🚀 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. ✅ 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

⚙️ Prérequis : Vérifier votre connexion GitHub

Avant de commencer, assurez-vous que vous pouvez pousser du code sur votre repository GitHub.

1. Allez dans votre repository cloné

```
cd chemin/vers/votre/repository
```

2. Testez votre connexion

```
git push origin main
```

Résultats possibles :

- ✅ **Everything up-to-date** → Parfait, vous êtes prêt !
- ❌ **Error: authentication failed** → Suivez les instructions ci-dessous

3. Configuration HTTPS (si vous avez une erreur)

Créer un Personal Access Token (PAT)

1. Sur GitHub, allez dans : **Settings** → **Developer settings** → **Personal access tokens** → **Tokens (classic)**
2. Cliquez sur **"Generate new token (classic)"**
3. Configurez :
   - Note : `TP EDL`
   - Expiration : `90 days`
   - Cochez : ✅ `repo` (accès complet aux repositories)

4. Cliquez sur **"Generate token"**
5. ⚠️ **Copiez le token immédiatement** (vous ne pourrez plus le revoir !)

Utiliser le token

Lors du prochain `git push`, Git vous demandera :

- **Username :** Votre nom d'utilisateur GitHub
- **Password : Collez votre token** (PAS votre mot de passe GitHub !)

📚 Structure d'un Workflow GitHub Actions

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

Structure de base :

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

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

jobs:                       # 3️⃣ Les tâches à faire
  test:
    runs-on: ubuntu-latest  # 4️⃣ Machine virtuelle
    steps:                  # 5️⃣ 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` avec ce squelette :**

```
name: Backend Tests

on:
  push:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest

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

      # TODO : Setup Python 3.11 avec cache pip

      # TODO : Installez UV avec pip

      # TODO : Installez les dépendances (cd backend && uv sync)
```

```
      # TODO : Lancez les tests
```

3. **Complétez les TODO en vous aidant de la documentation**

   📖 **Documentation utile :**

   - actions/setup-python : Pour Setup Python avec cache

   💡 **Indices :**

   - **Setup Python** : Utilisez `actions/setup-python@v5` avec les paramètres :
     - `python-version: '3.11'`
     - `cache: 'pip'` (pour activer le cache automatique)
   - **Commandes shell** : Utilisez `run:` suivi de la commande
   - **Multi-lignes** : Utilisez `run: |` pour exécuter plusieurs commandes

   Exemple de step avec run :

```
- name: Mon étape
  run: |
    cd mon-dossier
    ma-commande
```

4. **Testez localement avant de pousser :**

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

5. **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` avec ce squelette :**

```yaml
name: Frontend Tests

on:
  push:
    branches: [main]

jobs:
  test:
    runs-on: ubuntu-latest

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

      # TODO : Setup Node.js 18 avec cache npm
      # Indice : cache-dependency-path: frontend/package-lock.json

      # TODO : Installez les dépendances (cd frontend && npm ci)
```

```
      # TODO : Lancez les tests

      # TODO : Vérifiez le build
```

2. **Complétez les TODO en vous aidant de la documentation**

   📖 **Documentation utile :**

   - actions/setup-node : Pour Setup Node.js avec cache

   💡 **Indices :**

   - **Setup Node** : Utilisez `actions/setup-node@v4` avec les paramètres :
     - `node-version: '18'`
     - `cache: 'npm'`
     - `cache-dependency-path: frontend/package-lock.json`
   - **npm ci** : Installe exactement les versions de `package-lock.json`

3. **Testez localement :**

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

4. **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 : Déboguer 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

📖 C'est quoi une Pull Request (PR) ?

Concept simple : Au lieu de pousser directement sur `main`, vous créez une **branche**, faites vos modifications, puis demandez à merger via une **Pull Request**.

Workflow visuel :

```
main ──────────────────────────●  (production, protégée)
                               ↑
                        [Pull Request]
                               |
                        ✅ Tests passent ?
```

```
                                  |
feature/ma-branche ──●──────●──┘  (votre travail)
```

Avantages :

- ✅ Les tests s'exécutent **avant** le merge
- ✅ Quelqu'un peut **review** votre code
- ✅ La branche `main` reste **stable**

Dans cet exercice, vous allez vivre cette situation !

🎯 Objectif

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

📖 Partie 1 : Activer la Protection de Branche

⚠ **Prérequis :** Vous devez avoir déjà poussé vos workflows des exercices 1 et 2, et ils doivent avoir tourné au moins une fois.

1. **Sur GitHub, allez dans votre repository → Settings → Branches**
2. **Cliquez sur "Add branch protection rule"**
3. **Configurez ces paramètres :**

   a) Branch name pattern

   - Entrez : `main`

   b) Require status checks to pass before merging

   - ✅ Cochez la case **"Require status checks to pass before merging"**
   - Dans la barre de recherche qui apparaît, tapez : `test`
   - Cliquez sur `test` dans les résultats (c'est le nom du job de vos workflows)

   💡 **Si `test` n'apparaît pas :** Vos workflows n'ont pas encore tourné. Retournez sur l'onglet "Actions" et vérifiez qu'ils ont bien été exécutés.

4. **Scrollez tout en bas et cliquez sur "Create"**

⚠ **Important :** Ne cochez RIEN d'autre ! Les autres options sont pour des cas avancés.

🧪 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 :**

```
# Si c'est la première fois, configurez le repo par défaut
gh repo set-default
# Sélectionnez VOTRE fork (pas le repo d'origine)

# Créez la PR
gh pr create --title "Test branch protection" --body "Testing if broke
```

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.

✍️ 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 `pull_request` et `workflow_call` aux déclencheurs :

   ```
   on:
     push:
       branches: [main]
     pull_request:  # ✨ Nouveau !
       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.addinivalue_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 : Créer plusieurs tâches et les lister."""
    # Créer la première tâche
    response = client.post("/tasks", json={
        "title": "Tâche E2E 1",
        "description": "Première tâche"
    })
    assert response.status_code == 201
    task1_id = response.json()["id"]

    # Créer la deuxième tâche
    response = client.post("/tasks", json={
        "title": "Tâche E2E 2",
        "description": "Deuxième tâche"
    })
    assert response.status_code == 201
    task2_id = response.json()["id"]

    # Lister toutes les tâches
    response = client.get("/tasks")
    assert response.status_code == 200
    tasks = response.json()
    assert len(tasks) >= 2

    # Vérifier que nos deux tâches sont dans la liste
    task_ids = [task["id"] for task in tasks]
    assert task1_id in task_ids
    assert task2_id in task_ids
```

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 : Modifier le workflow backend pour séparer les tests

4. **Modifiez `.github/workflows/backend.yml` pour ajouter 2 jobs au lieu d'un seul :**

   Remplacez le contenu complet par :

```
name: Backend Tests

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]
  workflow_call:

jobs:
  unit-tests:
    name: Unit Tests
    runs-on: ubuntu-latest

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

      - name: 🐍 Setup Python 3.11
        uses: actions/setup-python@v5
        with:
          python-version: '3.11'
          cache: 'pip'

      - name: 📦 Install UV
        run: pip install uv

      - name: 📚 Install dependencies
        run: |
```

```
          cd backend
          uv sync

      - name: 🧪 Run unit tests only
        run: |
          cd backend
          uv run pytest -v -m "not e2e"

  e2e-tests:
    name: E2E Tests
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'  # ✨ Seulement sur main !
    needs: unit-tests

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

      - name: 🐍 Setup Python 3.11
        uses: actions/setup-python@v5
        with:
          python-version: '3.11'
          cache: 'pip'

      - name: 📦 Install UV
        run: pip install uv

      - name: 📚 Install dependencies
        run: |
          cd backend
          uv sync

      - name: 🧪 Run E2E tests only
        run: |
          cd backend
          uv run pytest -v -m "e2e"
```

Changements clés :

- **2 jobs** au lieu d'un seul : `unit-tests` et `e2e-tests`
- `if: github.ref == 'refs/heads/main'` → Le job E2E ne tourne que sur main

- `needs: unit-tests` → Les E2E attendent que les tests unitaires passent

5. **Testez le comportement différent entre PR et main :**

   Étape 1 : Tester sur main d'abord

   Poussez vos changements sur main :

   ```
   git add .
   git commit -m "feat: split unit and E2E tests"
   git push origin main
   ```

   → Sur GitHub, allez dans **Actions** → Cliquez sur le workflow **"Backend Tests"** → Vous devriez voir **les deux jobs : "Unit Tests" ET "E2E Tests"** s'exécuter.

   Étape 2 : Tester sur une PR

   Créez une branche et une PR pour voir que les E2E ne tournent pas :

   ```
   # Assurez-vous d'être sur main et à jour
   git checkout main
   git pull origin main

   # Créez une nouvelle branche
   git checkout -b test/pr-no-e2e
   echo "# Test PR" >> README.md
   git add .
   git commit -m "test: verify E2E don't run on PR"
   git push origin test/pr-no-e2e
   gh pr create --title "Test E2E sur PR" --body "Vérifier que E2E ne tou
   ```

   → Sur GitHub, allez dans **Actions** → Cliquez sur le workflow **"Backend Tests"** de la PR → Vous devriez voir **seulement le job "Unit Tests"** s'exécuter (pas de E2E Tests).

✅ Résultat attendu

- **Sur PR** : Seulement le job **"Unit Tests"** s'exécute ⚡ (rapide)
- **Sur main** : Les jobs **"Unit Tests" ET "E2E Tests"** s'exécutent 🐢 (plus lent mais complet)

✍️ Exercice 7 : 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** : 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

Déboguer : 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 débloquera

❌ 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 : 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:
```

```yaml
    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.ja
          java -cp .:../lib/junit-4.13.2.jar:../lib/hamcrest-core-1.3.jar

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

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

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"