CI entraîne une mise à jour en production sans intervention manuelle.

APPRENTISSAGES PRINCIPAUX ET DIFFICULTÉS SURNOMONTÉES

Ce projet, bien que structuré, a été ponctué de défis techniques caractéristiques du développement moderne, dont la résolution a été formatrice :

- Gestion des environnements et des dépendances : La confrontation à des incompatibilités de versions Python a souligné l'importance cruciale de figer et de tester les environnements, un pilier de la reproductibilité.
- Précision dans la configuration: Des erreurs de syntaxe dans des fichiers de configuration (YAML pour GitHub Actions, Dockerfile) ont démontré que la rigueur et l'attention aux détails sont essentielles dans les pratiques DevOps, où l'infrastructure est définie par du code.
- Abstraction et utilisation des services cloud: La limitation matérielle locale (virtualisation désactivée) a été contournée en utilisant intelligemment les services cloud (GitHub Actions, Railway), illustrant la puissance et la flexibilité des outils DevOps contemporains.
- Résilience et débogage: Chaque obstacle, depuis les problèmes d'exécution de scripts PowerShell jusqu'à la configuration des domaines sur Railway, a renforcé la capacité à lire les messages d'erreur, rechercher des solutions et adapter la démarche.

BILAN ET PERSPECTIVES

Le projet a abouti à la création d'une chaîne de déploiement entièrement automatisée et opérationnelle. Le pipeline CI/CD constitue désormais un système fiable qui garantit que tout nouveau code est testé, empaqueté et déployé en production dès qu'il est validé sur la branche principale. L'ensemble du processus est transparent et consultable, du code source sur *https://github.com/solangetomiolama-dot/ci-cd-api*. à l'application en ligne sur Railway https://ci-cd-api-production.up.railway.app

Cette réalisation concrétise plusieurs principes fondamentaux du DevOps : l'automatisation pour gagner en efficacité et en fiabilité, l'infrastructure as code pour la reproductibilité, et l'intégration continue pour une détection précoce des anomalies.

- ***Perspectives d'évolution*** : Ce socle fonctionnel ouvre la voie à de nombreuses améliorations, telles que l'ajout de tests d'intégration, la mise en place d'environnements de pré-production, l'intégration d'outils de qualité de code, l'ajout d'une base de données, ou la mise en place de stratégies de déploiement plus avancées (blue-green, canary).

.

RÉFÉRENCES ET LIENS DU PROJET :

- Dépôt GitHub (code source et historique) : *https://github.com/solangetomiolama-dot/ci-cd-api*
- Application en production :** https://ci-cd-api-production.up.railway.app
- Pipeline CI/CD :** [*https://github.com/solangetomiolama-dot/ci-cd-api/actions*