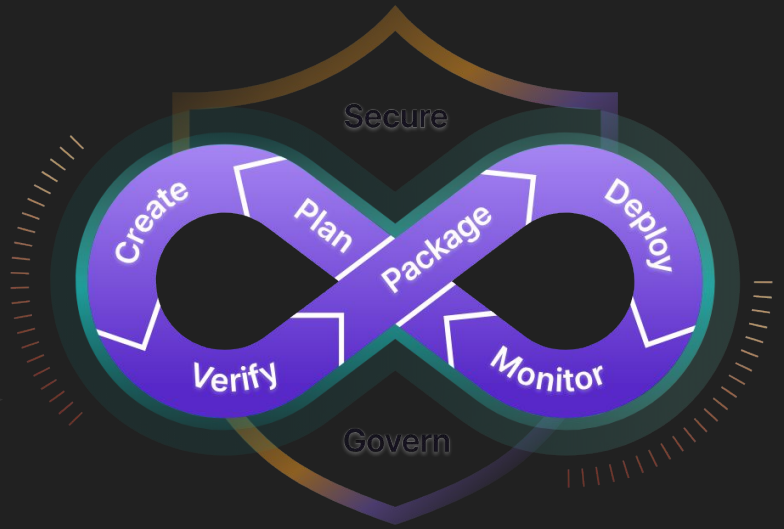


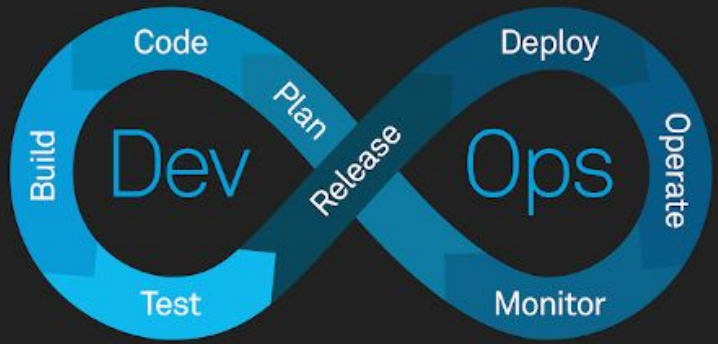
Atelier Github-Actions

18/02/26

CI/CD

- pratiques qui automatisent l'intégration, les tests et la livraison du code.
  - **CI** (Continuous Integration): tester et valider automatiquement le code.
  - **CD** (Continuous Delivery / Deployment) : Livrer ou déployer automatiquement après validation.

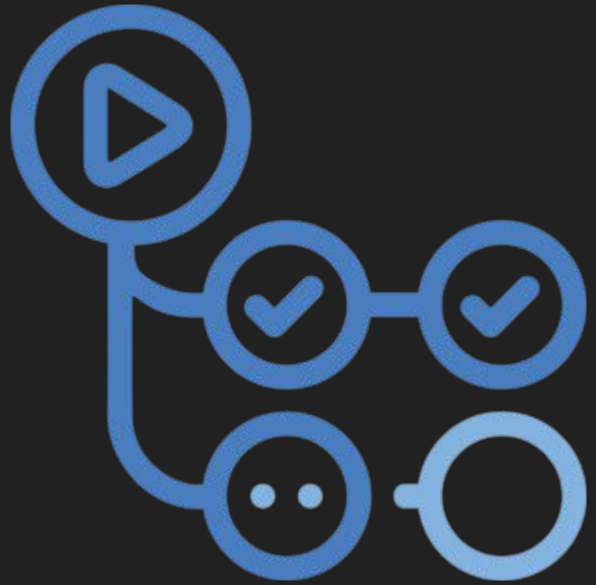
→ rendre les livraisons plus rapides, plus fiables et répétables.



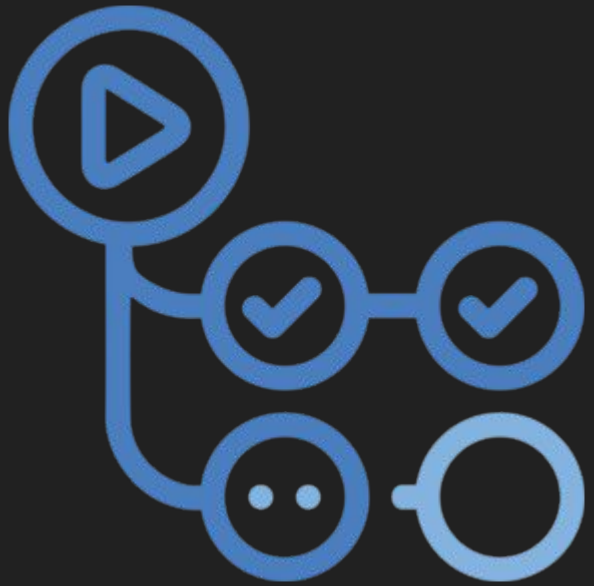
Github actions et CI/CD

- Où se place GitHub Actions dans l’écosystème ?
  - GitHub Actions est un outil de CI/CD intégré directement à GitHub.

- Permet de :
  - définir des workflows automatisés
  - lancer des tests
  - réagir aux événements GitHub (push, pull request, merge…)



Github-Actions, c'est :

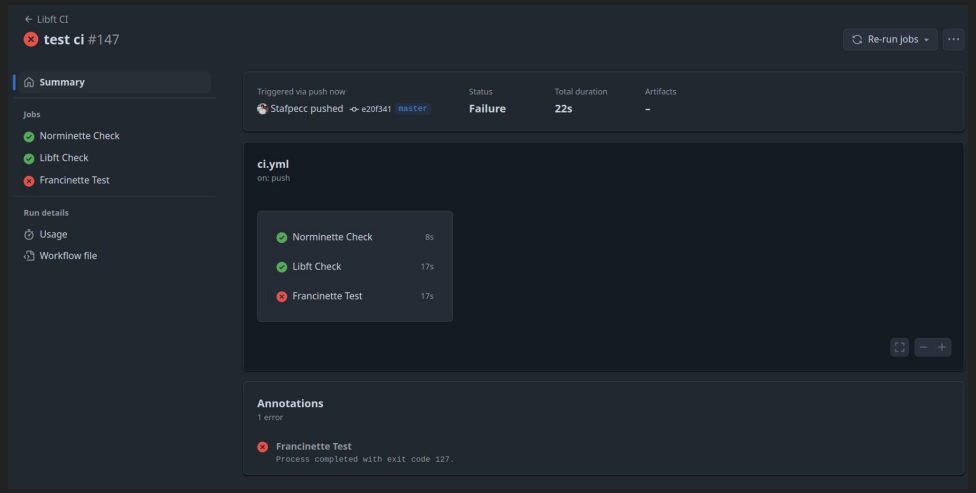


- GitHub Actions sert à automatiser des workflows (tests, builds, déploiements) directement depuis GitHub.

- 🔁 **Automatiser les tâches répétitives**
  → Plus besoin de lancer manuellement les tests, builds, ou déploiements
- ✅ **Améliorer la qualité du code**
  → Lancer automatiquement des tests à chaque `push` ou `pull request`
- 🚀 **Déployer facilement et rapidement (CI/CD)**
  → Déploiement continu dès qu'une branche est validée
- 🧩 **S'intégrer parfaitement à GitHub**
  → Pas besoin d'outil externe, déclenchement natif sur vos dépôts

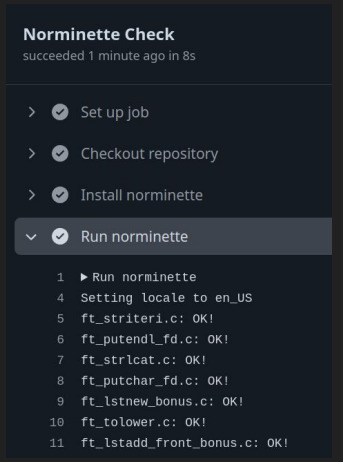
Pourquoi Github-Actions ?

- ⏱️ **Gain de temps** : automatisation des tâches répétitives
- ✅ **Qualité du code** : exécuter des tests automatiquement à chaque push
- 🚀 **Déploiement continu** : déployer automatiquement une app après validation
- 🔗 **Intégré à GitHub** : pas besoin de service externe
- 🧩 **Large écosystème** : des centaines d'actions prêtes à l'emploi

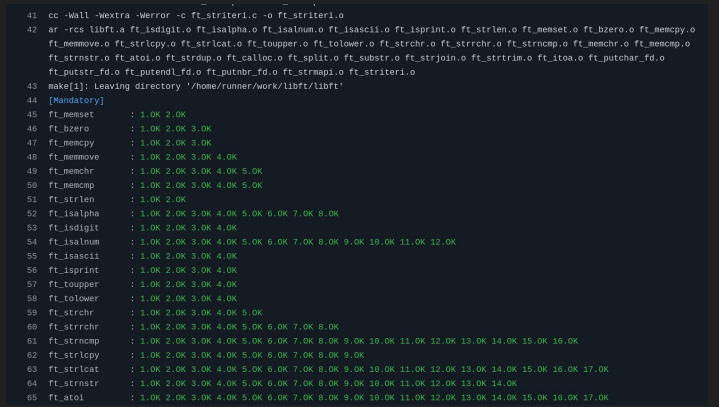


L’onglet “Actions” sur Github

- Chaque workflow apparaît avec :
  - Son **nom** (défini dans le fichier YAML)
  - L’**événement déclencheur** (`push`, `pull_request`, etc.)
  - La **branche concernée**
  - Le **statut** (✅ Succès, ❌ Échec, 🟡 En cours)



- En cliquant sur un workflow, on voit :
  - Les **jobs** exécutés (ex: `build`, `test`, `deploy`)
  - Chaque **étape (step)** à l’intérieur du job
  - Les **logs** de chaque commande (`npm install`, `npm test`, etc.)



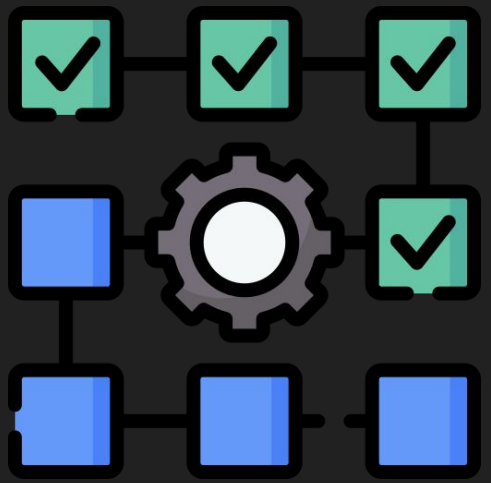
Tests unitaires

- ✅ Vérifier le bon fonctionnement d'une petite unité de code (fonction, méthode, composant) de manière isolée.

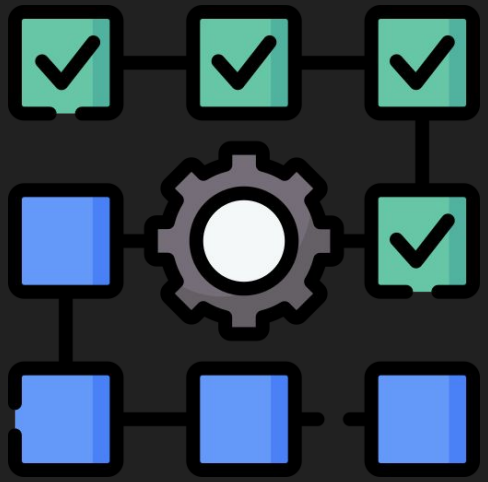
  → s'assurer que chaque brique du code se comporte comme prévu.

- Pourquoi les automatiser ?
  - éviter les tests manuels répétitifs
  - garantir une qualité constante du code
  - donner un feedback immédiat

→ Des tests automatisés = moins de bugs en production.



Tests unitaires



- Bloquer une PR si les tests échouent
  - la CI passe en échec
  - la Pull Request est bloquée
  - le code ne peut pas être fusionné tant que les tests ne passent pas

→ Cela garantit que seul du code validé arrive sur la branche principale.

- Définir les tests unitaires
  - 🔎 Identifier ce qui doit être testé
  - 🧪 Définir des cas de test
  - 🔁 Rendre les tests clairs et maintenables
  - 📄 Nommer et structurer les tests
    - (void test_given_hello_when_strlen_then_returns_5())
  - ✅ Vérifier régulièrement via la CI

Exemple de test unitaires:

```c
#include "strjoin_tests.h"

int ft_strjoin_null_test(void)
{
    char    *result;

    result = ft_strjoin(NULL, NULL);
    if (result == NULL)
    {
        return (0);
    }
    else
    {
        free(result);
        return (-1);
    }
}
```

```c
#include "strdup_tests.h"

int ft_strdup_basic_test(void)
{
    char    *dest;

    dest = ft_strdup("coucou");
    if (dest && ft_strcmp(dest, "coucou") == 0)
    {
        free(dest);
        return (0);
    }
    else
    {
        if (!dest)
            free(dest);
        return (-1);
    }
}
```

.github/workflows/yml

🧱 Syntaxe de base

- **Clé : valeur**
- **Indentation par espaces uniquement** (⚠️ jamais de tabulations)
- **Listes** avec - devant chaque élément

🎯 Pourquoi YAML ?

- ✅ Simple et lisible par les humains
- 🧩 Idéal pour les configurations déclaratives
- 🔄 Supporté nativement par GitHub Actions

Documents

```yaml
document: this is document 1

jedis-list:
  - Yoda
  - Qui-Gon Jinn
  - Obi-Wan Kenobi
  - Luke Skywalker

jedi:
  name: Obi-Wan Kenobi
  home-planet: Stewjon
  height: 1.82m

requests:
  - http://example.com/
  - url: http://example.com/
    method: GET
---
document: this is document 2

reporting:
- module: final-stats
- module: console
---
- item_1
- item_2
```

Construire une ci en Yaml

- **name :** nom du workflow (titre du scénario d'automatisation).
- **on :** événement(s) qui déclenchent le workflow.
- **jobs :** ensemble des jobs (grandes étapes du workflow).
- **runs-on :** type de machine (runner) utilisé pour exécuter le job.
- **steps :** liste des étapes individuelles d'un job.
- **run :** commande exécutée dans une step.

Exemple pour faire tourner des tests

.github > workflows > ! ci.yml

```yaml
name: CI

on:
  push:
    branches:
      - master
  pull_request:
    branches:
      - master

jobs:
  libft-test:
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: .libftTester

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

      - name: Install dependencies
        run: sudo apt-get install -y make gcc valgrind

      - name: Run tests
        run: TERM=xterm make
```

Merci de nous avoir écouté !!

Prenez votre projet actuel/libft **ou le projet que vous voulez** puis :

- 📏 **Ajoutez la Norminette**
- ✅ **Ajoutez une série de tests automatisés (basic, null, empty, one char, big…)**
- ⚙️ Le tout exécuté **automatiquement à chaque push grâce à GitHub Actions**