

Atelier Github-Actions

18/02/26

CI / CD

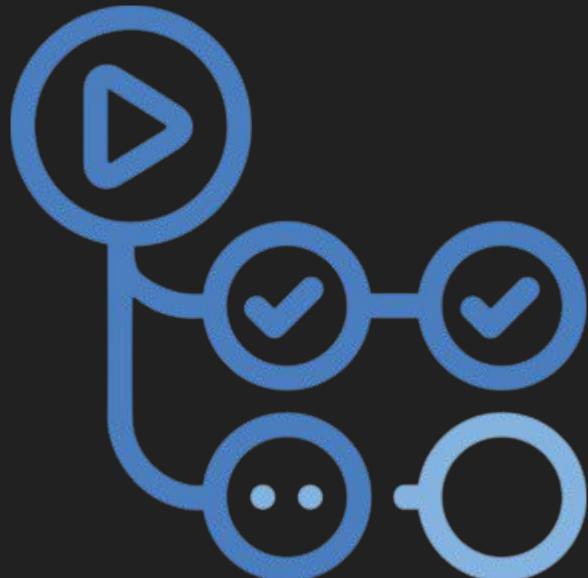
- pratiques qui automatisent l'intégration, les tests et la livraison du code.
 - **CI** (Continuous Integration): tester et valider automatiquement le code.
 - **CD** (Continuous Delivery / Deployment) : Livrer ou déployer automatiquement après validation.

→ 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...)



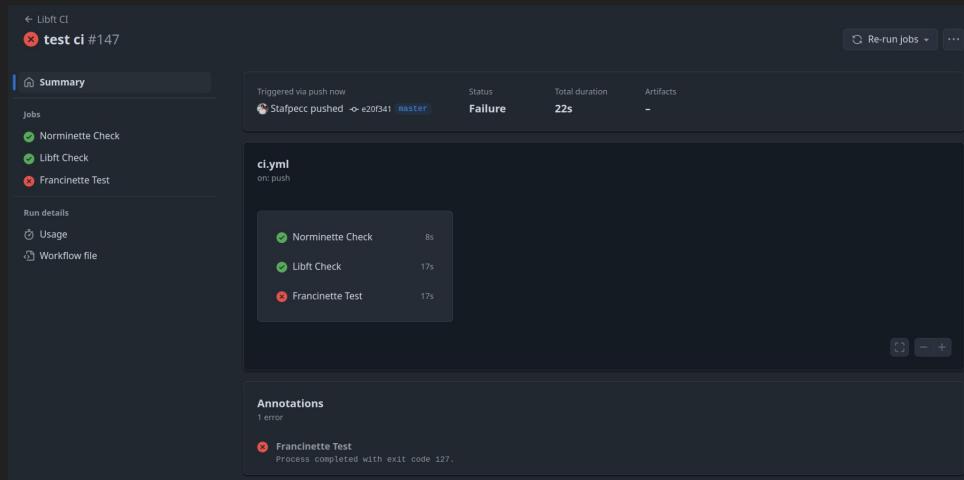
Github-Actions, c'est :



- 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 prêtées 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.)

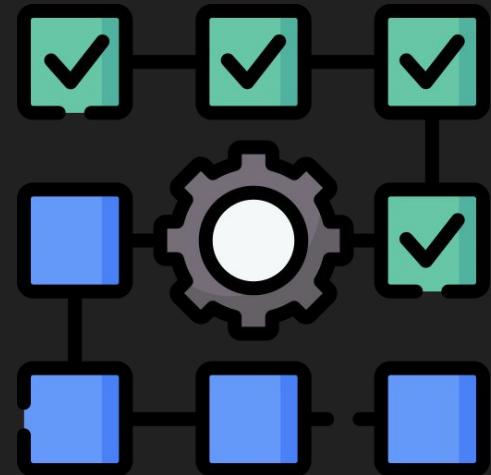
```
1 ► Run norminette
  4 Setting locale to en_US
  5 ft_striteri.c: OK!
  6 ft_putendl_fd.c: OK!
  7 ft_stricat.c: OK!
  8 ft_putchar_fd.c: OK!
  9 ft_lstnew_bonus.c: OK!
  10 ft_tolower.c: OK!
  11 ft_lstadd_front_bonus.c: OK!
```

```
41 cc -Wall -Wextra -Werror -c ft_striteri.c -o ft_striteri.o
42 ar -rcs libft.a ft_isdigit.o ft_isalpha.o ft_isalnum.o ft_isascii.o ft_ftsprint.o ft_strlen.o ft_memset.o ft_bzero.o ft_memcpy.o
   ft_memmove.o ft_strcpy.o ft_strcat.o ft_toupper.o ft_tolower.o ft_strchr.o ft_strichr.o ft_stncmp.o ft_mennchr.o ft_memcmp.o
   ft_putstr.o ft_atoi.o ft_strdup.o ft_calloc.o ft_split.o ft_substr.o ft_strjoin.o ft_strtrim.o ft_itoa.o ft_putchar.o ft_putchar_fd.o
   ft_putstr_fd.o ft_putendl_fd.o ft_putnbr_fd.o ft_strmapi.o ft_strstr.o ft_strstrr.o ft_striteri.o
43 [Mandatory]
44 Leaving directory '/home/runner/work/libft/libft'
45 ft_remake : 1.OK 2.OK
46 ft_bzero : 1.OK 2.OK 3.OK
47 ft_memcpy : 1.OK 2.OK 3.OK
48 ft_memmove : 1.OK 2.OK 3.OK 4.OK
49 ft_mennchr : 1.OK 2.OK 3.OK 4.OK 5.OK
50 ft_memcmp : 1.OK 2.OK 3.OK 4.OK 5.OK
51 ft_strlen : 1.OK 2.OK
52 ft_isalpha : 1.OK 2.OK 3.OK 4.OK 5.OK 6.OK 7.OK 8.OK
53 ft_isdigit : 1.OK 2.OK 3.OK 4.OK
54 ft_isalnum : 1.OK 2.OK 3.OK 4.OK 5.OK 6.OK 7.OK 8.OK 9.OK 10.OK 11.OK 12.OK
55 ft_isascii : 1.OK 2.OK 3.OK 4.OK
56 ft_isprint : 1.OK 2.OK 3.OK 4.OK
57 ft_toupper : 1.OK 2.OK 3.OK 4.OK
58 ft_tolower : 1.OK 2.OK 3.OK 4.OK
59 ft_strichr : 1.OK 2.OK 3.OK 4.OK 5.OK
60 ft_strchr : 1.OK 2.OK 3.OK 4.OK 5.OK 6.OK 7.OK 8.OK
61 ft_strncmp : 1.OK 2.OK 3.OK 4.OK 5.OK 6.OK 7.OK 8.OK 9.OK 10.OK 11.OK 12.OK 13.OK 14.OK 15.OK 16.OK
62 ft_strlcpy : 1.OK 2.OK 3.OK 4.OK 5.OK 6.OK 7.OK 8.OK 9.OK
63 ft_strlcat : 1.OK 2.OK 3.OK 4.OK 5.OK 6.OK 7.OK 8.OK 9.OK 10.OK 11.OK 12.OK 13.OK 14.OK 15.OK 16.OK 17.OK
64 ft_strnsrr : 1.OK 2.OK 3.OK 4.OK 5.OK 6.OK 7.OK 8.OK 9.OK 10.OK 11.OK 12.OK 13.OK 14.OK 15.OK 16.OK 17.OK
65 ft_atoi : 1.OK 2.OK 3.OK 4.OK 5.OK 6.OK 7.OK 8.OK 9.OK 10.OK 11.OK 12.OK 13.OK 14.OK 15.OK 16.OK 17.OK
```

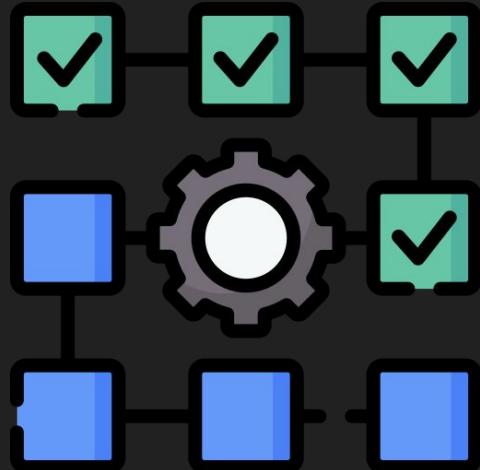
Tests unitaires

-  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:

```
13     #include "strjoin_tests.h"
14
15     int ft_strjoin_null_test(void)
16     {
17         char    *result;
18
19         result = ft_strjoin(NULL, NULL);
20         if (result == NULL)
21         {
22             return (0);
23         }
24         else
25         {
26             free(result);
27             return (-1);
28         }
29     }
30
31     #include "strdup_tests.h"
32
33     int ft_strdup_basic_test(void)
34     {
35         char    *dest;
36
37         dest = ft_strdup("coucou");
38         if (dest && ft_strcmp(dest, "coucou") == 0)
39         {
40             free(dest);
41             return (0);
42         }
43         else
44         {
45             if (!dest)
46                 free(dest);
47             return (-1);
48         }
49     }
```



.github/workflows/yml

brick Syntaxe de base

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

clock Pourquoi YAML ?

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

Documents

```
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

Exemple pour faire tourner des tests

- **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.

```
.github > workflows > ! ci.yml
1   name: CI
2
3   on:
4     push:
5       branches:
6         - master
7     pull_request:
8       branches:
9         - master
10
11  jobs:
12    libft-test:
13      runs-on: ubuntu-latest
14      defaults:
15        run:
16          working-directory: .libftTester
17
18      steps:
19        - name: Checkout code
20          uses: actions/checkout@v4
21
22        - name: Install dependencies
23          run: sudo apt-get install -y make gcc valgrind
24
25        - name: Run tests
26          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