



TP 2 : CI/CD avec GitHub Actions

Prérequis : TP 1 terminé (backend et frontend avec tests)



Objectifs du TP

Objectif principal : Automatiser les tests avec GitHub Actions

À la fin de ce TP, vous aurez :

1. Créé un **workflow backend** qui teste automatiquement votre code Python
2. Créé un **workflow frontend** qui teste et build votre code TypeScript
3. Compris comment **déboguer** un workflow qui échoue
4. Protégé votre branche **main** pour empêcher les bugs d'arriver en production
5. Créé des **workflows réutilisables** et des **pipelines CI**
6. Séparé les **tests rapides** (unitaires) des **tests lents** (E2E)
7. Créé une **chaîne de jobs** frontend optimisée
8. Ajouté des **badges de status** à votre README



Qu'est-ce que CI/CD ?

CI (Continuous Integration) :

- Intégration Continue
- À chaque push, les tests s'exécutent automatiquement
- Détecte les bugs immédiatement

CD (Continuous Deployment) :

- Déploiement Continu (TP 3)
- Si les tests passent, déploiement automatique

GitHub Actions :

- Service gratuit de GitHub
- Exécute vos tests sur des serveurs GitHub
- Vérifie chaque commit et pull request



Structure d'un Workflow GitHub Actions

Un workflow GitHub Actions est un fichier **YAML** dans `.github/workflows/`.

Structure de base :

```
name: Mon Workflow          # 1 Nom affiché dans GitHub

on:                          # 2 Quand s'exécute-t-il ?
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:                        # 3 Les tâches à faire
  test:
    runs-on: ubuntu-latest  # 4 Machine virtuelle
    steps:                  # 5 Les étapes
      - uses: actions/checkout@v4
      - run: pytest
```

Concepts clés :

- **name** : Nom du workflow
- **on** : Déclencheurs (push, pull_request, etc.)
- **jobs** : Tâches parallèles
- **steps** : Étapes séquentielles
- **uses** : Action pré-faite
- **run** : Commande shell

Exercice 1 : Workflow Backend

Objectif

Créer un workflow qui teste automatiquement le backend à chaque push.

Instructions

1. Créez la structure des workflows :

```
mkdir -p .github/workflows
```

2. Créez le fichier `.github/workflows/backend.yml`

3. Configurez le workflow avec :

- Nom : "Backend Tests"
- Déclencheurs : push et pull_request sur `main`
- Job nommé "test" qui s'exécute sur `ubuntu-latest`

4. Ajoutez les étapes suivantes (dans l'ordre) :

- Récupérer le code avec `actions/checkout@v4`
- Installer Python 3.11 avec `actions/setup-python@v5`
 - **Activez le cache pip automatique :** `cache: 'pip'`
- Installer UV avec pip : `pip install uv`
- Installer les dépendances : `cd backend && uv sync`
- Lancer les tests : `cd backend && uv run pytest -v --cov`

💡 **Note sur le cache :** En ajoutant `cache: 'pip'`, GitHub Actions met automatiquement en cache les dépendances Python. Vous verrez "Cache restored" lors des exécutions suivantes, ce qui accélère le workflow !

5. Testez localement avant de pousser :

```
cd backend
uv run pytest -v --cov
```

6. Poussez et vérifiez sur GitHub Actions :

```
git add .github/workflows/backend.yml
git commit -m "ci: add backend workflow"
git push origin main
```

✅ Résultat attendu

Dans l'onglet "Actions" sur GitHub, vous devriez voir :

```
✅ Backend Tests
└─ test
    ├─ Checkout code
    ├─ Setup Python
    ├─ Install UV
    ├─ Install dependencies
    └─ Run tests
```

📝 Exercice 2 : Workflow Frontend

Objectif

Créer un workflow qui teste et build le frontend automatiquement.

Instructions

1. Créez le fichier `.github/workflows/frontend.yml`
2. Configurez le workflow similairement au backend :
 - Nom : "Frontend Tests"
 - Mêmes déclencheurs que le backend
3. Ajoutez les étapes suivantes :
 - Récupérer le code
 - Installer Node.js 18 avec `actions/setup-node@v4`

- Activez le cache npm : `cache: 'npm'`
- Spécifiez le chemin : `cache-dependency-path: frontend/package-lock.json`
- Installer les dépendances : `cd frontend && npm ci`
- Lancer les tests : `cd frontend && npm test -- --run`
- Vérifier le build : `cd frontend && npm run build`

4. Testez localement :

```
cd frontend
npm test -- --run
npm run build
```

5. Poussez et vérifiez :

```
git add .github/workflows/frontend.yml
git commit -m "ci: add frontend workflow"
git push origin main
```

✅ Résultat attendu

Vous devriez voir **2 workflows en parallèle** :

- ✅ Backend Tests
- ✅ Frontend Tests

💡 Note importante

`npm ci` vs `npm install` :

- `npm ci` : Installe exactement ce qui est dans `package-lock.json` (déterministe)
- `npm install` : Peut mettre à jour les versions (moins fiable pour CI)

🔪 Exercice 3 : Débuguer un Échec

Objectif

Apprendre à lire les logs et corriger les erreurs de workflow.

Instructions

1. Introduisez volontairement un bug dans `backend/tests/test_api.py` :

```
def test_health_check(client):  
    response = client.get("/health")  
    assert response.status_code == 200  
    assert response.json()["status"] == "BROKEN" # ❌ Faux exprès !
```

2. Poussez le bug :

```
git add backend/tests/test_api.py  
git commit -m "test: intentional failure for learning"  
git push origin main
```

3. Observez l'échec sur GitHub Actions :

- Allez dans "Actions"
- Cliquez sur le workflow ❌ rouge
- Cliquez sur l'étape "Run tests"

4. Analysez les logs :

- Quel test échoue ?
- À quelle ligne ?
- Quelle est l'erreur exacte ?

5. Reproduisez localement :

```
cd backend  
uv run pytest tests/test_api.py::test_health_check -v
```

6. Corrigez le bug :

```
assert response.json()["status"] == "healthy" # ✅ Correct
```

7. Vérifiez localement puis poussez :

```
uv run pytest tests/test_api.py::test_health_check -v
git add backend/tests/test_api.py
git commit -m "fix: correct health check assertion"
git push origin main
```

✅ Leçon apprise

Si ça passe localement, ça passera sur GitHub !



Exercice 4 : Protection de Branches



Objectif

Empêcher les merges sur `main` si les tests échouent. Situation réelle : créer une branche avec un bug, ouvrir une PR, et voir GitHub bloquer le merge !



Partie 1 : Activer la Protection de Branche

1. Sur GitHub, allez dans votre repository → Settings → Branches
2. Cliquez sur "Add branch protection rule"
3. Configurez la règle :
 - Branch name pattern : `main`
 - ✅ Require status checks to pass before merging
 - Cliquez sur "Add required status check"
 - Cherchez et ajoutez : `test` (le nom du job dans vos workflows)
 - ✅ Require branches to be up to date before merging
4. Cliquez sur "Create" en bas de la page

Partie 2 : Tester avec une Branche Qui Casse les Tests

Scénario réaliste : Vous introduisez un bug accidentellement. GitHub doit vous empêcher de merger !

1. Créez une nouvelle branche :

```
git checkout -b feature/test-branch-protection
```

2. Introduisez un bug dans `backend/tests/test_api.py` :

```
def test_health_check(client):  
    response = client.get("/health")  
    assert response.status_code == 200  
    assert response.json()["status"] == "BROKEN" # ❌ Bug volontaire
```

3. Commitez et poussez :

```
git add backend/tests/test_api.py  
git commit -m "test: intentionally break health check"  
git push origin feature/test-branch-protection
```

4. Créez une Pull Request :

- Allez sur GitHub dans votre repository
- Vous verrez un bouton **"Compare & pull request"** → Cliquez dessus
- Titre : "Test branch protection"
- Description : "Testing if broken tests block merge"
- Cliquez sur **"Create pull request"**

5. Observez ce qui se passe :

- ⌚ Les workflows s'exécutent automatiquement
- ❌ Le job `test` échoue (tests backend en erreur)
- 🔒 Le bouton **"Merge pull request"** devient **grisé et inutilisable**
- ⚠️ GitHub affiche : *"Required status check 'test' has not been successful"*

✅ Partie 3 : Corriger et Merger

1. Corrigez le bug (toujours sur la même branche) :

```
assert response.json()["status"] == "healthy" # ✅ Correct
```

2. Commitez et poussez la correction :

```
git add backend/tests/test_api.py
git commit -m "fix: correct health check assertion"
git push origin feature/test-branch-protection
```

3. Observez la PR :

- ✅ Les workflows se relancent **automatiquement**
- ✅ Les tests passent maintenant
- ✅ Le bouton **"Merge pull request"** devient **vert et cliquable**

4. Mergez la PR :

- Cliquez sur **"Merge pull request"**
- Confirmez avec **"Confirm merge"**

5. Nettoyez votre environnement local :

```
git checkout main
git pull origin main
git branch -d feature/test-branch-protection
```

🔍 Ce Que Vous Devriez Voir

Étape 5 - PR bloquée :

```
⚠️ Merging is blocked
❌ Required status check "test" has not been successful

Some checks were not successful
❌ Backend Tests / test – Failed
```

This branch has not been approved

🔒 Merge blocked

Étape 3 (après fix) - PR débloquée :

- ✅ All checks have passed
- ✅ Backend Tests / test – Passed
- ✅ Frontend Tests / test – Passed

This branch has no conflicts with the base branch

🎉 Ready to merge

💡 Points Clés à Comprendre

Q1 : Pourquoi est-ce important ?

- **R** : Empêche les bugs d'arriver en production. Si un développeur casse quelque chose, GitHub le force à corriger **avant** de merger.

Q2 : Est-ce que ça ralentit le développement ?

- **R** : Non ! Au contraire, ça évite de perdre du temps à déboguer en production. *"Fail fast, fix fast"*.

Q3 : Peut-on contourner cette protection ?

- **R** : Oui, les admins du repo peuvent forcer le merge. Mais **c'est une mauvaise pratique** sauf urgence critique.

✅ Critères de Réussite

Vous avez réussi si :

- ✅ Vous avez créé une PR avec des tests qui échouent
- ✅ GitHub a bloqué le merge (bouton grisé)
- ✅ Après correction, le merge est devenu possible
- ✅ La branche a été mergée dans `main`

🎁 BONUS : Protection Avancée

Si vous finissez en avance, ajoutez ces règles supplémentaires :

1. Retournez dans **Settings** → **Branches** → **Edit rule**

2. Activez :

- ☒ **Require a pull request before merging**
 - ☒ **Require approvals : 1**
- ☒ **Do not allow bypassing the above settings**

3. **Testez en créant une autre PR** : Vous ne pourrez plus merger même si les tests passent, il faudra une **approbation** d'un autre développeur !

Exercice 5 : Workflows Réutilisables

Objectif

Créer un pipeline CI global qui orchestre backend et frontend.

Instructions

1. **Rendez vos workflows réutilisables :**

Dans `backend.yml` et `frontend.yml`, ajoutez `workflow_call` aux déclencheurs :

```
on:
  push:
    branches: [main]
  pull_request:
    branches: [main]
  workflow_call: # ✨ Nouveau !
```

2. **Créez `.github/workflows/ci-pipeline.yml` :**

```
name: CI Pipeline

on:
  push:
```

```

    branches: [main]
  pull_request:
    branches: [main]

  jobs:
    backend:
      name: Backend Tests
      uses: ../github/workflows/backend.yml

    frontend:
      name: Frontend Tests
      uses: ../github/workflows/frontend.yml

    summary:
      name: All Tests Passed
      needs: [backend, frontend]
      runs-on: ubuntu-latest
      steps:
        - name: 🎉 Success
          run: echo "🎉 Tous les tests sont passés !"

```

3. Poussez et observez :

```

git add .github/workflows/
git commit -m "ci: add reusable workflows and pipeline"
git push origin main

```

✅ Résultat

Vous verrez maintenant **3 workflows** :

- ✅ Backend Tests
- ✅ Frontend Tests
- ✅ CI Pipeline (résumé global)

Le job `summary` attend que backend **ET** frontend soient terminés avant de s'exécuter.



Exercice 6 : Séparer Tests Unitaires et E2E

Objectif

Exécuter les tests rapides (unitaires) sur toutes les branches, mais les tests lents (E2E) seulement sur `main`.

Instructions

Partie 1 : Marquer les tests E2E

1. Dans `backend/tests/conftest.py`, ajoutez :

```
def pytest_configure(config):
    """Enregistre les markers personnalisés"""
    config.addinvalue_line(
        "markers",
        "e2e: mark test as end-to-end test (slow)"
    )
```

2. Dans `backend/tests/test_api.py`, créez un test E2E :

```
import pytest

@pytest.mark.e2e
def test_complete_task_lifecycle(client):
    """Test E2E : CRUD complet d'une tâche."""
    # Créer
    response = client.post("/tasks", json={
        "title": "Test E2E",
        "description": "Test complet"
    })
    assert response.status_code == 201
    task_id = response.json()["id"]

    # Lire
    response = client.get(f"/tasks/{task_id}")
    assert response.status_code == 200

    # Mettre à jour
```

```

response = client.put(f"/tasks/{task_id}", json={
    "title": "Updated",
    "description": "Modified"
})
assert response.status_code == 200

# Supprimer
response = client.delete(f"/tasks/{task_id}")
assert response.status_code == 204

# Vérifier suppression
response = client.get(f"/tasks/{task_id}")
assert response.status_code == 404

```

3. Testez localement les différentes commandes :

```

cd backend
# Tests unitaires seulement
uv run pytest -v -m "not e2e"

# Tests E2E seulement
uv run pytest -v -m "e2e"

# Tous les tests
uv run pytest -v

```

Partie 2 : Créer le workflow séparé

4. Créez `.github/workflows/backend-split.yml` avec 2 jobs :

- **Job 1 : unit-tests** (toujours)
 - Exécute : `pytest -v -m "not e2e"`
- **Job 2 : e2e-tests** (seulement sur main)
 - Ajoute la condition : `if: github.ref == 'refs/heads/main'`
 - Exécute : `pytest -v -m "e2e"`

5. Testez avec une Pull Request :

```
git checkout -b test/split-tests
echo "# Test" >> README.md
git add .
git commit -m "test: verify E2E don't run on PR"
git push origin test/split-tests
```

✅ Résultat attendu

- **Sur PR** : Seulement "Unit Tests" s'exécute
- **Sur main** : "Unit Tests" **ET** "E2E Tests" s'exécutent



Exercice 7 : Chaîne de Jobs Frontend

Objectif

Créer une chaîne Lint → Test → Build pour optimiser le feedback.

Instructions

1. Créez `.github/workflows/frontend-chain.yml` avec 3 jobs :

Job 1 : lint

- Installe les dépendances
- Exécute : `npm run lint`

Job 2 : test

- Dépend de `lint` avec `needs: lint`
- Installe les dépendances
- Exécute : `npm test -- --run`

Job 3 : build

- Dépend de `test` avec `needs: test`
- Installe les dépendances

- Exécute : `npm run build`
- Upload les artifacts avec `actions/upload-artifact@v4` :

```
- name: 📁 Upload build artifacts
  uses: actions/upload-artifact@v4
  with:
    name: frontend-build
    path: frontend/dist/
```

2. Poussez et observez :

```
git add .github/workflows/frontend-chain.yml
git commit -m "ci: add frontend chain"
git push origin main
```

✅ Avantages

- Si lint échoue → tests et build ne s'exécutent pas
- Feedback plus rapide (lint = 10s vs build = 2min)
- Build artifacts disponibles pour téléchargement

👉 Exercice 8 : Badges de Status

Objectif

Afficher le statut des workflows dans votre README.

Instructions

1. Modifiez `README.md` et ajoutez au début :

```
# TaskFlow API

![Backend Tests](https://github.com/VOTRE_NOM/VOTRE_REPO/workflows/Bac
```



```
![Frontend Tests](https://github.com/VOTRE_NOM/VOTRE_REPO/workflows/Fr
![CI Pipeline](https://github.com/VOTRE_NOM/VOTRE_REPO/workflows/CI%20
```

2. Remplacez :

- `VOTRE_NOM` → Votre username GitHub
- `VOTRE_REPO` → Nom de votre repo

3. Poussez :

```
git add README.md
git commit -m "docs: add CI badges"
git push origin main
```

✓ Résultat

Sur GitHub, vous verrez des badges qui se mettent à jour automatiquement :

✓ Backend Tests ✓ Frontend Tests ✓ CI Pipeline

Récapitulatif








Félicitations ! Vous avez maintenant :

✓ **Exercice 1** : Workflow backend automatisé (avec cache pip automatique) ✓
Exercice 2 : Workflow frontend automatisé (avec cache npm automatique) ✓ **Exercice 3** : Compétences en débogage de workflows ✓ **Exercice 4** : Protection de branches pour empêcher les bugs d'arriver en production ✓ **Exercice 5** : Pipeline CI global avec workflows réutilisables ✓ **Exercice 6** : Séparation tests unitaires / E2E ✓ **Exercice 7** : Chaîne de jobs frontend optimisée ✓ **Exercice 8** : Badges de status dans le README

Compétences Acquises

Vous savez maintenant :

- ✓ Créer et configurer des workflows GitHub Actions

-  Utiliser le cache automatique pour accélérer les builds
-  Déboguer des workflows qui échouent
-  Protéger la branche `main` contre les bugs
-  Créer des Pull Requests et comprendre le processus de review
-  Organiser des pipelines CI complexes
-  Séparer tests rapides et tests lents
-  Afficher le statut de vos workflows avec des badges

Ces compétences sont directement utilisables en entreprise ! 🚀

Temps total estimé : 4-5 heures

Erreurs Fréquentes

`uv: command not found`

Cause : UV n'est pas installé ou pas dans le PATH **Solution :** Vérifiez que vous avez bien `pip install uv` dans votre workflow

Tests qui passent localement mais échouent sur GitHub

Causes possibles :

1. Variable d'environnement manquante
2. Dépendance système manquante
3. Timezone différente
4. Version de Python/Node différente

Débuguer : Reproduisez exactement les mêmes commandes localement avec la même version

"Required status check has not been successful"

Cause : Vous avez activé la protection de branche mais les tests échouent **Solution :** C'est normal ! Corrigez vos tests sur la branche, poussez à nouveau, et le merge se débloquent

❌ Cache qui ne se restaure pas

Cause : Le cache pip/npm automatique ne fonctionne que si les fichiers de dépendances (`requirements.txt` , `package-lock.json` , etc.) n'ont pas changé **Solution :** C'est normal si vous avez modifié vos dépendances. Le cache se reconstruira automatiquement



BONUS 1 : Cache UV Manuel (Pour les Curieux)

Vous voulez comprendre comment fonctionne le cache en profondeur ?

Le cache automatique (`cache: 'pip'`) est pratique, mais vous pouvez aussi créer des caches personnalisés avec `actions/cache@v4` .

Exemple : Cache UV Manuel

```
- name: 📦 Cache UV dependencies
  uses: actions/cache@v4
  with:
    path: ~/.cache/uv
    key: ${{ runner.os }}-uv-${{ hashFiles('backend/pyproject.toml', 'bac
    restore-keys: |
      ${{ runner.os }}-uv-
```

Comment ça marche ?

1. `path` : Où sont stockées les dépendances
2. `key` : Clé unique basée sur le contenu des fichiers
 - `${{ runner.os }}` : Linux/macOS/Windows
 - `${{ hashFiles(...) }}` : Hash MD5 des fichiers de dépendances
3. `restore-keys` : Clés de fallback si la clé exacte n'existe pas

Quand utiliser un cache manuel ?

- Pour des outils qui n'ont pas de cache automatique
- Pour cacher des artifacts de build (compilés binaires, etc.)

-  Pour optimiser des étapes personnalisées

Note : Pour Python et Node.js, le cache automatique suffit dans 99% des cas !

BONUS 2 : Workflow Java (Optionnel)

Pour les étudiants qui ont fait les exercices Java du TP 1.

Objectif

Tester automatiquement les 3 projets Java (calculator, string-utils, bank-account).


Instructions


1. Créez `.github/workflows/java.yml` :

```
name: Java Tests (Optional)

on:
  push:
    branches: [main]
    paths:
      - 'java-exercises/**'
  pull_request:
    branches: [main]
    paths:
      - 'java-exercises/**'
  workflow_dispatch:

jobs:
  test:
    name: Test Java Exercises
    runs-on: ubuntu-latest

    steps:
      - name:  Checkout code
        uses: actions/checkout@v4

      - name:  Setup Java
```

```

uses: actions/setup-java@v4
with:
  distribution: 'temurin'
  java-version: '17'

- name: 🧮 Test Calculator
  working-directory: java-exercises/calculator
  run: |
    javac -cp ../../lib/junit-4.13.2.jar:../lib/hamcrest-core-1.3.jar *.java
    java -cp ../../lib/junit-4.13.2.jar:../lib/hamcrest-core-1.3.jar CalculatorTest

- name: 📝 Test String Utils
  working-directory: java-exercises/string-utils
  run: |
    javac -cp ../../lib/junit-4.13.2.jar:../lib/hamcrest-core-1.3.jar *.java
    java -cp ../../lib/junit-4.13.2.jar:../lib/hamcrest-core-1.3.jar StringUtilsTest

- name: 🏠 Test Bank Account
  working-directory: java-exercises/bank-account
  run: |
    javac -cp ../../lib/junit-4.13.2.jar:../lib/hamcrest-core-1.3.jar *.java
    java -cp ../../lib/junit-4.13.2.jar:../lib/hamcrest-core-1.3.jar BankAccountTest

```

Nouveaux concepts

paths: - Déclenchement conditionnel

```

on:
  push:
    paths:
      - 'java-exercises/**'

```

Le workflow ne s'exécute que si vous modifiez des fichiers Java.

workflow_dispatch: - **Lancement manuel** Vous pouvez lancer le workflow manuellement depuis l'onglet Actions.

working-directory: - Répertoire de travail

```
- name: Test Calculator  
  working-directory: java-exercises/calculator
```

Plus propre que d'utiliser `cd` dans chaque commande.

Test

Option 1 : Modifier un fichier Java

```
echo "// Test CI" >> java-exercises/calculator/Calculator.java  
git add java-exercises/  
git commit -m "test: trigger Java workflow"  
git push
```

Option 2 : Lancement manuel

1. Allez dans "Actions" → "Java Tests (Optional)"
2. Cliquez sur "Run workflow"
3. Sélectionnez "main" et cliquez "Run workflow"