



25/04/2024

# Notice explicative de l'infrastructure du projet Git



Maëlle SOPRANSI CIR3

## Introduction au projet

Ce projet a pour objectif de déployer un serveur git sur WSL, avec une interface web accessible depuis Windows fonctionnelle hors internet, qui doit contenir un outil d'intégration continue permettant d'avoir des tests de non-régression.

La première étape du projet était de décider des outils existants que j'allais utiliser, après des recherches j'ai décidé d'utiliser Gitlab, car il y avait beaucoup de documentation sur internet pour expliquer son fonctionnement et sur ses outils d'intégration continue.

## Installation Gitlab

J'ai donc commencer par installer et configurer les packages Gitlab sur ma WSL grâce à la documentation sur le site officiel de Gitlab (cf. sources). Il existe plusieurs version de Gitlab téléchargeable sur WSL, j'ai utilisé la version Gitlab Community Edition (CE), qui est une version gratuite et open-source. J'ai donc commencer par installer les dépendances liées à Gitlab :

```
sudo apt-get install -y ca-certificates curl openssh-server perl
```

Pour configurer Gitlab sur ma WSL, j'avais besoin de modifier la variable « EXTERNAL\_URL » et le mot de passe du root « GITLAB\_ROOT\_PASSWORD » lors de l'installation. Ce qui modifie directement les information dans le fichier gitlab.rb, qui est le fichier de configuration situé dans etc/gitlab/.

J'ai pu faire ceci avec la commande :

```
curl -LO https://packages.gitlab.com/install/repositories/gitlab/gitlab-ce/script.deb.sh
sudo bash /tmp/script.deb.sh
sudo GITLAB_ROOT_PASSWORD="fYT69ntf4vo9Jr4L" EXTERNAL_URL="http://maellesite.com" apt install gitlab-ce
```

Pour que notre solution soit utilisable en local, il fallait aller ajouter notre url dans le fichier host de notre machine en tant qu'administrateur, ainsi lorsque l'on recherche notre url sur un navigateur, notre Gitlab local s'ouvre sur la page de connexion.

Pour réaliser ceci durant l'exécution du fichier « deployment.sh », j'ai utiliser la lecture d'une réponse dans le terminal (*read*) et une condition avec un test (*if test*), lorsque l'utilisateur à réaliser ce qui était demandé, il renvoie la réponse demandée et le script continue.

Il fallait ensuite lancer Gitlab, pour ceci il fallait lancer une commande puis l'arrêter, pour ceci j'ai utilisé « nohup » pour exécuter la commande, puis je l'ai laissé durée 10 secondes avant de l'arrêter, et de lancer gitlab avec *start*.

```
nohup sudo /opt/gitlab/embedded/bin/runsvdir-start &  
pid=$!  
sleep 10  
kill -9 $pid  
sudo gitlab-ctl start
```

Pour vérifier son fonctionnement, la première étape est de se connecter sur la page de connexion en tant que root avec le mot de passe choisi lors de l'installation.



GitLab Community Edition

Nom d'utilisateur ou adresse de courriel principale

Mot de passe

[Mot de passe oublié ?](#)

☐ Se souvenir de moi

Connexion

[Vous n'avez pas encore de compte ? Inscrivez-vous maintenant](#)

Pour pouvoir effectuer mes tests plus tard, j'ai ensuite importer le projet python mis à disposition avec les fichiers de tests.

## Outil d'intégration continue

Pour déployer un outil d'intégration continue permettant de faire des tests de non-régression, un des outils proposé par Gitlab est l'utilisation de runner<sup>1</sup> qui exécute les étapes du pipeline<sup>2</sup>.

Il existe deux types de runners, un runner partagé qui est utilisé par plusieurs projets ou un runner spécifique à un projet. Pour la réalisation de ce projet j'ai décidé d'utiliser un runner partagé pour me simplifier la tâche vu que le runner n'aura qu'un seul projet sur le Gitlab, et n'aura que des tâches simple à effectuer.

J'ai donc installer les packages Gitlab Runner sur ma WSL avec :

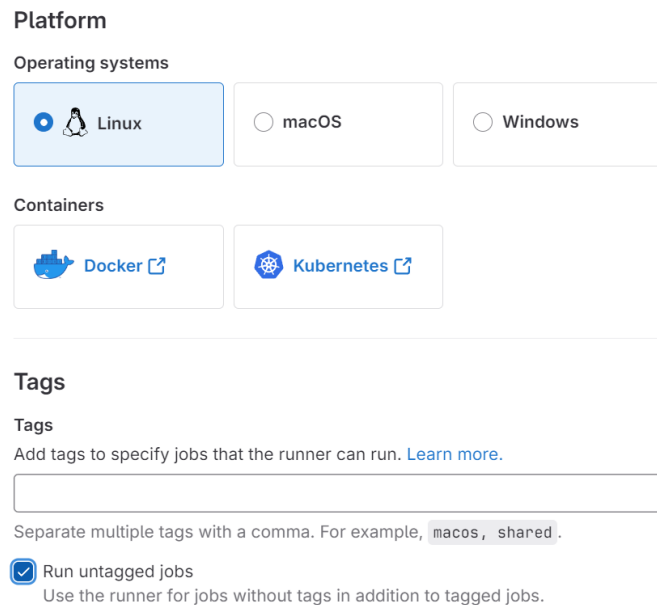
```
curl -L https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.deb.sh | sudo bash  
sudo apt-get install gitlab-runner
```

---

<sup>1</sup> Runner : sont des agents logiciels.

<sup>2</sup> Pipeline : ensemble des étapes automatisées qui décrivent le processus de test.

Pour créer un runner, j'ai suivi des documentations que j'ai trouvé sur internet (cf. sources), à partir de notre version Gitlab en local j'ai été dans la partie administrateur et ai créé un runner dans la rubrique CI/CD. Pour que ce runner soit un runner partagé j'ai utilisé l'option « run untagged jobs ».



**Platform**

Operating systems

☒ Linux ☐ macOS ☐ Windows

Containers

☒ Docker ☐ Kubernetes

**Tags**

Tags

Add tags to specify jobs that the runner can run. [Learn more.](#)

Separate multiple tags with a comma. For example, `macos, shared`.

☒ Run untagged jobs  
Use the runner for jobs without tags in addition to tagged jobs.

Lorsque le runner est créé sur le gitlab, il faut ensuite le lier à notre WSL, pour ceci j'ai utilisé la commande suivante, qui demande à l'utilisateur le token du runner créé, le token est à récupérer directement sur le Gitlab ; et en précisant directement l'url du site et l'exécuteur à utiliser.

```
echo "Après avoir créé un runner sur Gitlab (suivre le readme), entrer la valeur du token :"  
read reponse2  
sudo gitlab-runner register --url http://maellesite.com --token $reponse2 --executor shell
```

Nous pouvons maintenant lancer Gitlab Runner avec :

```
sudo gitlab-runner start
```

Il fallait ensuite indiquer à ce runner le travail à faire, ceci est fait à travers le fichier « .gitlab-ci.yml », qui définit la structure du pipeline et détermine les tâches à effectuer par les runners. Le pipeline lui est automatiquement déclenché lorsqu'un commit est effectué, j'ai donc créé ce fichier à la racine du projet.

J'ai commencé par lui indiquer l'image Docker à utiliser lors de l'exécution des jobs du pipeline, ici l'image Docker officielle de Python avec la version 3.9. Je lui ai ensuite indiqué les différentes étapes avec la clé « stage ». Puis, pour l'étape test qui exécute les tests unitaires, j'ai utilisé unittest<sup>3</sup> sur le fichier « test\_gaussian\_sum\_formula.py ».

---

<sup>3</sup> Unittest est un framework de test unitaire intégré à Python.

```

image: python:3.9













stages:
- lint
- test

unittest:
  stage: test
  script:
    - python3 -m unittest test_gaussian_sum_formula.py

```

Pour vérifier le bon fonctionnement du runner, j'ai dans un premier temps tester les fichiers sans modifications, puis j'ai modifié le fichier « gaussian\_sum\_formula.py », où le runner a bien échoué lors de sa vérification. Après avoir remodifié le fichier correctement, le runner a bien réussi sa vérification, ce qui confirme le bon fonctionnement de celui-ci.

On peut ainsi observer le retour du runner :

Status	Pipeline	3ème vérification : avec erreur rétablie	Created by	Stages
<div>✔ Passed</div> <div>🕒 00:00:02</div> <div>📅 just now</div>	<div><a href="#">3ème vérification : avec erreur rétablie...</a></div> <div>#54  main  49ac720b </div> <div>latest</div>		<div>✔</div> <div>✔</div>	
<div>✖ Failed</div> <div>🕒 00:00:02</div> <div>📅 just now</div>	<div><a href="#">2ème vérification : avec erreur</a></div> <div>#53  main  5b3b16cf </div>		<div>✔</div> <div>✖</div>	
<div>✔ Passed</div> <div>🕒 00:00:03</div> <div>📅 1 minute ago</div>	<div><a href="#">1ère vérification</a></div> <div>#52  main  427fc988 </div>		<div>✔</div> <div>✔</div>	

## Amélioration : le linting

Le linting consiste à examiner le code source pour y déceler des erreurs de syntaxe, de programmation, des vulnérabilités potentielles...

Pour ce test, j'ai choisi d'utiliser l'outil Pylint, et l'ai installé sur ma WSL, avec les commandes suivantes :

```

sudo apt-get install python3
sudo apt-get install python3-pip
pip install pylint

```

J'ai ainsi rajouter au fichier .gitlab-ci.yml une section pour utiliser l'outil pylint sur le fichier python « gaussian\_sum\_formula.py » du projet :

```
lint:
  stage: lint
  allow_failure: true
  script:
    - /home/isen/.local/bin/pylint gaussian_sum_formula.py
```

Dans cette étape, j'ai rajouté la ligne « allow\_failure : true », pour que même si l'étape du linting échoue, le runner fasse le test unitaire qui suit sur le fichier python « test\_gaussian\_sum\_formula.py », ce qui me permettait de pouvoir vérifier la globalité des étapes faites par le runner.

J'ai ensuite commit le fichier pour vérifier le bon fonctionnement de l'étape lint avec le fichier.

## Sources principales

<https://about.gitlab.com/fr-fr/install/#debian>

[https://docs.gitlab.com/ee/install/next\\_steps.html](https://docs.gitlab.com/ee/install/next_steps.html)

<https://docs.gitlab.com/runner/>

<https://guillaumebriday.fr/installer-et-utiliser-les-gitlab-runners>

<https://docs.gitlab.com/runner/register/index.html>

<https://www.younup.fr/blog/demystifier-gitlab-ci-cd>

<https://docs.gitlab.com/ee/ci/yaml/>

<https://pypi.org/project/pylint/>