🚀 Atelier 2 : CI/CD avec GitHub Actions

Durée estimée : 2h30 **Prérequis :** Atelier 1 terminé (backend et frontend avec tests)

🎯 Objectifs de l'Atelier

Objectif principal : Automatiser les tests avec GitHub Actions

À la fin de cet atelier, vous aurez **construit** :

1. ✅ Un **workflow backend** qui teste automatiquement votre code Python
2. ✅ Un **workflow frontend** qui teste et build votre code TypeScript
3. ✅ Compris comment **déboguer** un workflow qui échoue
4. ✅ Créé votre premier **pipeline CI/CD** complet

📦 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 (Atelier 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

Phase 1 : Comprendre GitHub Actions (20 min)

Étape 1.1 : Anatomie d'un Workflow

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]        # Sur push vers main
  pull_request:
    branches: [main]        # Sur pull request vers main

jobs:                       # 3️⃣ Les tâches à faire
  test:                     # Nom du job
    runs-on: ubuntu-latest  # 4️⃣ Machine virtuelle Linux

    steps:                  # 5️⃣ Les étapes du job
      - name: Récupérer le code
        uses: actions/checkout@v4    # ✅ Action pré-faite

      - name: Lancer les tests
        run: pytest                  # ✅ Commande shell
```

Concepts clés :

1. `name` : Le nom qui apparaît sur GitHub
2. `on` : Les déclencheurs (push, pull_request, schedule, etc.)
3. `jobs` : Les tâches (peuvent s'exécuter en parallèle)
4. `runs-on` : Le système d'exploitation (ubuntu, windows, macos)
5. `steps` : Les étapes du job (séquentielles)

Deux types de steps :

- `uses` : Utilise une action pré-faite (ex: `actions/checkout@v4`)
- `run` : Exécute une commande shell (ex: `pytest`)

Étape 1.2 : Où Trouver les Actions ?

Actions officielles GitHub :

- `actions/checkout@v4` - Clone le repo
- `actions/setup-python@v5` - Installe Python
- `actions/setup-node@v4` - Installe Node.js

Marketplace :

- https://github.com/marketplace?type=actions
- Des milliers d'actions pré-faites

Documentation :

- https://docs.github.com/en/actions

Phase 2 : Workflow Backend (40 min)

Étape 2.1 : Créer le Fichier Workflow

```
mkdir -p .github/workflows
touch .github/workflows/backend.yml
```

Étape 2.2 : Écrire le Workflow Backend

Ouvrez `.github/workflows/backend.yml` et copiez ce contenu :

```
name: Backend Tests
```

```yaml
on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  test:
    name: Test Backend
    runs-on: ubuntu-latest

    steps:
      # Étape 1 : Récupérer le code
      - name: 📥 Checkout code
        uses: actions/checkout@v4

      # Étape 2 : Installer Python
      - name: 🐍 Setup Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.11'

      # Étape 3 : Installer UV
      - name: 📦 Install UV
        run: |
          curl -LsSf https://astral.sh/uv/install.sh | sh
          echo "$HOME/.cargo/bin" >> $GITHUB_PATH

      # Étape 4 : Installer les dépendances
      - name: 📚 Install dependencies
        run: |
          cd backend
          uv sync

      # Étape 5 : Lancer les tests
      - name: 🧪 Run tests
        run: |
          cd backend
          uv run pytest -v --cov
```

Étape 2.3 : Comprendre Chaque Ligne

Ligne par ligne :

```
name: Backend Tests           # Nom affiché dans l'onglet Actions
```

```
on:
  push:
    branches: [main]          # Déclenche sur push vers main
  pull_request:
    branches: [main]          # Déclenche sur PR vers main
```

```
jobs:
  test:                       # ID du job
    name: Test Backend        # Nom affiché
    runs-on: ubuntu-latest    # Ubuntu (gratuit et rapide)
```

```
steps:
  - name: 📥 Checkout code
    uses: actions/checkout@v4  # Clone le repo
```

Pourquoi `actions/checkout@v4` ?

- Sans ça, GitHub Actions ne voit pas votre code !
- C'est toujours la première étape

```
  - name: 🐍 Setup Python
    uses: actions/setup-python@v5
    with:
      python-version: '3.11'   # Version Python
```

```
  - name: 📦 Install UV
    run: |
      curl -LsSf https://astral.sh/uv/install.sh | sh
      echo "$HOME/.cargo/bin" >> $GITHUB_PATH
```

Explication :

- `curl -LsSf ... | sh` : Télécharge et installe UV

- `echo "$HOME/.cargo/bin" >> $GITHUB_PATH` : Ajoute UV au PATH pour les étapes suivantes
- Sans cette ligne, `uv` ne serait pas trouvé dans les étapes suivantes

```
  - name: 📚 Install dependencies
    run: |                      # | permet plusieurs lignes
      cd backend
      uv sync
```

```
  - name: 🧪 Run tests
    run: |
      cd backend
      uv run pytest -v --cov
```

Important : Ce sont les **mêmes commandes** que vous exécutez localement !

Étape 2.4 : Tester Localement Avant de Pousser

Avant de pousser, vérifiez que ça marche localement :

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

✅ Si ça passe localement, ça devrait passer sur GitHub !

Étape 2.5 : Pousser et Observer

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

Observer sur GitHub :

1. Allez sur votre repo GitHub

2. Cliquez sur l'onglet **Actions**
3. Vous verrez votre workflow en cours d'exécution
4. Cliquez dessus pour voir les détails

Résultat attendu :

```
✅ Backend Tests
  └─ Test Backend
      ├─ 📥 Checkout code
      ├─ 🐍 Setup Python
      ├─ 📦 Install UV
      ├─ 📚 Install dependencies
      └─ 🧪 Run tests
```

Phase 3 : Workflow Frontend (40 min)

Étape 3.1 : Créer le Workflow Frontend

```
touch .github/workflows/frontend.yml
```

Étape 3.2 : Écrire le Workflow Frontend

Ouvrez `.github/workflows/frontend.yml` :

```yaml
name: Frontend Tests

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

jobs:
  test:
    name: Test Frontend
    runs-on: ubuntu-latest
```

```
    steps:
      # Étape 1 : Récupérer le code
      - name: 📥 Checkout code
        uses: actions/checkout@v4

      # Étape 2 : Installer Node.js
      - name: 🟢 Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '18'
          cache: 'npm'
          cache-dependency-path: frontend/package-lock.json

      # Étape 3 : Installer les dépendances
      - name: 📦 Install dependencies
        run: |
          cd frontend
          npm ci

      # Étape 4 : Lancer les tests
      - name: 🧪 Run tests
        run: |
          cd frontend
          npm test -- --run

      # Étape 5 : Vérifier le build
      - name: 🏗️ Build check
        run: |
          cd frontend
          npm run build
```

Étape 3.3 : Comprendre les Différences avec le Backend

`npm ci` vs `npm install` :

```
- name: 📦 Install dependencies
  run: npm ci    # ✅ Plus rapide et déterministe (pour CI)
```

- `npm ci` : Installe exactement ce qui est dans `package-lock.json`

- `npm install` : Peut mettre à jour les versions (moins fiable)

Cache npm :

```
- name: 🟢 Setup Node.js
  uses: actions/setup-node@v4
  with:
    cache: 'npm'   # ✅ Met en cache node_modules
```

Accélère les builds (évite de re-télécharger chaque fois).

Tests en mode "run once" :

```
npm test -- --run   # ✅ Lance les tests une fois (pas en mode watch)
```

Build check :

```
npm run build   # ✅ Vérifie que le build fonctionne (détecte les erreurs
```

Étape 3.4 : Pousser et Observer

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

Vous verrez maintenant 2 workflows en parallèle :

```
✅ Backend Tests
✅ Frontend Tests
```

Les deux s'exécutent en même temps ! 🚀

Phase 4 : Déboguer un Échec Volontaire (30 min)

Étape 4.1 : Pourquoi Apprendre à Déboguer ?

Dans la vraie vie :

- ❌ Les workflows échouent souvent
- 🔍 Il faut savoir lire les logs
- 🐛 Reproduire localement pour corriger

Apprenons en cassant quelque chose exprès !

✍️ Exercice : Introduire un Bug (10 min)

Objectif : Modifier un test pour qu'il échoue volontairement.

Ouvrez `backend/tests/test_api.py` et **modifiez** le test `test_health_check` :

```
def test_health_check(client):
    """The health endpoint should confirm the API is running."""
    response = client.get("/health")

    assert response.status_code == 200
    assert response.json()["status"] == "BROKEN"  # ❌ Volontairement fau
```

Pourquoi "BROKEN" ?

- Le vrai statut est `"healthy"`
- Ce test va échouer !

Pousser le bug :

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

Étape 4.2 : Observer l'Échec (5 min)

Sur GitHub Actions :

1. Allez dans l'onglet **Actions**
2. Vous verrez ❌ **Backend Tests** en rouge
3. Cliquez dessus

Vous verrez :

```
❌ Backend Tests
  └─ Test Backend
      ├─ ✅ 📩 Checkout code
      ├─ ✅ 🐍 Setup Python
      ├─ ✅ 📦 Install UV
      ├─ ✅ 📚 Install dependencies
      └─ ❌ 🧪 Run tests  ← ICI LE PROBLÈME
```

Étape 4.3 : Analyser les Logs (10 min)

Cliquez sur l'étape "🧪 Run tests".

Vous verrez les logs :

```
tests/test_api.py::test_health_check FAILED

================================= FAILURES =================================
_________________________ test_health_check _________________________

client = <starlette.testclient.TestClient object at 0x...>

    def test_health_check(client):
        response = client.get("/health")
        assert response.status_code == 200
>       assert response.json()["status"] == "BROKEN"
E       AssertionError: assert 'healthy' == 'BROKEN'
E         - BROKEN
E         + healthy

tests/test_api.py:20: AssertionError
```

```
======================== short test summary info ========================
FAILED tests/test_api.py::test_health_check - AssertionError: ...
======================== 1 failed, 18 passed in 0.52s ===================
```

Questions à se poser :

1. **Quel test échoue ?** → `test_health_check`
2. **Quelle ligne ?** → `tests/test_api.py:20`
3. **Quelle est l'erreur ?** → Attend "BROKEN", reçoit "healthy"
4. **Comment reproduire localement ?**

Étape 4.4 : Reproduire Localement (5 min)

Même commande que dans le workflow :

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

Vous verrez la même erreur !

```
FAILED tests/test_api.py::test_health_check - AssertionError: assert 'hea
```

Maintenant corrigez :

```
def test_health_check(client):
    """The health endpoint should confirm the API is running."""
    response = client.get("/health")

    assert response.status_code == 200
    assert response.json()["status"] == "healthy"  # ✅ Correct !
```

Vérifiez localement :

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

✅ **Le test passe !**

Étape 4.5 : Pousser la Correction (5 min)

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

Sur GitHub Actions :

```
✅ Backend Tests  ← De nouveau vert !
```

Étape 4.6 : Leçons Apprises

Ce que vous avez appris :

1. ✅ Lire les logs GitHub Actions
2. ✅ Identifier la ligne qui échoue
3. ✅ Reproduire l'erreur localement
4. ✅ Corriger et vérifier
5. ✅ Re-pousser

Principe clé : Si ça passe localement, ça passera sur GitHub !

Phase 5 : Vérification Finale (20 min)

Étape 5.1 : Créer une Pull Request (10 min)

Pourquoi une PR ?

Les workflows s'exécutent aussi sur les Pull Requests !

Créer une branche :

```
git checkout -b feature/test-pr
```

Faire un petit changement :

```
# Dans backend/src/app.py
@app.get("/")
async def root():
    return {
        "message": "Welcome to TaskFlow API v2.0",  # Changé !
        "version": "1.0.0",
        "docs": "/docs"
    }
```

Pousser la branche :

```
git add backend/src/app.py
git commit -m "feat: update welcome message"
git push origin feature/test-pr
```

Créer la PR sur GitHub :

1. Allez sur votre repo GitHub
2. Cliquez sur **"Compare & pull request"**
3. Créez la PR

Vous verrez les checks s'exécuter :

```
⏳ Backend Tests - In progress
⏳ Frontend Tests - In progress
```

Puis :

```
✅ Backend Tests - Passed
✅ Frontend Tests - Passed
✅ All checks have passed
```

Vous pouvez maintenant merger en toute confiance !

🎁 BONUS : Workflow Java (Optionnel - 30 min)

Pour les étudiants qui ont terminé les 5 phases principales.

Objectif

Appliquer les concepts CI/CD sur les exercices Java de l'Atelier 1.

Étape Bonus 1 : Rappel des Exercices Java

Si vous avez fait les exercices BONUS de l'Atelier 1, vous avez 3 projets Java :

```
java-exercises/
├── calculator/        # Calculatrice avec opérations de base
├── string-utils/      # Manipulation de chaînes
└── bank-account/      # Gestion de compte bancaire
```

Étape Bonus 2 : Créer le Workflow Java

Créez `.github/workflows/java.yml` :

```yaml
name: Java Tests (Optional)

# Workflow optionnel pour les exercices bonus Java
on:
  push:
    branches: [main]
    paths:
      - 'java-exercises/**'
  pull_request:
    branches: [main]
    paths:
      - 'java-exercises/**'
  workflow_dispatch:  # Permet lancement manuel
```

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

    steps:
      # Étape 1 : Récupérer le code
      - name: 📥 Checkout code
        uses: actions/checkout@v4

      # Étape 2 : Installer Java
      - name: ☕ Setup Java
        uses: actions/setup-java@v4
        with:
          distribution: 'temurin'
          java-version: '17'

      # Étape 3 : Tester Calculator
      - 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

      # Étape 4 : Tester String Utils
      - 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

      # Étape 5 : Tester Bank Account
      - 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
```

Étape Bonus 3 : Comprendre les Différences

`paths:` - Déclenchement Conditionnel

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

➡️ Le workflow ne s'exécute **que** si vous modifiez des fichiers dans `java-exercises/`

`workflow_dispatch:` - Lancement Manuel

```
on:
  workflow_dispatch:
```

➡️ Vous pouvez lancer le workflow manuellement depuis l'onglet **Actions** sur GitHub

`working-directory:` - Répertoire de Travail

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

➡️ Définit le répertoire de travail pour toutes les commandes `run` de cette étape

Pourquoi `working-directory` au lieu de `cd` ?

- ✅ Plus propre et plus clair
- ✅ Fonctionne mieux avec les chemins relatifs
- ✅ Standard GitHub Actions

`javac` et `java` - Compilation et Exécution

```
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 org.junit
```

- `-cp` : Classpath (où trouver JUnit)

- `.:../lib/...` : Dossier actuel + JARs dans ../lib
- `*.java` : Compile tous les fichiers Java
- `JUnitCore` : Lance les tests JUnit

Étape Bonus 4 : Tester le Workflow

Option 1 : Push un changement Java

```
# 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 sur **Actions** dans GitHub
2. Cliquez sur **Java Tests (Optional)**
3. Cliquez sur **Run workflow**
4. Sélectionnez la branche `main`
5. Cliquez sur **Run workflow**

Étape Bonus 5 : Voir les Résultats

Vous devriez voir dans les logs :

```
🧮 Test Calculator
  Compiling...
  Running tests...
  JUnit version 4.13.2
  ..........
  Time: 0.012
  OK (10 tests)
```

```
📝 Test String Utils
  ...

🏦 Test Bank Account
  ...
```

✅ **Tous vos exercices Java sont testés automatiquement !**

🤔 Exercice de Réflexion

Pourquoi 3 workflows séparés (backend, frontend, java) plutôt qu'un seul ?

▶ Cliquez pour voir la réponse

🐛 Erreurs Fréquentes

❌ Workflow ne se déclenche pas

Cause : Fichier mal placé ou syntaxe YAML invalide

Solution : Vérifiez :

- Le fichier est dans `.github/workflows/`
- L'extension est `.yml` ou `.yaml`
- Pas d'erreurs de syntaxe (indentation !)

❌ `uv: command not found`

Cause : UV n'est pas dans le PATH après installation

Solution : Ajoutez `echo "$HOME/.cargo/bin" >> $GITHUB_PATH` après l'installation de UV

❌ `actions/checkout@v4` ne fonctionne pas

Cause : Problème de permissions GitHub

Solution : Ajoutez l'étape `actions/setup-node@v4`

❌ Tests qui passent localement mais échouent sur GitHub

Causes possibles :

1. Variable d'environnement manquante
2. Dépendance système manquante
3. Timezone différente

Déboguer : Reproduisez exactement les mêmes commandes localement

---

📚 Ressources

- Documentation GitHub Actions
- Marketplace Actions
- YAML Syntax
- Actions Workflow Syntax

🚀 Prochaine Étape : Atelier 3

Dans l'Atelier 3, vous allez **déployer votre application** :

- Migrer vers PostgreSQL (base de données réelle)
- Déployer sur Render (production)
- Configurer le CD (Continuous Deployment)

Prêt pour la production ? 🚀

Version 2.0 - Atelier 2 CI/CD Simplifié