

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

The screenshot displays the GitHub Actions interface for a workflow named 'test ci #147'. The workflow was triggered by a push to the 'master' branch by user 'Stafpecc'. The overall status is 'Failure' with a total duration of 22 seconds. The workflow file is 'ci.yml' and it runs on push events. The jobs section lists three jobs: 'Norminette Check' (8s, success), 'Libft Check' (17s, success), and 'Francinette Test' (17s, failure). The 'Annotations' section shows an error for 'Francinette Test' with the message 'Process completed with exit code 127.'.

| Jobs | Status | Duration |
|------------------|---------|----------|
| Norminette Check | Success | 8s |
| Libft Check | Success | 17s |
| Francinette Test | Failure | 17s |

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.)

Norminette Check


succeeded 1 minute ago in 8s

- > ✅ Set up job
- > ✅ Checkout repository
- > ✅ Install norminette
- ▼ ✅ Run norminette

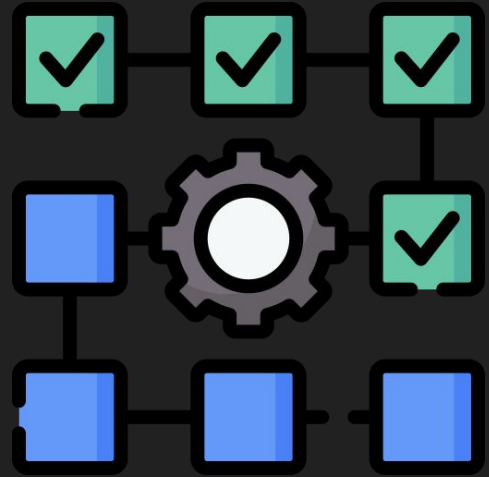
```
1 ▶ Run norminette
4 Setting locale to en_US
5 ft_striteri.c: OK!
6 ft_putendl_fd.c: OK!
7 ft_strlcat.c: OK!
8 ft_putchar_fd.c: OK!
9 ft_lstnew_bonus.c: OK!
10 ft_tolower.c: OK!
11 ft_lstadd_front_bonus.c: OK!
```

```
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_isprint.o ft_strlen.o ft_memset.o ft_bzero.o ft_memcpy.o
ft_memmove.o ft_strncpy.o ft_strcat.o ft_toupper.o ft_tolower.o ft_strchr.o ft_strrchr.o ft_strncmp.o ft_memchr.o ft_memcmp.o
ft_strnstr.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_fd.o
ft_putstr_fd.o ft_putendl_fd.o ft_putnbr_fd.o ft_strnapi.o ft_striteri.o
43 make[1]: Leaving directory '/home/runner/work/libft/libft'
44 [Mandatory]
45 ft_memset : 1.0K 2.0K
46 ft_bzero : 1.0K 2.0K 3.0K
47 ft_memcpy : 1.0K 2.0K 3.0K
48 ft_memmove : 1.0K 2.0K 3.0K 4.0K
49 ft_memchr : 1.0K 2.0K 3.0K 4.0K 5.0K
50 ft_memcmp : 1.0K 2.0K 3.0K 4.0K 5.0K
51 ft_strlen : 1.0K 2.0K
52 ft_isalpha : 1.0K 2.0K 3.0K 4.0K 5.0K 6.0K 7.0K 8.0K
53 ft_isdigit : 1.0K 2.0K 3.0K 4.0K
54 ft_isalnum : 1.0K 2.0K 3.0K 4.0K 5.0K 6.0K 7.0K 8.0K 9.0K 10.0K 11.0K 12.0K
55 ft_isascii : 1.0K 2.0K 3.0K 4.0K
56 ft_isprint : 1.0K 2.0K 3.0K 4.0K
57 ft_toupper : 1.0K 2.0K 3.0K 4.0K
58 ft_tolower : 1.0K 2.0K 3.0K 4.0K
59 ft_strchr : 1.0K 2.0K 3.0K 4.0K 5.0K
60 ft_strrchr : 1.0K 2.0K 3.0K 4.0K 5.0K 6.0K 7.0K 8.0K
61 ft_strncmp : 1.0K 2.0K 3.0K 4.0K 5.0K 6.0K 7.0K 8.0K 9.0K 10.0K 11.0K 12.0K 13.0K 14.0K 15.0K 16.0K
62 ft_strncpy : 1.0K 2.0K 3.0K 4.0K 5.0K 6.0K 7.0K 8.0K 9.0K
63 ft_strcat : 1.0K 2.0K 3.0K 4.0K 5.0K 6.0K 7.0K 8.0K 9.0K 10.0K 11.0K 12.0K 13.0K 14.0K 15.0K 16.0K 17.0K
64 ft_strstr : 1.0K 2.0K 3.0K 4.0K 5.0K 6.0K 7.0K 8.0K 9.0K 10.0K 11.0K 12.0K 13.0K 14.0K
65 ft_atoi : 1.0K 2.0K 3.0K 4.0K 5.0K 6.0K 7.0K 8.0K 9.0K 10.0K 11.0K 12.0K 13.0K 14.0K 15.0K 16.0K 17.0K
```

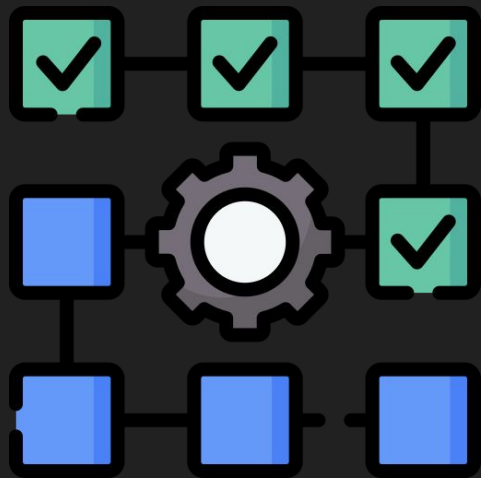
Tests unitaires

-  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  }
```

```
13  #include "strdup_tests.h"
14
15  int ft_strdup_basic_test(void)
16  {
17      char    *dest;
18
19      dest = ft_strdup("coucou");
20      if (dest && ft_strcmp(dest, "coucou") == 0)
21      {
22          free(dest);
23          return (0);
24      }
25      else
26      {
27          if (!dest)
28              free(dest);
29          return (-1);
30      }
31  }
```






.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

```
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
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