



**DEPARTEMENT MATHS-INFORMATIQUE**  
**SECTION INFORMATIQUE**  
**-- UCAD --**

**PROJET D'ATELIER DE**  
**GÉNIE LOGICIEL**

Réalisé par:

Ndeye Astou Thiam  
Mouhamed Amar  
Mouhamed Diop

Encadreur:

M. Ndong Samb

Année Universitaire 2019-2020

# Table des matières

I. Introduction.....	3
II. Installation des outils.....	3
II-1.Maven.....	3
II-2. Git.....	3
II-3.Docker.....	4
III .Création du projet java avec maven.....	5
IV. Écriture de tests unitaires.....	8
V. Mise à disposition du code sur Github.....	9
VI. Création d'un fichier Dockerfile .....	9
VII .Mise en place d'un CI/CD avec github actions.....,,.....	11
VIII. Lancement de tests unitaires avec github actions. ....	13
IX. Publication de l'application Java sous forme d'image sur le docker hub avec github Actions.....	14
X. Conclusion.....	18

---

## I. Introduction

---

Le développement d'un logiciel est un parcours semé d'embûches. Toutefois, certaines pratiques et méthodologies permettent de rendre l'expérience moins fastidieuse. Le génie logiciel (software engineering) représente l'application de principes d'ingénierie au domaine de la création de logiciels. Il consiste à identifier et à utiliser des méthodes, des pratiques et des outils permettant de maximiser les chances de réussite d'un projet logiciel.

---

## II. Installation des outils

---

### II-1.Maven

L'installation d'Apache Maven est un processus simple d'extraction de l'archive et d'ajout du dossier bin avec la commande `mvn` au `PATH`.

Les étapes détaillées sont :

- ✓ On s'est d'abord assuré que notre `Java_Home` variable d'environnement est définie et pointe vers notre installation JDK
- ✓ L'étape suivante consiste à télécharger le dossier zippé `apache-maven-3.3.3` et d'extraire l'archive de distribution dans le répertoire de notre choix
- ✓ Nous avons, par la suite, ajouté le répertoire `bin` du répertoire créé `apache-maven-3.3.3` à la variable d'environnement `PATH`.

Maintenant l'installation se termine ici et pour vérifier que tout s'est bien passé, on a lancé la commande `mvn -v` juste pour voir la version installée. Et le résultat obtenu est le suivant :



```
Invite de commandes
Microsoft Windows [version 10.0.18363.1198]
(c) 2019 Microsoft Corporation. Tous droits réservés.

C:\Users\PC>mvn -v
Apache Maven 3.6.3 (cecedd343002696d0abb50b32b541b8a6ba2883f)
Maven home: C:\apache-maven-3.6.3\bin\..
Java version: 1.8.0_201, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk1.8.0_201\jre
Default locale: fr_FR, platform encoding: Cp1252
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"

C:\Users\PC>
```

### II-2.Git

Git est un logiciel de gestion de version décentralisé parmi les plus populaires avec 12 000 000 d'utilisateurs dans le monde. C'est un logiciel libre créé par [Linus Torvalds](#), auteur du noyau Linux. Git est donc totalement gratuit.

Ce qu'on a fait c'est alors de télécharger simplement le fichier d'installation [.exe](#) (l'installateur) depuis la page officielle Github.



Pour installer l'outil, on a double-cliqué simplement sur l'icône du fichier déjà téléchargé. Une interface se lance alors et nous propose, page après page, de configurer l'installation de Git. Nous avons laissé les valeurs par défaut et simplement cliqué sur [suivant](#). Une fois arrivé sur la dernière page, nous avons cliqué sur [installer](#).

Nous venons alors d'installer les éléments suivants sur notre machine :

- ✓ l'outil [Git](#)
- ✓ [Git bash](#): terminal qui nous permet d'utiliser git en ligne de commande
- ✓ [git gui](#): interface graphique qui permet de gérer les commits.
- ✓ [gitk](#): interface graphique qui permet de gérer l'historique de notre dépôt

Reste maintenant à vérifier que l'installation s'est bien déroulée. Pour ce faire, nous avons utilisé une commande de base de l'outil qui est « [git version](#) » et qui permet d'afficher le numéro de version de Git.

A screenshot of a terminal window titled 'MINGW64:/c/Users/PC'. The prompt is 'PC@NDEYA MINGW64 ~'. The user has entered the command '\$ git version' and the output is 'git version 2.28.0.windows.1'. The prompt is now '\$ |'.

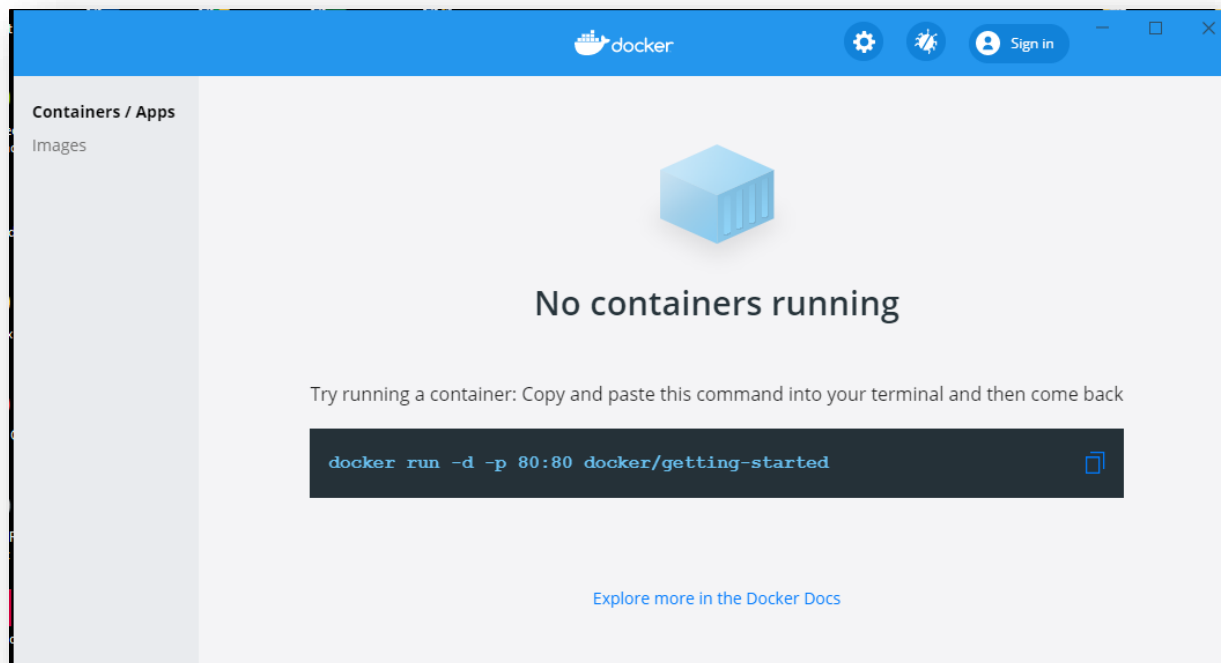
## II-3.Docker

Comme tout autre logiciel, nous avons commencé par télécharger [Docker Desktop](#) pour Windows à partir de [Docker Hub](#) et après cela, suivre ensuite ces différentes étapes :

1. Double-cliquer sur Docker Desktop Installer.exe pour exécuter le programme d'installation.
2. Activer les fonctionnalités Windows Hyper-V sélectionnées sur la page de configuration.
3. Suivre les instructions de l'assistant d'installation pour autoriser le programme d'installation et poursuivre l'installation.

4. Une fois l'installation réussie, cliquer sur [Fermer](#) pour terminer le processus d'installation.

Sur ce, nous venons d'avoir Docker installé sur notre ordinateur.



---

### III .Création du projet java avec maven

---

On part, pour cela, d'un projet Java vide, sous Eclipse.

La création d'un nouveau projet Maven se fait de façon classique, comme tout autre projet.

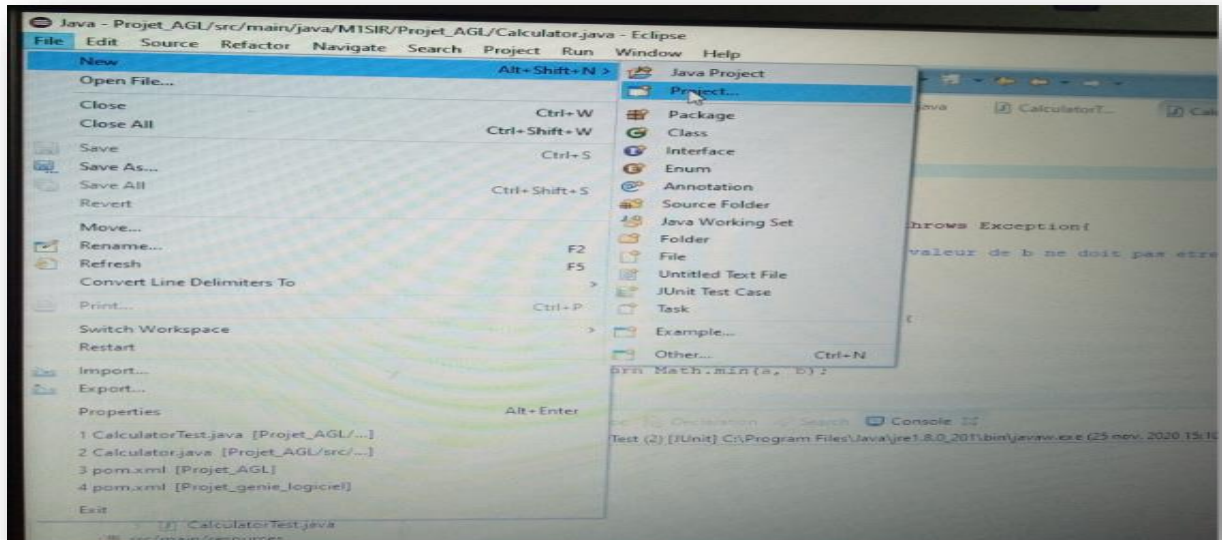


Figure 1 : Création d'un nouveau projet dans Eclipse

Le panneau de création de projet s'ouvre alors. Notons qu'il existe de nombreuses options dans ce panneau mais on s'intéresse ici à l'option **Maven**, que l'on peut filtrer en tapant Maven dans la barre de filtrage.

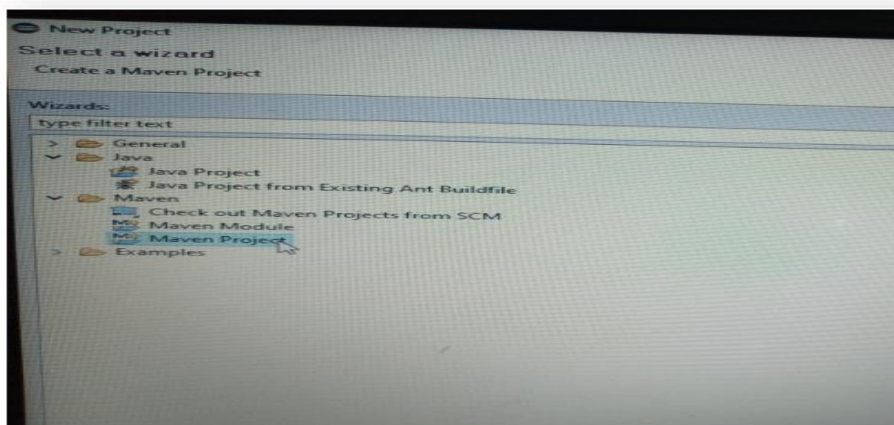


Figure 2 : Choix d'un projet Maven

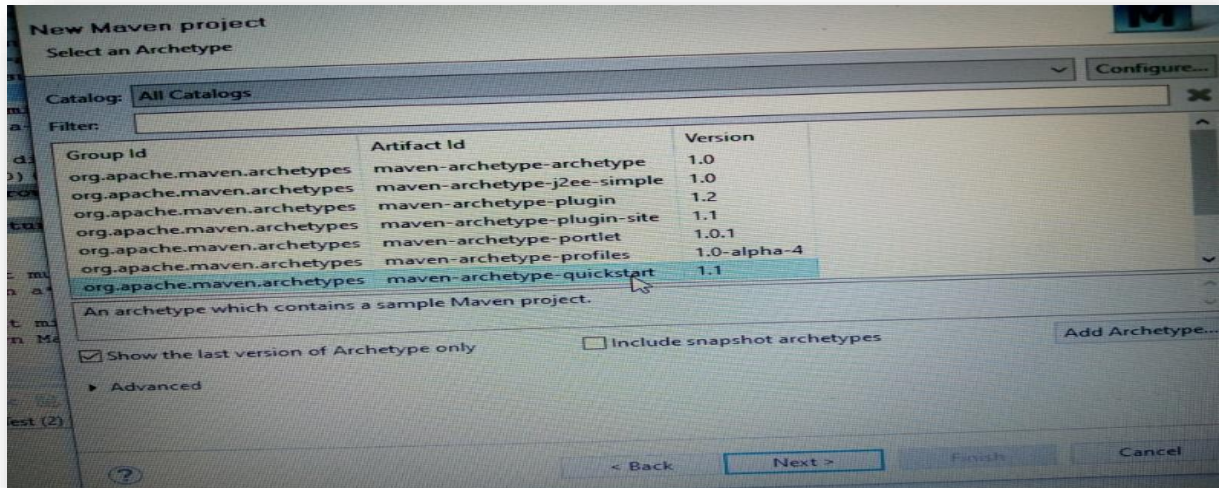


Figure 3 : Choix de l'Archétype [quickstart](#)

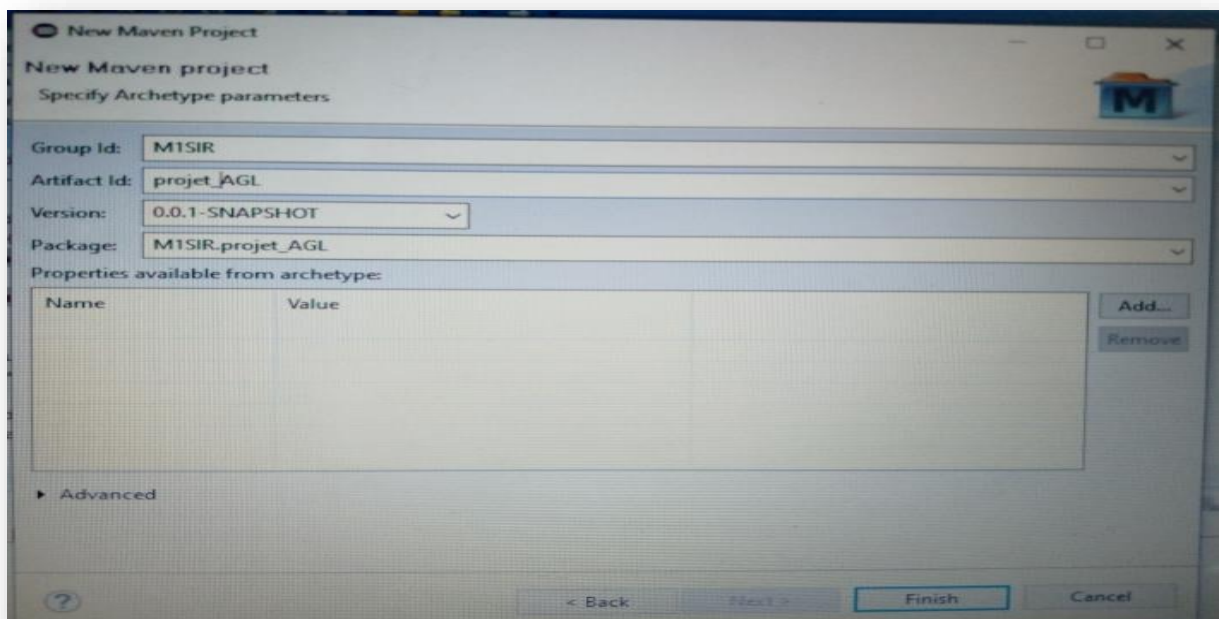


Figure 4 : Derniers paramétrages

Après ces étapes de création de projet et consorts, on a créé la classe [Calculator.java](#) dans [src/main/java](#) et [CalculatorTest.java](#) dans [src/test/java](#) qui va, par ailleurs, nous permettre d'exécuter nos tests unitaires plus tard.



---

## IV. Écriture de tests unitaires

---

Pour cette partie, c'est l'intégration de [junit](#) (qui propose un Framework pour écrire les classes de tests) dans notre fichier [pom.xml](#) du projet et aussi de [Hamcrest](#) qui nous ont permis, de par l'annotation, `@test` à pouvoir faire des tests unitaires sur les différentes méthodes.

```
9      <!-- https://mvnrepository.com/artifact/org.hamcrest/hamcrest-core -->
10 <dependency>
11     <groupId>org.hamcrest</groupId>
12     <artifactId>hamcrest-core</artifactId>
13     <version>1.3</version>
14     <scope>test</scope>
15 </dependency>
16 <!-- https://mvnrepository.com/artifact/junit/junit -->
17 <dependency>
18     <groupId>junit</groupId>
19     <artifactId>junit</artifactId>
20     <version>4.13</version>
21     <scope>test</scope>
22 </dependency>
23
```

Chaque cas de tests fait l'objet d'une méthode dans la classe de `CalculatorTest.java`. Chacune de ces méthodes contient généralement des traitements en trois étapes :

- ✓ Instanciation des objets requis
- ✓ Invocation des traitements sur les objets
- ✓ Vérification des résultats des traitements

En guise d'exemple :

```
@Before
public void initialise() throws Exception {
    // Instanciation des objets requis
    calculator = new Calculator ();
}

@Test
public void testSum () {
    // Invocation des traitements sur les objets
    int result = calculator.sum(3, 2);
    // Vérification des résultats des traitements
    if (result != 5) {
        Assert.fail();
    }
}
```



---

## V. Mise à disposition du code sur Github

---

Pour ne pas créer des conflits ou bien encore pour bien s'organiser, notre équipe a d'abord commencé par créer des branches sur le compte principal Github d'un d'entre nous .En sous-entendant bien évidemment qu'un étudiant a utilisé son compte et a ajouté les autres membres comme collaborateurs pour qu'ils puissent avoir les droits d'accès et tout .

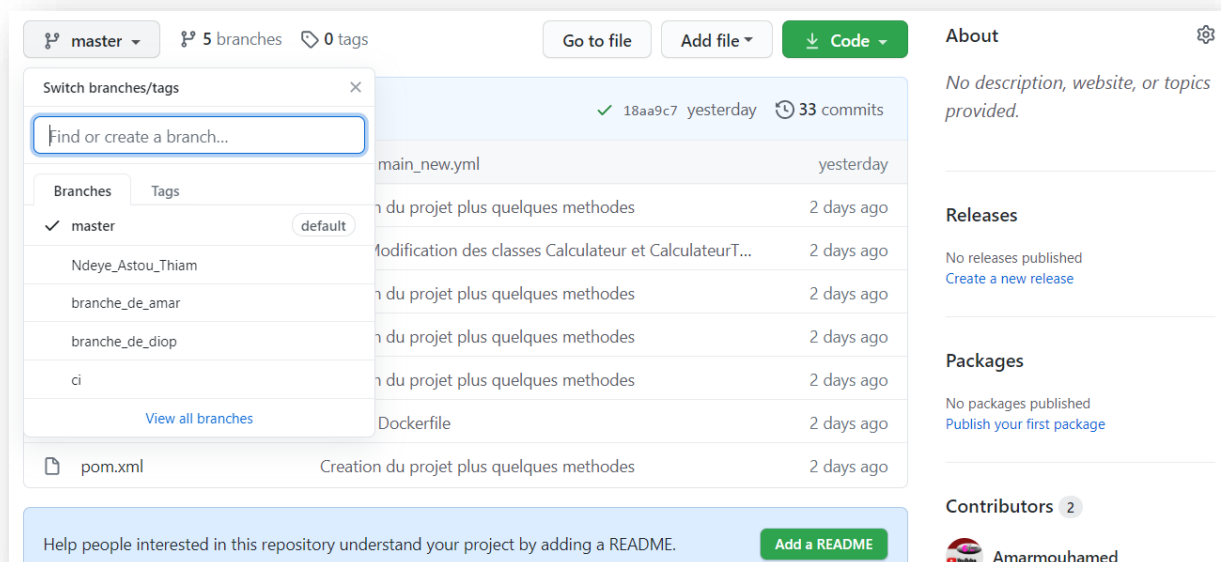
Du coup, chaque étudiant a créé sa propre branche .Et bien sûr ,on n'a pas d'un seul coup ajouté tout le projet mais on a d'abord commencé par faire une partie : c'est-à-dire la création du projet plus quelques méthodes .Donc le premier étudiant a d'abord poussé cette première partie du projet dans sa branche ([branche\\_ndeye\\_astou](#)) et après vérification, ce premier étudiant [merge](#) ce qu'il a eu à ajouter dans la branche principale à savoir la branche [master](#) .

En deuxième lieu, les deux autres étudiants ont cloné ce projet avec la commande : `git clone https://github.com/missndeya/AGL.git` .

Après cette étape, chacun d'entre eux ajoute quelques méthodes dans les classes [Calculator.java](#) et [CalculatorTest.java](#).

Comme pour la première personne aussi, ces deux derniers ajoutent d'abord leurs travaux dans leurs branches respectives avant de merger, à leur tour dans la branche master.

C'est donc comment on a procédé tout au long de ce projet pour ajouter ou modifier des ressources en son sein.



---

## VI. Création d'un fichier Dockerfile

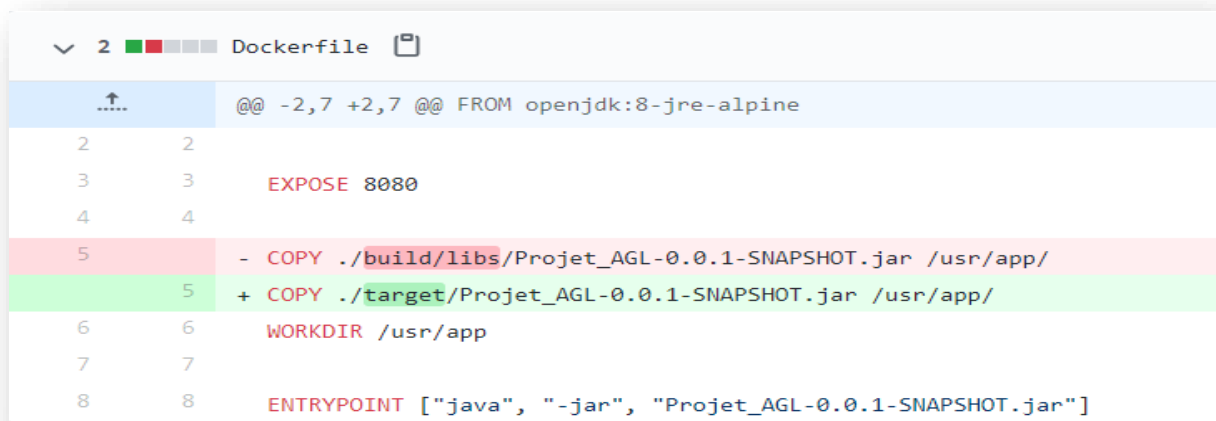
---

Un Dockerfile sert à construire d'emblée l'image Docker qu'il nous faut, couche par couche et de façon logique et organisée. Un Dockerfile est un fichier texte décrivant les différentes étapes permettant de partir d'une base pour aller vers une application fonctionnelle. Docker lit les instructions que l'on met dans le Dockerfile pour créer automatiquement l'image requise.

Avant la création de ce fichier, on a d'abord généré le [jar](#) de notre projet avec la commande [mvn package](#) vu que maven est déjà installé sur notre ordinateur. Par la suite, c'est ce jar là qu'on renseignera plus tard dans ledit fichier.

Pour commencer, on a créé un fichier texte que nous avons nommé «Dockerfile» sans extension.

La seconde étape consiste, ensuite, à éditer ce fichier avec la commande [nano Dockerfile](#) dans [git bash](#) pour y mettre des instructions et le nom du jar de notre projet généré.



```
2 2 FROM openjdk:8-jre-alpine
3 3 EXPOSE 8080
4 4
5 5 - COPY ./build/libs/Projet_AGL-0.0.1-SNAPSHOT.jar /usr/app/
6 6 + COPY ./target/Projet_AGL-0.0.1-SNAPSHOT.jar /usr/app/
7 7 WORKDIR /usr/app
8 8 ENTRYPOINT ["java", "-jar", "Projet_AGL-0.0.1-SNAPSHOT.jar"]
```

---

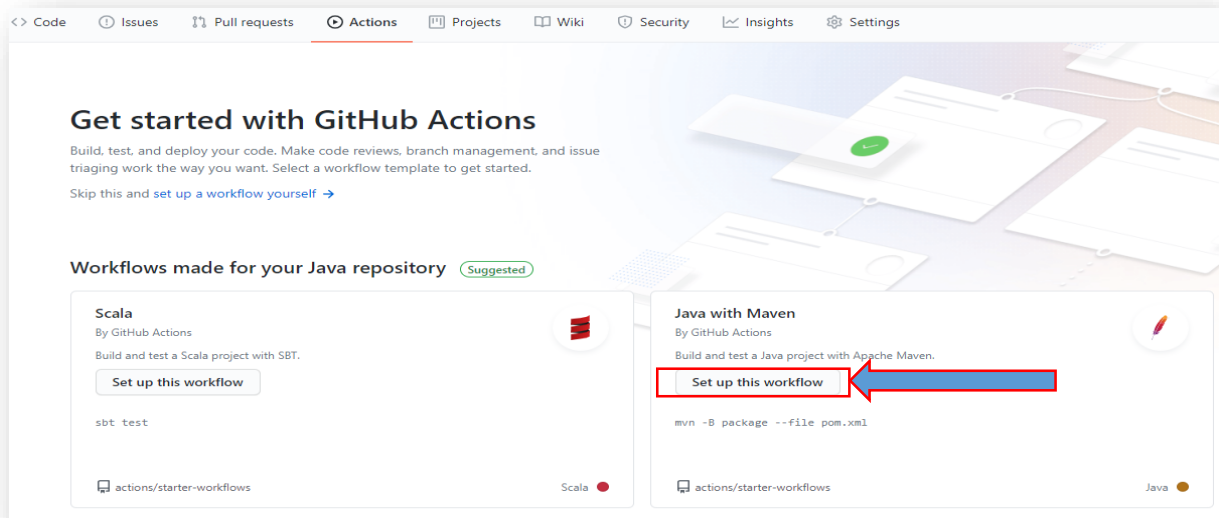
## VII .Mise en place d'un CI/CD avec github actions

---

[GitHub Actions](#) reprend le concept d'intégration continue qui nécessitait jusque-là un service tiers, mais en l'intégrant à ses serveurs et en lui apportant une interface graphique, plus simple d'accès. Cette fonction peut servir à de nombreuses choses, mais le plus simple est le déploiement automatique d'un projet, souvent après une série de tests.

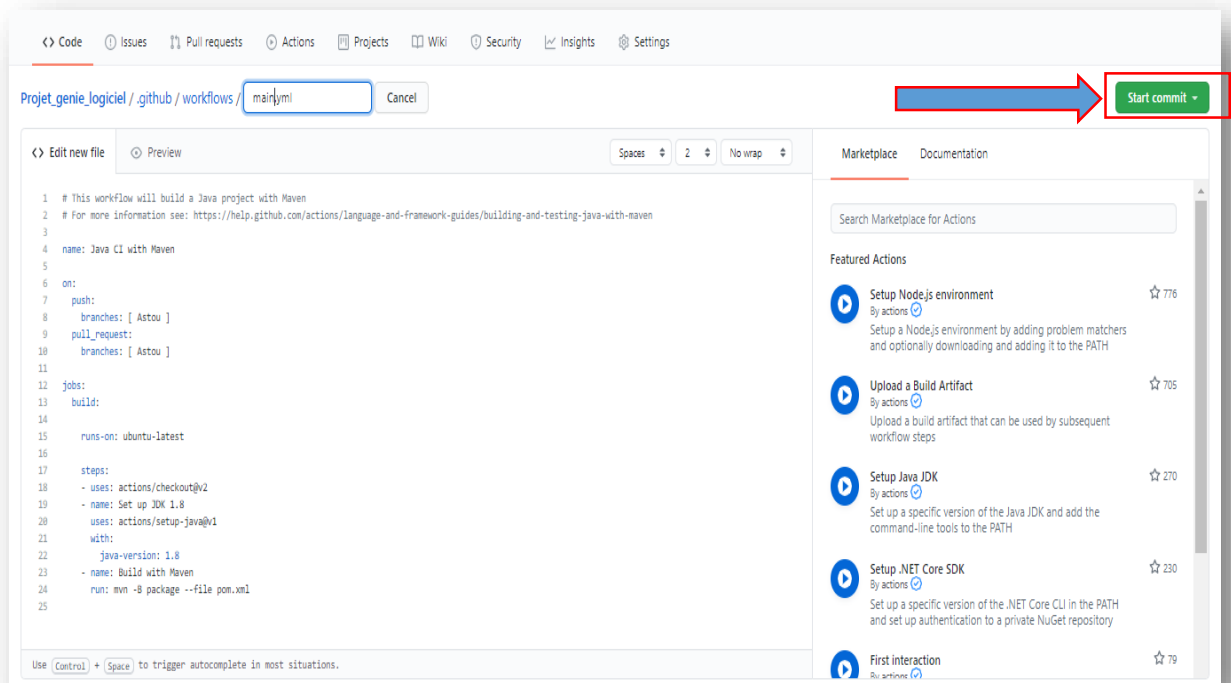
🔧 Processus de la mise en place de l'outil d'intégration continue

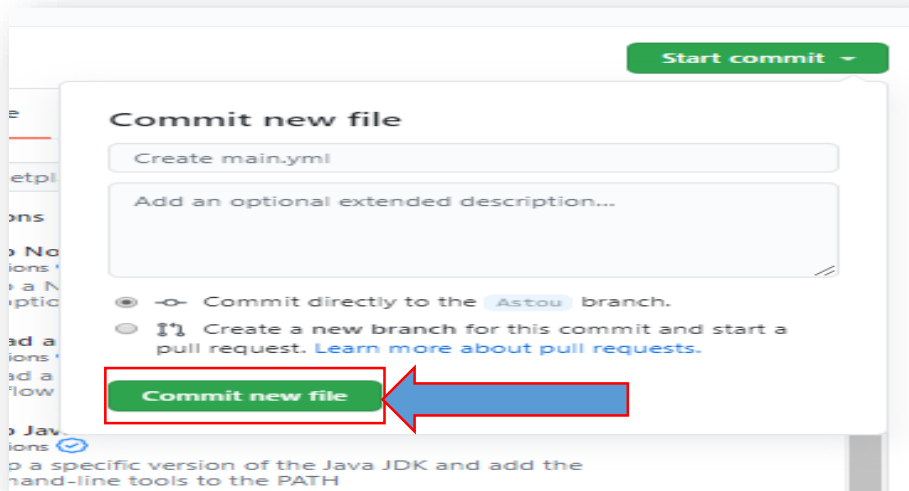
GitHub nous propose une API [GitHub Actions](#) qui nous permet d'orchestrer n'importe quel outil d'intégration continue qui est, en d'autres termes, un flux de travail (workflow). Et naturellement, comme on a travaillé avec un projet maven, on a choisi le workflow [java with maven](#).



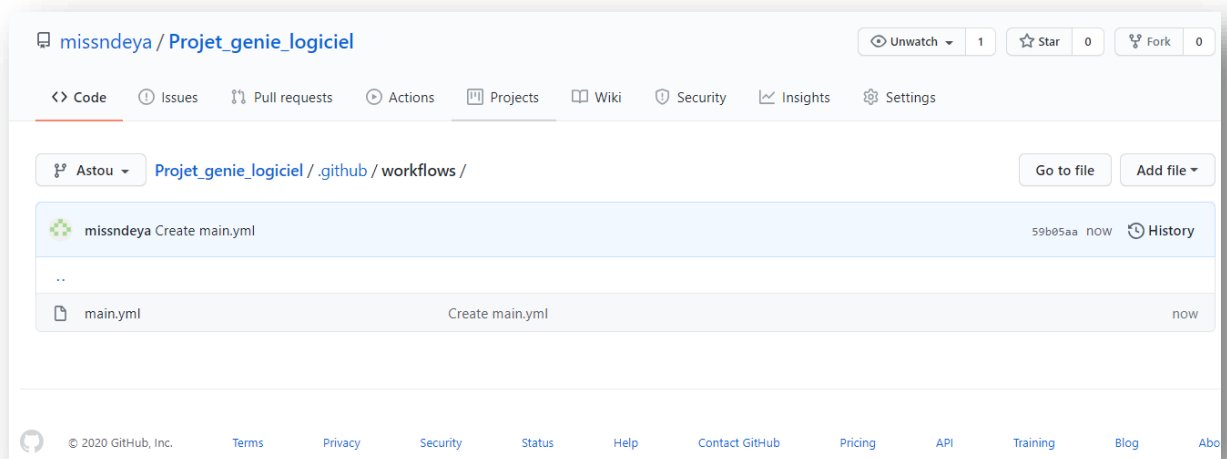
Avec GitHub Actions, les flux de travail et les étapes ne sont que du code dans un référentiel, on peut alors créer, partager, réutiliser et adapter nos pratiques de développement logiciel.

ET ci-dessus on a les différentes étapes pour finaliser la mise en place de l'outil d'intégration continue :





Et au final, on a obtenu un fichier [main.yml](#) dont le contenu est le suivant :



```
24 .github/workflows/main.yml
... @@ -0,0 +1,24 @@
1 + # This workflow will build a Java project with Maven
2 + # For more information see: https://help.github.com/actions/language-and-framework-guides/building-and-testing-java-with-maven
3 +
4 + name: Java CI with Maven
5 +
6 + on:
7 +   push:
8 +     branches: [ Astou ]
9 +   pull_request:
10 +     branches: [ Astou ]
11 +
12 + jobs:
13 +   build:
14 +
15 +     runs-on: ubuntu-latest
16 +
17 +     steps:
18 +       - uses: actions/checkout@v2
19 +       - name: Set up JDK 1.8
20 +         uses: actions/setup-java@v1
21 +         with:
22 +           java-version: 1.8
23 +       - name: Build with Maven
24 +         run: mvn -B package --file pom.xml
```

Ce workflow exécute les étapes suivantes:

1. L'étape **checkout** télécharge une copie de notre référentiel sur le **runner**
2. L'étape **setup-java** configure le JDK Java 1.8
3. L'étape **build with Maven** exécute le `package` cible en mode non interactif pour garantir que notre code se construit, que les tests réussissent et qu'un package peut être créé.

Les modèles de workflow par défaut sont d'excellents points de départ lors de la création d'un workflow de construction et de test, et nous pouvons personnaliser le modèle en fonction des besoins de notre projet.

---

## VIII. Lancement de tests unitaires avec github actions

---

Pour cette partie on a juste ajouté un bout de code qui nous permet, de mettre en œuvre la commande **mvn test** avec l'outil d'intégration continue.

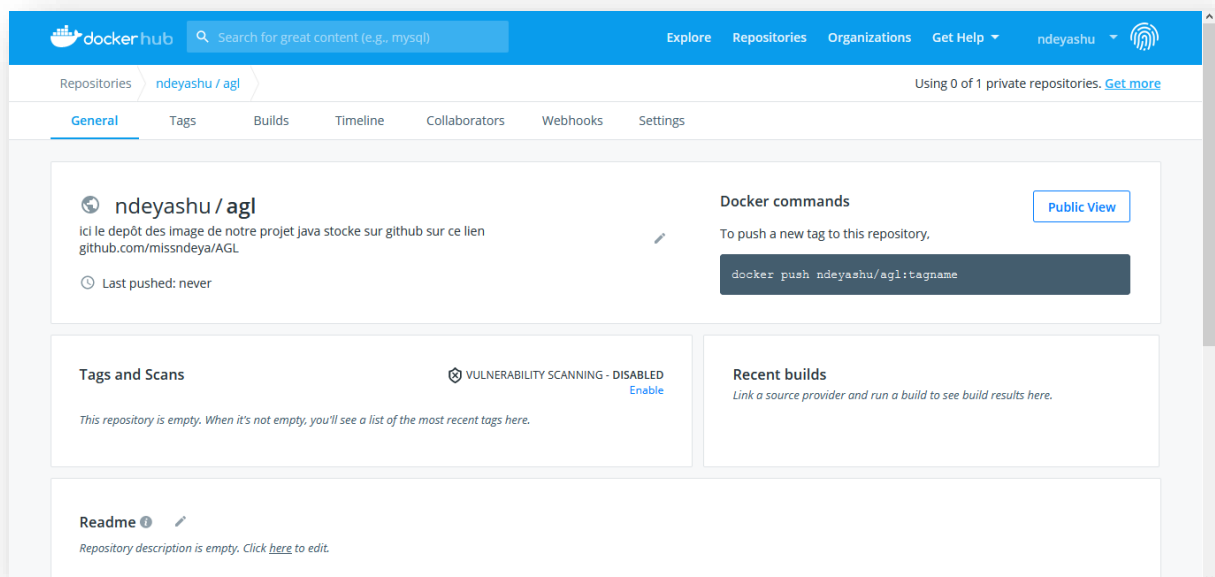
```

steps:
  - uses: actions/checkout@v2
  - name: Set up JDK 1.8
    uses: actions/setup-java@v1
    with:
      java-version: 1.8
  - name: Run Test
    run: mvn test
  - name: Build with Maven
    run: mvn -B package --file pom.xml

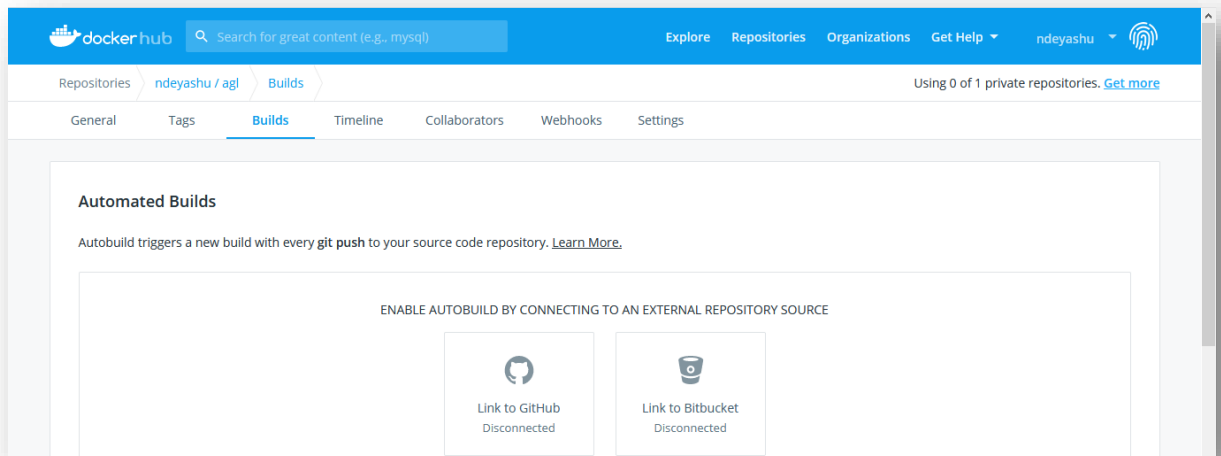
```

## IX. Publication de l'application Java sous forme d'image sur le docker hub avec github Action

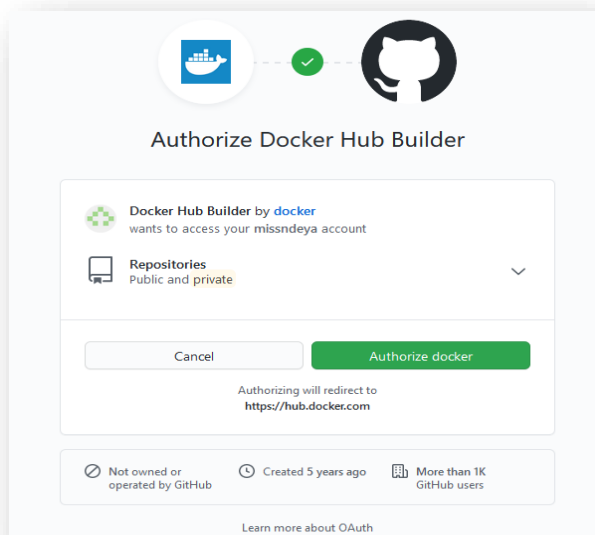
1. Après inscription et connexion sur <https://hub.docker.com>, on a créé un repository du nom d'agl



## 2. Puis on le connecte avec notre compte github



## 3. L'étape suivante consiste à confirmer l'autorisation



## 4. Configuration du build automatique avec le repository github



Search for great content (e.g., mysql)

[Explore](#)
[Repositories](#)
[Organizations](#)
[Get Help](#)

ndeeyashu

Repositories

ndeeyashu / agl

Builds

Edit

Using 0 of 1 private repositories. [Get more](#)

General

Tags

Builds

Timeline

Collaborators

Webhooks

Settings

Build configurations

SOURCE REPOSITORY

missndeya

×

Select repository

Required

NOTE: Changing source repository may affect existing build rules.

Cancel

Save

Build configurations

SOURCE REPOSITORY

missndeya

×

AGL

NOTE: Changing source repository may affect existing build rules.

BUILD LOCATION

Build on Docker Hub's infrastructure

AUTOTEST

Off

Internal Pull Requests

Internal and External Pull Requests

REPOSITORY LINKS

Off

Enable for Base Image

BUILD RULES

The build rules below specify how to build your source into Docker images.

Source Type	Source	Docker Tag	Dockerfile location	Build Context	Autobuild	Build Caching	
Branch	master	latest	Dockerfile	/	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	

View example build rules

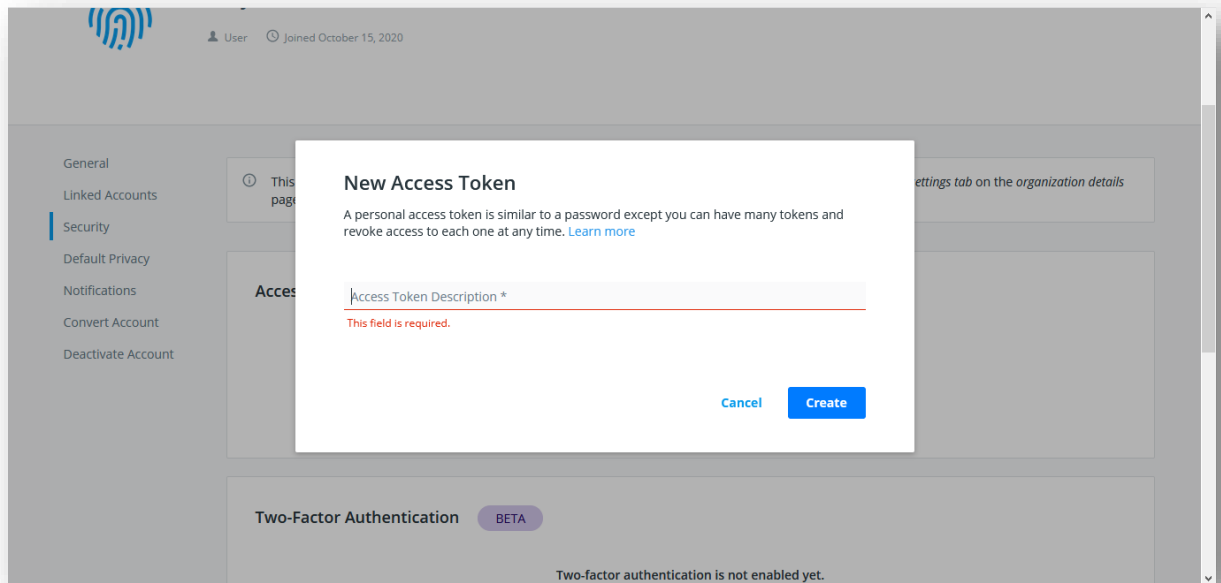
BUILD ENVIRONMENT VARIABLES

Cancel

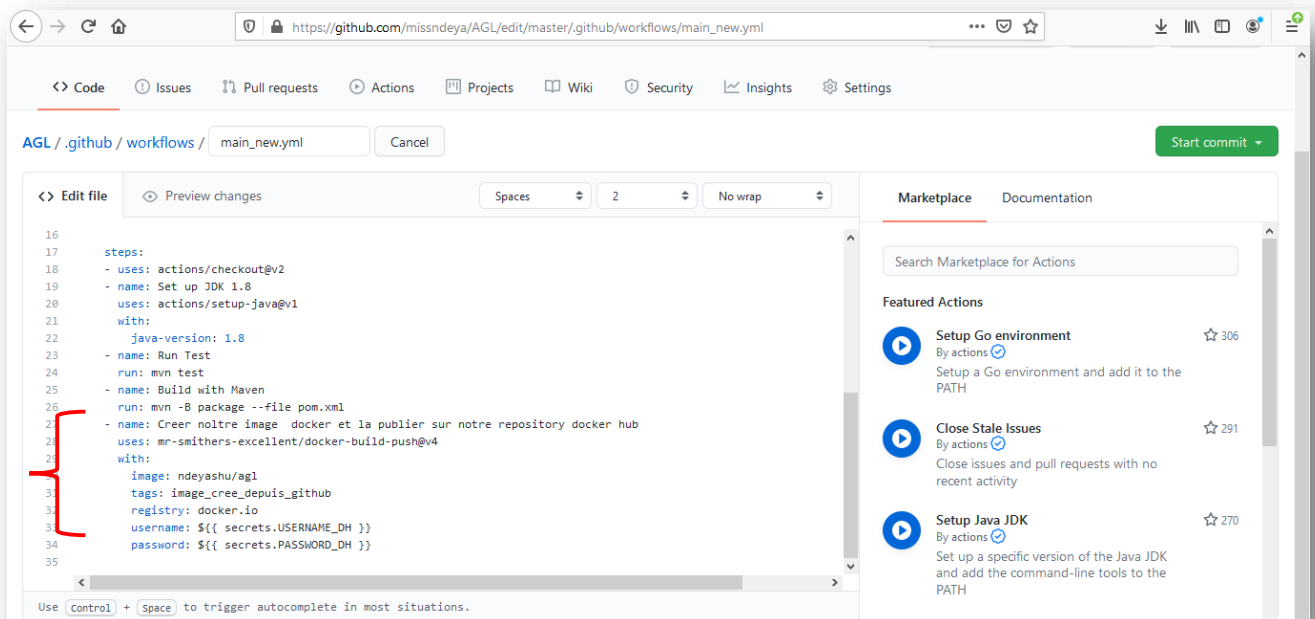
Save

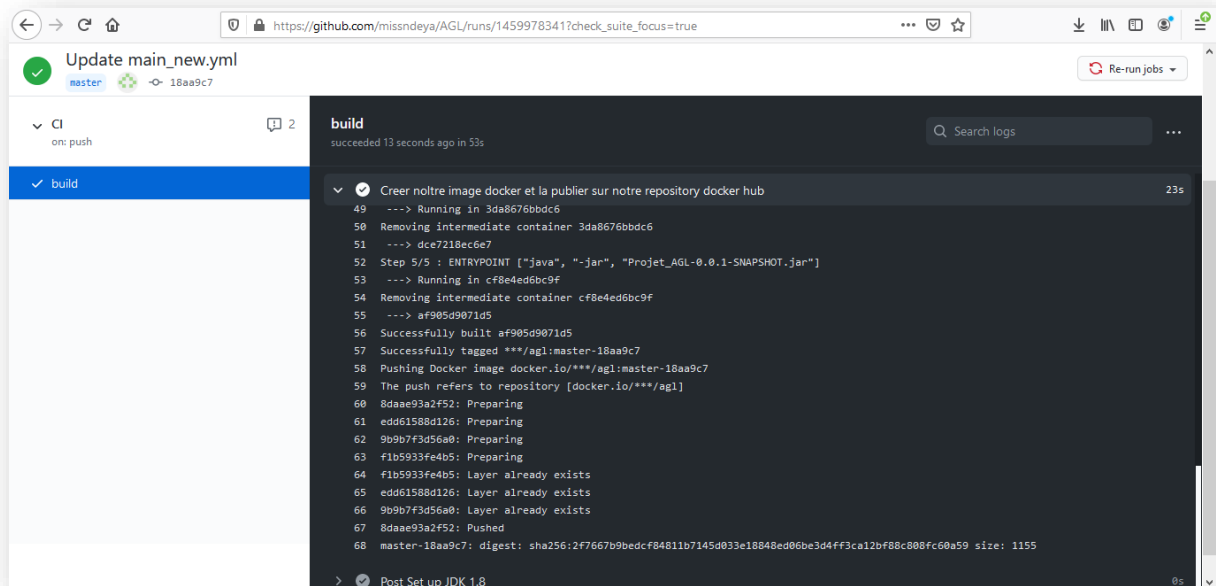
Save and Build

5. Après avoir configuré tout le nécessaire, nous avons Généré une clef d'accès: **build account > security > New Access Token**

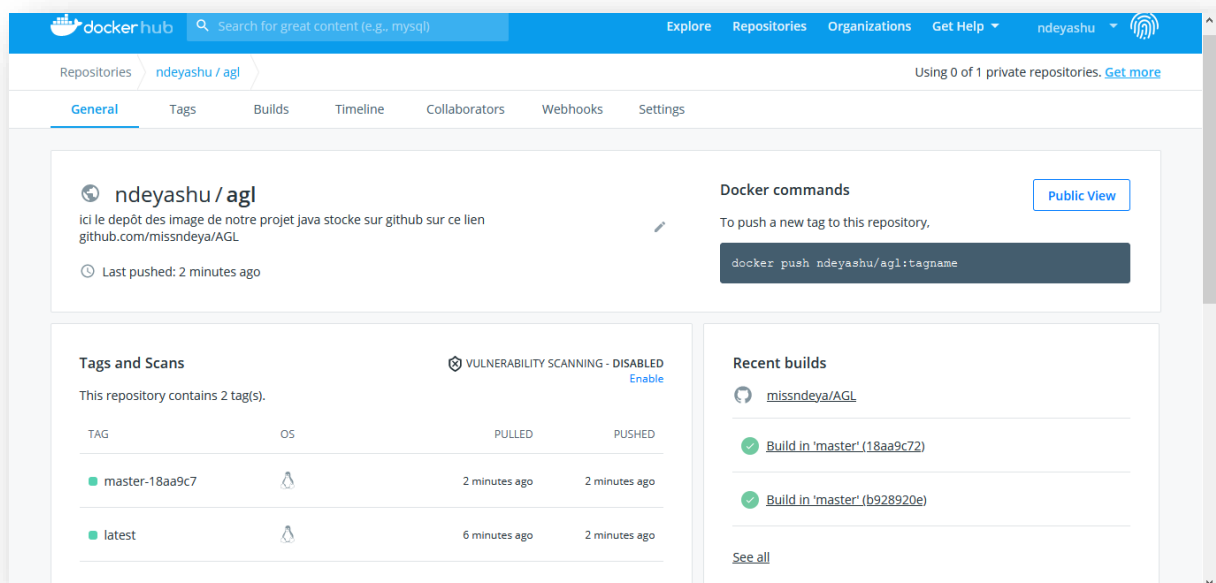


6. Dans cette section qui suit, on procède à l'exécution du push à partir de github actions





7. Et enfin, les résultats obtenus sont les suivants :



Voilà ! On vient donc de Publier notre application Java sous forme d'image sur le docker hub avec github Action.

## X. Conclusion

La vraie conclusion est optimiste si l'on s'éloigne du rêve, séduisant intellectuellement, de l'AGL universel, pour un espoir raisonnablement réalisable.

Les méthodologies qui émergeront sont celles qui exploiteront au mieux la créativité informatique, adaptables à des compétences "délocalisées". Ce sont les méthodologies qui permettront d'incorporer des développements innovants sans les considérer comme sauvages !

Les AGL sont plutôt destinés à devenir des plates-formes d'intégration logicielle. Alors, normes, connexion et interopérabilité assureront la pérennité des logiciels.