École Mohammadia d'Ingénieurs

Génie Logiciel / DevOps

TP n°2 : Création et analyse d'un pipeline CI/CD multi-technologies

---

Atelier DevOps – Pipeline CI/CD multi-projets GitLab & Jenkins

---

Élèves :
Sami FAOUZI

Encadré par :
Asmaa Retbi

2ème année Génie Informatique

24 novembre 2025

Table des matières

1 Introduction et objectifs du TP

Ce TP fait suite à la mise en place de l'environnement DevOps (TP précédent) et vise à **passer à l'automatisation** avec un pipeline CI/CD multi-projets.

Les objectifs principaux sont les suivants :

— créer un **groupe GitLab** et un **dépôt** contenant trois projets : Spring Boot, Node.js et .NET ;

— mettre en place un **pipeline GitLab CI/CD** capable de compiler et tester ces trois projets dans des jobs séparés ;

— utiliser **Docker** comme environnement d'exécution standard pour les jobs GitLab ;

— reproduire la même logique dans **Jenkins**, via trois jobs de type *Pipeline* et un *pipeline script* en Groovy ;

— analyser les **jobs exécutés**, les logs et les vues graphiques (*Stage View*) ;

— comparer le fonctionnement de **GitLab CI/CD** et **Jenkins**.

2 Partie GitLab CI/CD : mise en place du pipeline multi-projets

2.1 Création du groupe et du projet GitLab

Dans GitLab, un groupe a été créé pour centraliser les travaux du TP :

— création d'un **groupe** dédié au TP CI/CD ;

— création du projet `tp-ci-cd-multi-projets-faouzi` à l'intérieur de ce groupe.

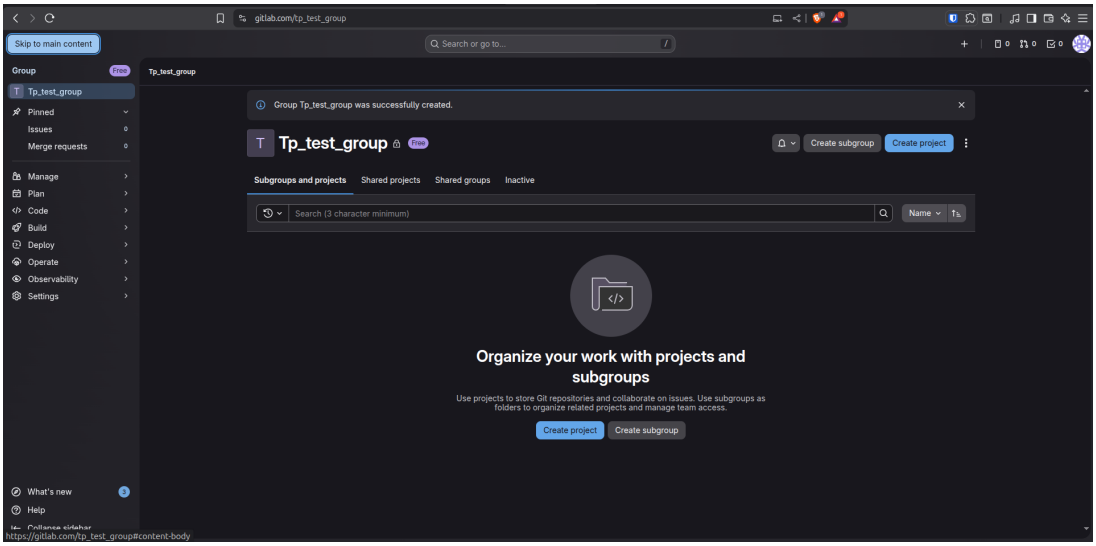

FIGURE 1 – Création du groupe GitLab dédié au TP CI/CD.

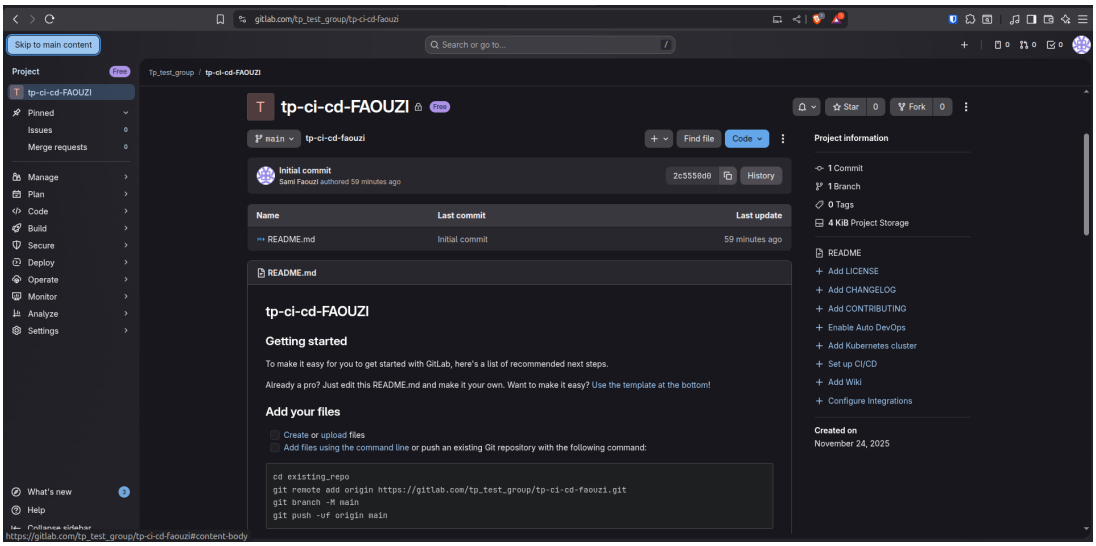

FIGURE 2 – Création du dépôt GitLab `tp-ci-cd-multi-projets-faouzi`.

2.2 Initialisation du dépôt local et première synchronisation

En local, un dossier de travail a été initialisé puis synchronisé avec le projet GitLab :

— `git init` pour initialiser le dépôt local ;

— ajout des dossiers `springboot-app/`, `nodejs-app/`, `dotnet-app/` ;

— configuration de la *remote* GitLab et premier `git push`.

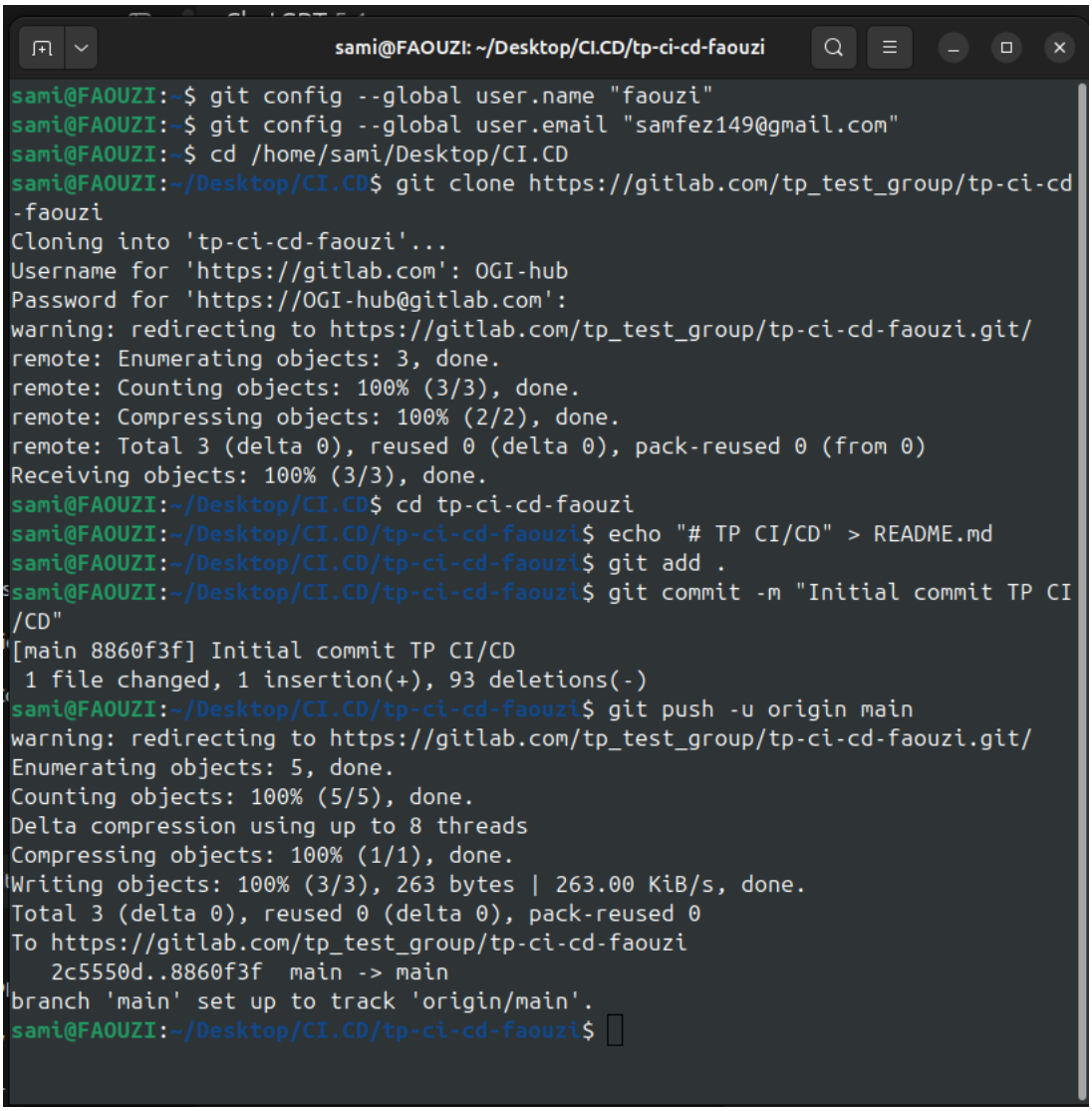

FIGURE 3 – Initialisation du dépôt local et première synchronisation avec GitLab.

2.3 Structure multi-projets dans le dépôt

Le dépôt GitLab contient trois projets applicatifs distincts, chacun avec ses propres fichiers de configuration (`pom.xml`, `package.json`, `.csproj`).

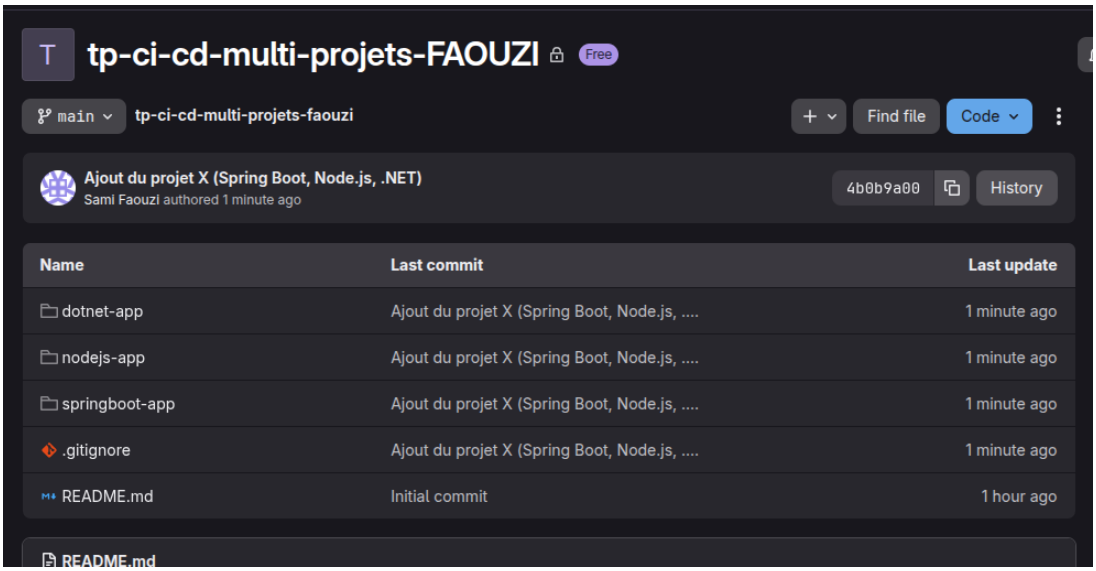

FIGURE 4 – Arborescence GitLab montrant les trois projets (Spring Boot, Node.js, .NET).

2.4 Pipeline minimal (`.gitlab-ci.yml`)

La première étape a consisté à définir un **pipeline minimal** avec un seul job, afin de valider le fonctionnement de GitLab CI/CD.

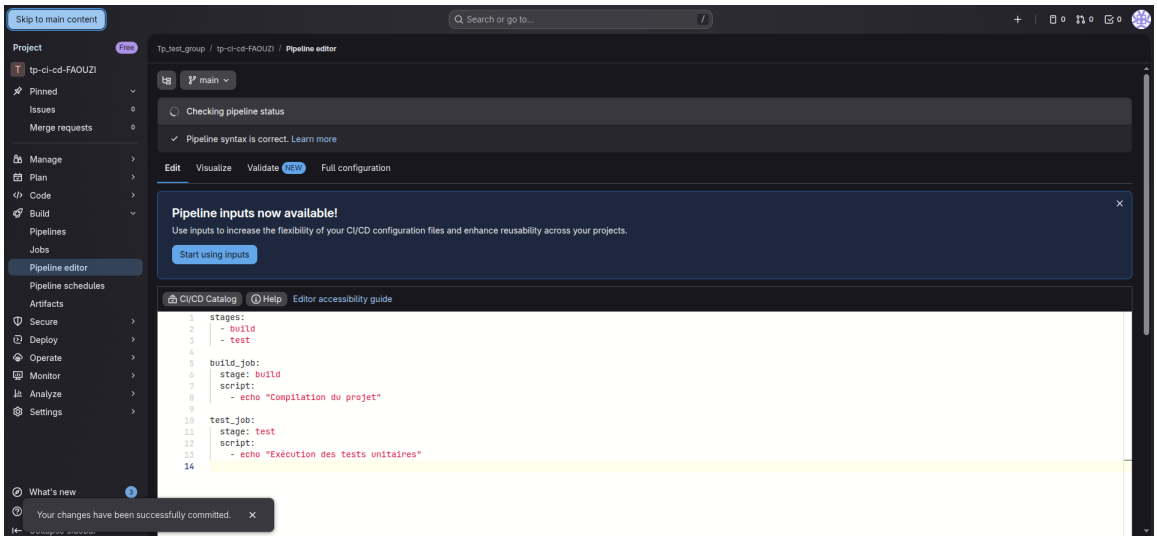

FIGURE 5 – Édition du fichier `.gitlab-ci.yml` dans le Pipeline Editor de GitLab.

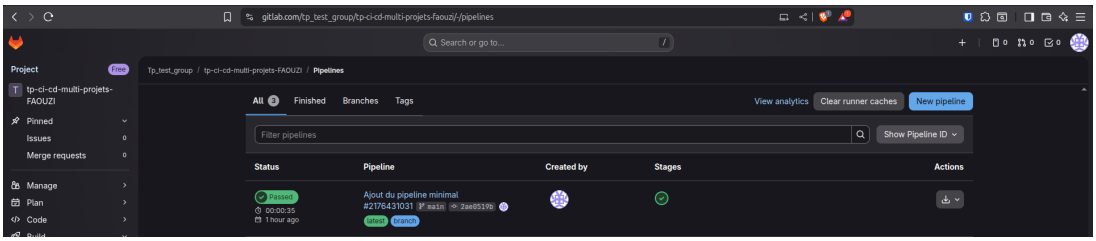

FIGURE 6 – Premier pipeline minimal GitLab CI/CD (un seul job).

2.5 Pipeline multi-projets : build et tests

Le fichier `.gitlab-ci.yml` a ensuite été enrichi pour gérer les trois projets avec les stages `build` et `test`. Le pipeline final est structuré comme suit :

```
stages:
  - build
  - test

# ---------- JOBS DE BUILD ----------
build-springboot:
  stage: build
  image: maven:3.9.9-eclipse-temurin-21-alpine
  script:
    - echo "===== Début du build Spring Boot ====="
    - cd springboot-app
    - mvn -B clean package -DskipTests
    - echo "===== Fin du build Spring Boot ====="

build-nodejs:
  stage: build
  image: node:20-alpine
  script:
    - echo "===== Début du build Node.js ====="
    - cd nodejs-app
    - npm install
    - npm run build
    - echo "===== Fin du build Node.js ====="

build-dotnet:
  stage: build
  image: mcr.microsoft.com/dotnet/sdk:8.0
  script:
    - echo "===== Début du build .NET ====="
    - cd dotnet-app/demoapp
    - dotnet restore
    - dotnet build --configuration Release
    - echo "===== Fin du build .NET ====="

# ---------- JOBS DE TEST ----------
```

```
test-springboot:
  stage: test
  image: maven:3.9.9-eclipse-temurin-21-alpine
  script:
    - echo "----- Début des tests Spring Boot -----"
    - cd springboot-app
    - mvn test
    - echo "----- Fin des tests Spring Boot -----"

test-nodejs:
  stage: test
  image: node:20-alpine
  script:
    - echo "----- Début des tests Node.js -----"
    - cd nodejs-app
    - npm test || echo "Pas de tests définis"

test-dotnet:
  stage: test
  image: mcr.microsoft.com/dotnet/sdk:8.0
  script:
    - echo "----- Début des tests .NET -----"
    - cd dotnet-app/demoapp
    - dotnet test
    - echo "----- Fin des tests .NET -----"
```

2.6 Validation du pipeline multi-jobs dans GitLab

Une fois le fichier `.gitlab-ci.yml` commité et poussé, GitLab déclenche automatiquement le pipeline. Les écrans suivants montrent la **validation du pipeline** multi-projets.

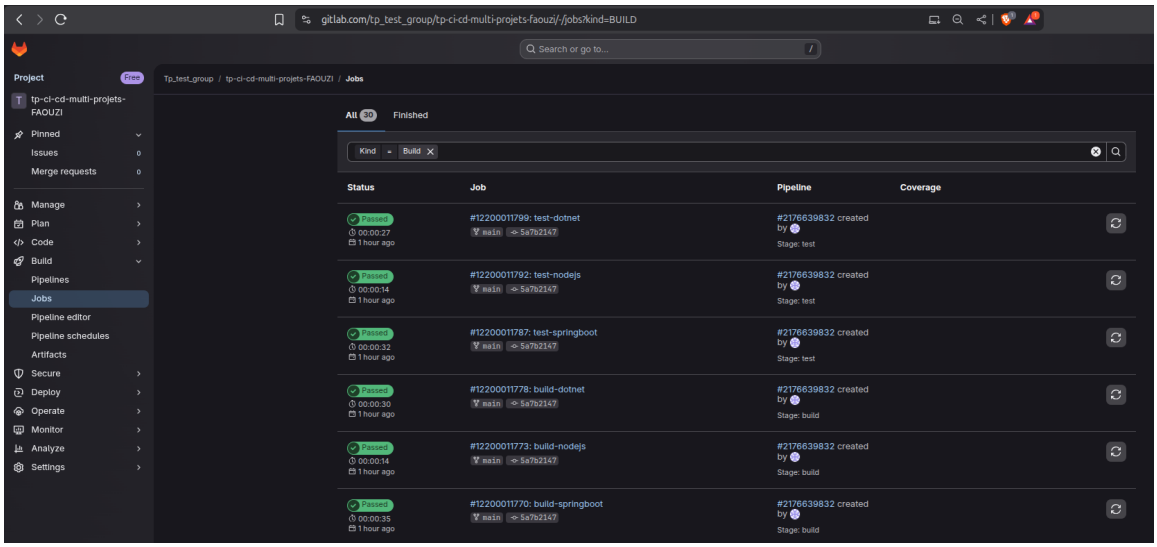

FIGURE 7 – Pipeline GitLab multi-projets avec les jobs de build et de test validés.

3 Intégration de Docker dans le pipeline GitLab

3.1 Images Docker pour les jobs GitLab

GitLab exécute chaque job dans un **conteneur Docker** défini par la clé `image` dans le fichier `.gitlab-ci.yml`. Dans ce TP, les images suivantes ont été utilisées :

- `maven:3.9.9-eclipse-temurin-21-alpine` pour les jobs Spring Boot ;
- `node:20-alpine` pour les jobs Node.js ;
- `mcr.microsoft.com/dotnet/sdk:8.0` pour les jobs .NET.

L'image permet d'embarquer toutes les dépendances (JDK, Maven, Node, SDK .NET) et garantit que le pipeline sera **reproductible** quel que soit le runner.

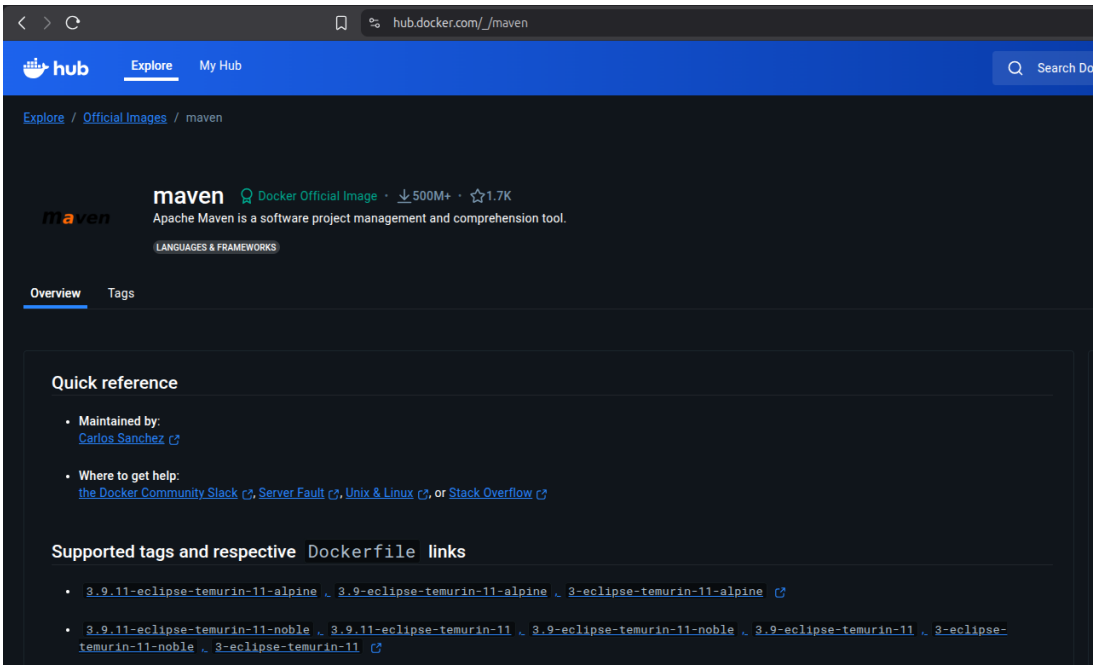

FIGURE 8 – Exemple d'image Docker Maven utilisée dans le pipeline GitLab.

3.2 Vérification locale de l'installation de Docker

Avant d'utiliser des images dans GitLab, il est nécessaire de vérifier que Docker est correctement installé sur la machine de travail :

— `docker -version` : vérifie la version du moteur Docker ;

— `docker run hello-world` : teste l'exécution d'un conteneur simple.

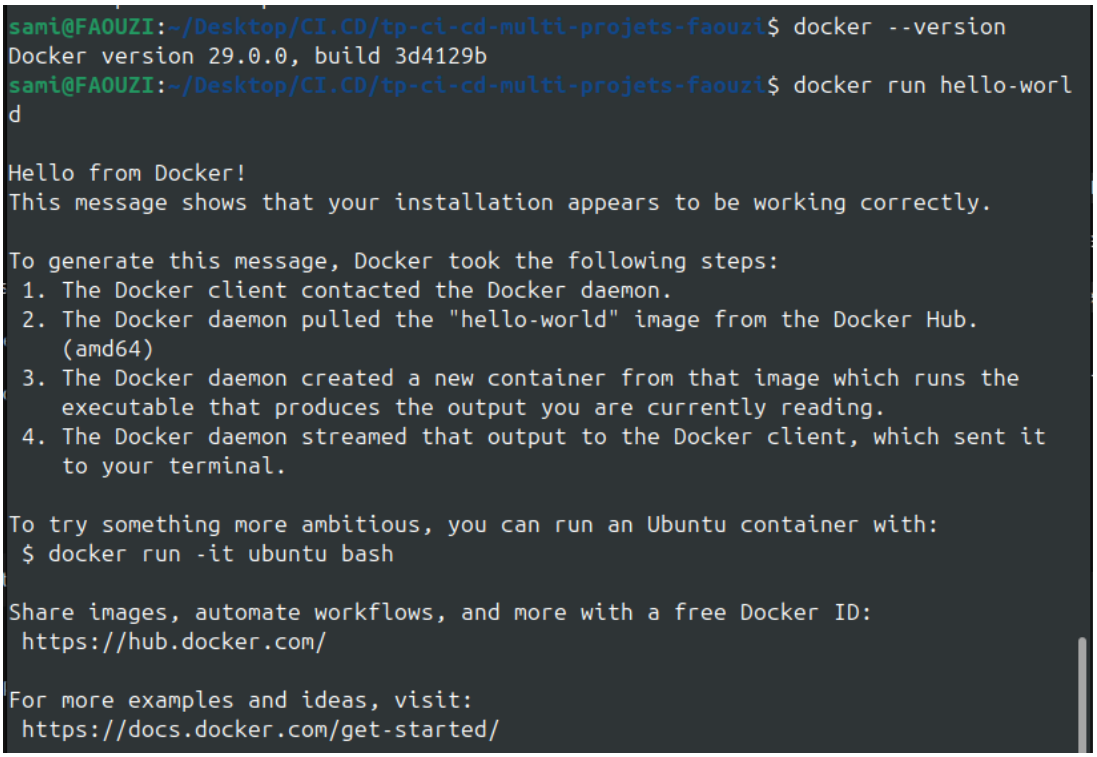

FIGURE 9 – Vérification de l'installation de Docker : `docker -version` et `docker run hello-world`.

Cette étape confirme que la machine est prête pour exécuter des jobs dans des conteneurs, que ce soit via GitLab CI/CD ou plus tard avec Jenkins.

4 Partie Jenkins : création des pipelines Spring Boot, Node.js et .NET

La suite du TP consiste à reproduire la logique du pipeline GitLab dans **Jenkins**, sous forme de trois *Pipeline jobs* : `springboot-pipeline`, `nodejs-pipeline`, `dotnet-pipeline`.

4.1 Liste des jobs Jenkins

Depuis la page d'accueil Jenkins, trois jobs de type *Pipeline* ont été créés (un par technologie).

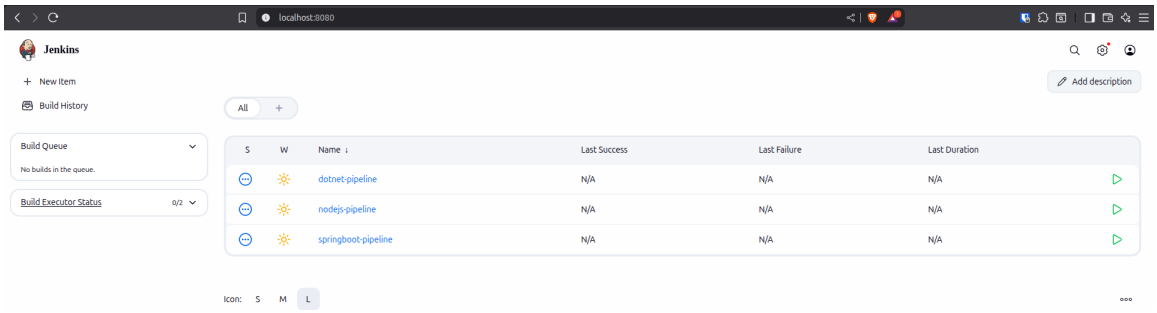

FIGURE 10 – Vue globale Jenkins : trois pipelines (Spring Boot, Node.js, .NET).

4.2 Structure générale du *pipeline script* Jenkins

Pour chaque job, la section **Pipeline** de la configuration contient un script Groovy de la forme suivante :

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                echo '=== Compilation du projet ==='
                dir('/chemin/vers/le/projet') {
                    sh 'commande de build'
                }
            }
        }

        stage('Test') {
            steps {
                echo '=== Exécution des tests ==='
                dir('/chemin/vers/le/projet') {
                    sh 'commande de tests'
                }
            }
        }
    }
}
```

La même structure est réutilisée pour les trois technologies, en adaptant uniquement le chemin et les commandes (`mvn`, `npm`, `dotnet`).

4.3 Exemple : pipeline Jenkins pour le projet .NET

```
pipeline {
    agent any

    stages {
        stage('Build') {
            steps {
                echo '=== Compilation du projet .NET ==='
                dir('/home/sami/Desktop/CI.CD/tp-ci-cd-multi-projets-faouzi/dotnet-app
                    sh 'dotnet restore'
                    sh 'dotnet build --configuration Release'
                }
            }
        }

        stage('Test') {
            steps {
                echo '=== Exécution des tests .NET ==='
                dir('/home/sami/Desktop/CI.CD/tp-ci-cd-multi-projets-faouzi/dotnet-app
                    sh 'dotnet test'
                }
            }
        }
    }
}
```

5 Observation des exécutions : Stage View et Console Output

Cette section correspond aux consignes de la diapo 10 : « Analyse des jobs ».

5.1 Stage View : enchaînement Build / Test

La *Stage View* permet de visualiser pour chaque job Jenkins les différents stages (`Build`, `Test`) et leur durée. Les cases vertes correspondent aux stages exécutés avec succès ; les cases rouges indiquent un échec.

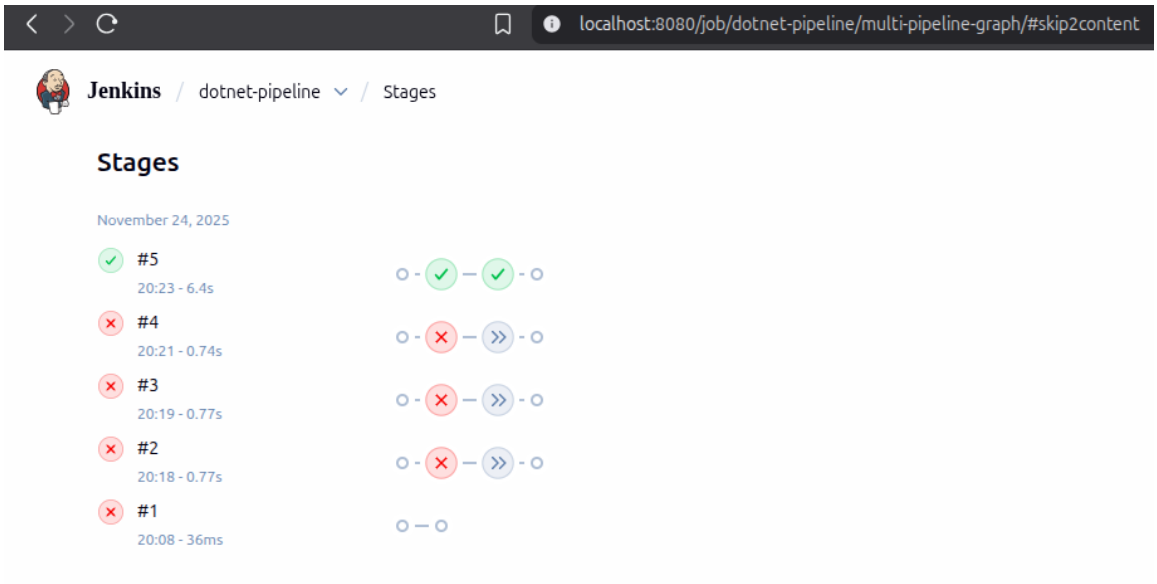

FIGURE 11 – Stage View Jenkins pour le pipeline .NET (stages `Build` et `Test`).

5.2 Console Jenkins : logs détaillés et statut

Pour chaque build, Jenkins fournit un **Console Output** qui récapitule :

— le déclencheur du job (ici, lancement manuel) ;

— les commandes exécutées (`mvn`, `npm`, `dotnet`) ;

— les messages d'erreur éventuels ;

— le statut final : **SUCCESS** ou **FAILURE**.

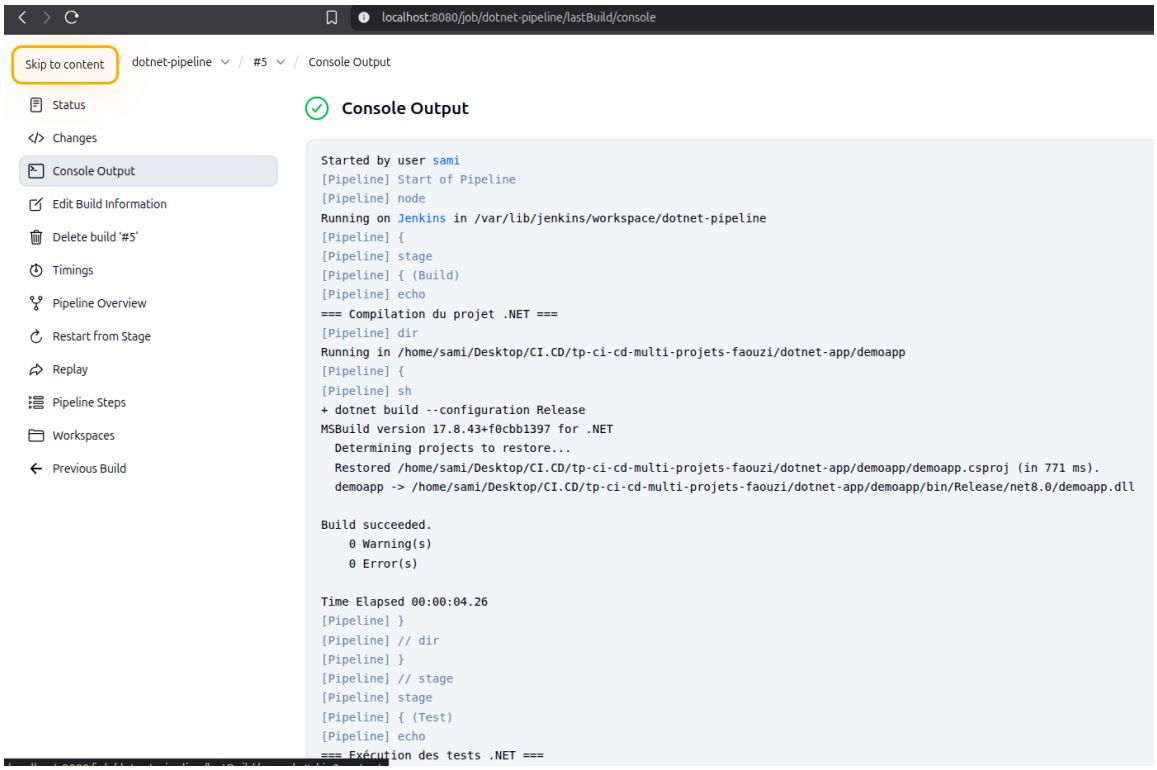

FIGURE 12 – Console Jenkins pour `dotnet-pipeline` : détail du build et des tests.

5.3 Synthèse visuelle des builds exécutés

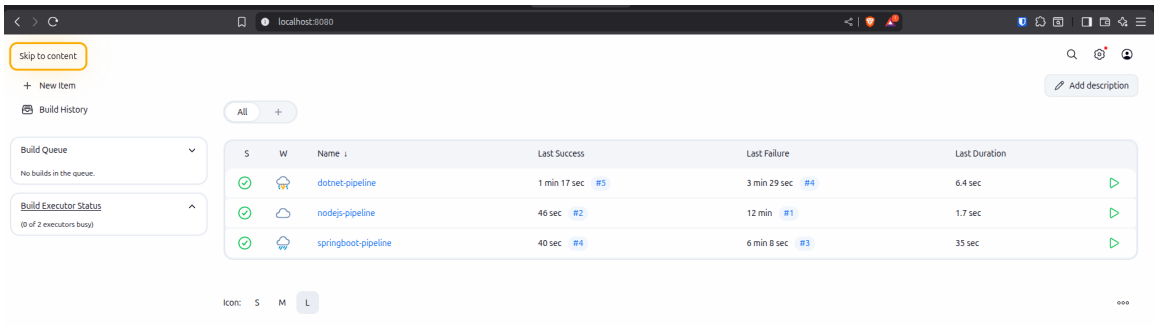

FIGURE 13 – Historique des builds exécutés pour les pipelines Jenkins.

Les captures demandées par la diapo 10 sont ainsi couvertes :

— **Vue Stage View** avec les stages `Build` et `Test` ;

— **Vue Console** montrant les logs détaillés et le statut success/failure ;

— **Vue globale des jobs** montrant les builds exécutés.

6 Comparaison GitLab CI/CD – Jenkins (diapo 11)

La dernière partie du TP demande de comparer GitLab CI/CD et Jenkins selon plusieurs critères. Le tableau suivant résume les principales observations.

Critère	GitLab CI/CD	Jenkins	Observation / Conclusion
Installation	GitLab SaaS déjà disponible en ligne. Le runner partagé est déjà configuré.	Doit être installé et maintenu par l'équipe (service Linux, plugins, mises à jour).	GitLab est plus simple à démarrer pour un TP. Jenkins offre plus de contrôle mais demande plus d'administration.
Configuration du pipeline	Pipeline défini dans le dépôt via un fichier YAML `.gitlab-ci.yml`, versionné avec le code.	Pipeline défini dans l'interface (*Pipeline script*) ou dans un `Jenkinsfile`.	GitLab favorise le « pipeline as code » directement lié au dépôt. Jenkins est très flexible grâce aux plugins.
Rapports et logs	Vue « Pipelines », détail des jobs, logs en temps réel dans le navigateur.	Stage View, Console Output très détaillé, historique des builds par job.	Les deux outils offrent des logs complets. GitLab est très intégré à Git ; Jenkins est plus orienté « usine à jobs ».
Docker et conteneurs	Utilisation directe de `image:` dans `.gitlab-ci.yml`. Les runners GitLab exécutent les jobs dans des conteneurs dédiés.	Nécessité d'installer et configurer le plugin « Docker Pipeline » ou des agents Docker.	GitLab facilite l'utilisation de conteneurs. Jenkins est plus puissant mais requiert une configuration supplémentaire.

TABLE 1 – Comparaison synthétique entre GitLab CI/CD et Jenkins.

7 Conclusion

Au terme de ce TP, les objectifs suivants ont été atteints :

- mise en place d'un **pipeline CI/CD multi-projets** dans GitLab pour trois technologies différentes (Spring Boot, Node.js, .NET) ;
- déclenchement automatique des jobs de **build** et de **test** à partir du fichier `.gitlab-ci.yml` versionné dans le dépôt ;
- utilisation d'**images Docker** pour exécuter les jobs dans des environnements maîtrisés et reproductibles ;
- création de trois **pipelines Jenkins** reproduisant la même logique (stages `Build` et `Test`) et analyse de leurs exécutions ;
- compréhension des vues principales : Pipelines GitLab, liste des jobs, Stage View Jenkins, Console Output ;
- comparaison structurée entre **GitLab CI/CD** et **Jenkins**, en lien direct avec les observations réalisées pendant le TP.

Ce travail prépare les prochains ateliers, où ces pipelines pourront être enrichis (par exemple avec des étapes d'analyse de qualité, de packaging d'artefacts ou de déploiement continu).