Hes·so VALAIS WALLIS
Haute Ecole d'Ingénierie
Hochschule für Ingenieurwissenschaften

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts
Western Switzerland

Filière Systèmes industriels

Orientation Infotronics

Travail de bachelor
Diplôme 2019

Marc Berguerand

Gitlab Intégration Continue avec « On Target Testing »

Professeur
Jérôme Corre

Expert
Michele Korell

Date de la remise du rapport
16.08.2019

HES-SO Valais

SYND	ETE	TEVI
X	X	X

Données du travail de diplôme
Daten der Diplomarbeit

FO 1.2.02.07.CB
haj/11/01/2016

Filière / Studiengang **SYND**	Année académique / *Studienjahr* **2018/19**	No TD / *Nr. DA* **it/2019/81**
Mandant / *Auftraggeber* ☒ HES—SO Valais ☐ Industrie ☐ Etablissement partenaire *Partnerinstitution*	Etudiant / *Student* **Marc Berguerand** Professeur / *Dozent* **Jérôme Corre**	Lieu d'exécution / *Ausführungsort* ☒ HES—SO Valais ☐ Industrie ☐ Etablissement partenaire *Partnerinstitution*
Travail confidentiel / *vertrauliche Arbeit* ☐ oui / ja [1] ☒ non / nein	Expert / *Experte (données complètes)*	

Titre / *Titel*

Gitlab Intégration Continue avec "On Target Testing"

Description / *Beschreibung*

L'école s'est équipée récemment de son propre serveur Gitlab pour héberger les travaux et projets de développements, permettant ainsi d'avoir un système de gestion de version fiable et équivalent aux attentes des partenaires commerciaux. Gitlab permet également d'exécuter des tests de régression (CI). Pour nos développements embarqués il est impératif de garantir et démontrer un niveau de qualité. L'école doit aussi inculquer ces bonnes pratiques aux partenaires commerciaux.
Dans le cadre de ce travail de Bachelor, l'objectif principal et de réaliser une plateforme d'exécution de tests liée au serveur Giltab de l'école. La plateforme doit permettre la compilation croisée, l'exécution des tests du code embarqué directement sur le microcontrôleur cible et enfin la possibilité de lancer des signaux dans les entrées et de tester les signaux de sortie du microcontrôleur. Eventuellement le travail peut être étendu à des cibles FPGA.

Tâches à réaliser:

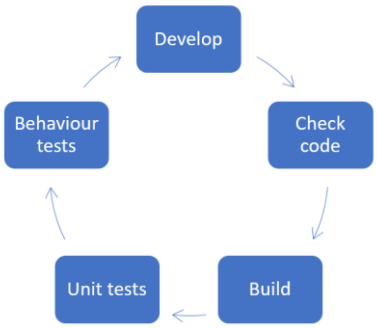
1. Choix et mise en place d'un démonstrateur d'exécution de test lié au serveur Gitlab de l'école
2. Démonstration de pipelines de tests, fonctionnels, comprenant le développement du code, l'archivage dans le serveur Gitlab, la compilation et l'exécution de tests d'intégration continue dans une Machine Virtuel ainsi que la compilation croisée et l'exécution de tests d'intégration continue sur la plateforme du microcontrôleur cible
3. Génération de signaux digitaux depuis le système de test vers les entrées de la cible, enregistrement de signaux digitaux générés aux sorties de la cible vers le système de test. Intégration de ces stimuli dans le pipeline de test
4. Si le temps le permet, réalisation de la même démonstration avec une cible FPGA

Signature ou visa / *Unterschrift oder Visum*	Délais / *Termine*
Responsable de l'orientation / filière *Leiter der Vertiefungsrichtung / Studiengang:* ..... [1] Etudiant / *Student* :	Attribution du thème / *Ausgabe des Auftrags:* **13.05.2019** Présentation intermédiaire / *Zwischenpräsentation* **13 – 14.06.2019** Remise du rapport / *Abgabe des Schlussberichts:* **16.08.2019, 12:00** Expositions / *Ausstellungen der Diplomarbeiten:* **28, 29 – 30.08.2019** Défense orale / *Mündliche Verfechtung:* **02 – 05.09.2019**

[1] *Par sa signature, l'étudiant-e s'engage à respecter strictement la directive DI.1.2.02.07 liée au travail de diplôme.*
Durch seine Unterschrift verpflichtet sich der/die Student/in, sich an die Richtlinie DI.1.2.02.07 der Diplomarbeit zu halten.

Rapport reçu le / *Schlussbericht erhalten am* ..... Visa du secrétariat / *Visum des Sekretariats* .....

test pipeline



Gitlab Continuous Integration « On Target Testing »

Diplômant Marc Berguerand

Objectif du projet

Réalisation d'un pipeline de tests pour une cible embarquée, qui doit pouvoir compiler du code, programmer la cible et un appareil simulant son environnement de travail, et vérifier le bon fonctionnement de la cible après la simulation.

Méthodes | Expériences | Résultats

Ce travail permet de démontrer qu'il est possible d'intégrer les tests hardware durant l'intégration continue.

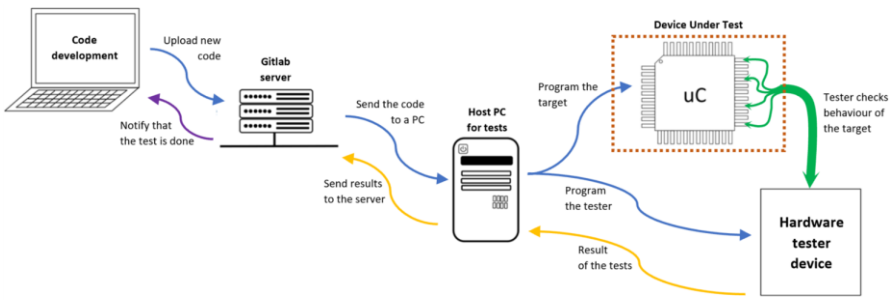
Divers outils permettant d'effectuer de l'intégration continue ont été testés afin de valider le choix d'utilisation du serveur Gitlab.
Un microcontrolleur ARM de ST (STM32F429i) a été choisie comme cible, et l'Analog Discovery 2 de Digilent en tant qu'appareil effectuant la simulation de l'environnement.
Un ordinateur a été lié au serveur Gitlab, qui va utiliser des images Docker durant le pipeline. Donc, des images Docker sont conçues pour avoir des environnements dédiés à l'exécution de diverses tâches :

- Lint
- Compilation
- Tests unitaires
- Programmation de l'ARM et de l'Analog Discovery 2 pour simuler l'environnement de l'ARM
- Analyse des résultats de la simulation

La simulation a permis de vérifier l'implémentation pour les deux cas d'exemples suivants :

- Portes logiques programmées dans l'ARM
- Traitement de données lors d'une communication UART

Le pipeline de test est donc fonctionnel. Il est donc possible de certifier le bon fonctionnement d'une cible à l'aide de l'intégration continue, le tout sur un serveur Gitlab.



Représentation des communications faites pour réaliser le pipeline de test

Travail de diplôme
| édition 2019 |

Filière
Systèmes industriels

Domaine d'application
Infotronics

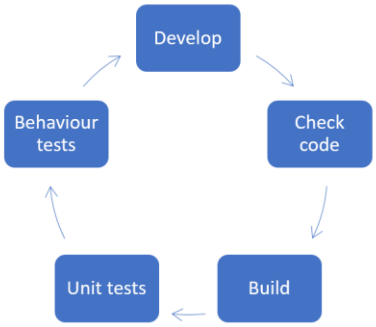
Professeur responsable
Jérôme Corre
jerome.corre@hevs.ch

Haute Ecole d'Ingénierie
Hochschule für Ingenieurwissenschaften

Gitlab Continuous Integration
« On Target Testing »

Diplomand Marc Berguerand

test pipeline



Diplomarbeit
| 2019 |

Studiengang
Systemtechnik

Anwendungsbereich
Infotronics

Verantwortliche/r Dozent/in
Jérôme Corre
jerome.corre@hevs.ch

Ziel des Projekts

Realisierung einer Testleitung für ein integriertes Ziel, die in der Lage sein muss, Code zu kompilieren, das Ziel und ein Gerät zu programmieren, das seine Arbeitshaltung simuliert, und nach der Simulation das ordnungsgemäße Funktionieren des Ziels überprüft.

Methoden | Experimente | Resultate

Diese Arbeit zeigt, dass es möglich ist, Hardwaretests während der kontinuierlichen Integration durchzuführen.

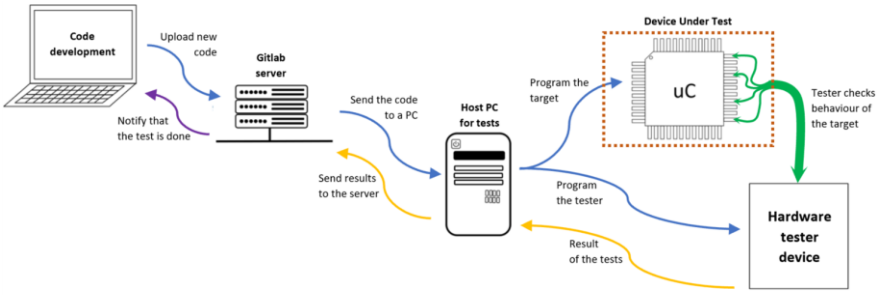
Verschiedene Tools für die kontinuierliche Integration wurden getestet, um die Wahl der Nutzung des Gitlab-Servers zu bestätigen.
Als Ziel wurde ein ST ARM-Mikrocontroller (STM32F429i) und das Digilent Analog Discovery 2 als Umgebungsimulationsgerät gewählt.
Ein Computer wurde mit dem Gitlab-Server verbunden, der während der Pipeline Docker-Bilder verwendet. Docker-Bildern sind also so konzipiert, dass sie Umgebungen besitzen, in denen verschiedene Aufgaben ausgeführt werden können:

- o Lint
- o Kompilierung
- o Einzeltests
- o Programmierung von ARM und Analog Discovery 2 zur Simulation der ARM-Umgebung
- o Analyse der Simulationsergebnisse

Die Simulation ermöglichte es, die Implementierung für die folgenden beiden Beispielfälle zu überprüfen:

- o Im ARM programmierte Logikgatter
- o Datenverarbeitung während einer UART-Kommunikation

Die Testschlaufe ist somit funktionsfähig. Dannach ist es möglich, die einwandfreie Funktion eines Ziels durch kontinuierliche Integration auf einem Gitlab-Server zu zertifizieren.



Darstellung der Kommunikation zum Bau der Testleitung

Remerciements

Ce projet n'aurait pu autant avancer sans la contribution de plusieurs personnes. Je tenais donc à les remercier pour leur soutien et conseils.

- **M. Jérôme Corre** en tant que professeur attentionné, proposant cette thèse et m'ayant conseillé dans les étapes de celle-ci.
- **M. Michele Korell** en tant qu'expert pour cette thèse, pour le temps passé à l'évaluation de celle-ci et pour ses conseils sur les images Docker et leurs exécutions.
- **M. Mickaël Clausen** pour l'indication de l'existence des outils type Analog discovery 2 et le maintien du serveur Gitlab de l'école.
- **M. Blaise Evequoz** pour le prêt d'un Analog Discovery 2, me permettant de m'orienter vers ce dispositif pour effectuer mes tests.
- **Service Informatique (plus précisément M. Yves-Benoît Zufferey)** pour leur aide à l'établissement de la connexion sécurisée entre le serveur et la machine physique.
- **Service Digilent (plus précisément le modérateur du forum technique - attila)** pour l'aide apportée à l'utilisation de l'Analog Discovery 2.

Sans oublier ma **famille**, **mes proches** et **mes amis** pour leur soutient et leur aide au dépassement de soi-même.

« Aucun de nous ne sait ce que nous savons tous, ensemble »

Euripide

Glossaire

Abréviation	Signification
CI	Continuous Integration « Intégration Continue »
CD	Continuous Delivery « Distribution Continue »
DUT	Device Under Test « Appareil sous tests »
UI	User Interface « Interface utilisateur »
TCP	Transmission Control Protocol « Protocole de contrôle de transmission »
SSH	Secure Shell « Protocole de transmission sécurisé »
TD	Travail de Diplôme
uC	Microcontroller « Microcontrolleur »
AD2	Analog Discovery 2
FPGA	Field-Programmable Gate Array « Circuit logique programmable »
USB	Universal Serial Bus « Bus universel en série »
UART	Universal Asynchronous Receiver Transmitter « Émetteur-récepteur asynchrone universel »
SDK	Software Development Kit « Kit de développement en informatique »
TDD	Test-Driven Development « Développement piloté »
CA	Certificate Authority « Autorité de certification »
PC	Personnal Computer « Ordinateur personnel »

Table des matières

Liste des figures

1 Introduction

Les gestionnaires de sources ne sont plus à présenter. Ils simplifient grandement la conception d'un logiciel au sein d'une équipe [1]. Il n'y a plus besoin de bloquer le développement de celle-ci pour que chacun ait une version qui ne contienne pas d'erreurs. Chaque développeur s'attarde sur sa partie logicielle et il sera alors possible, d'une fois qu'elle est terminée, de la fusionner avec le code fonctionnel. Autant dire qu'il s'agit là d'une grande innovation dans la conception software.

Dès l'apparition de ceux-ci, des plateformes d'hébergements pour tous ces projets liés à un gestionnaire de sources/versions voient le jour, telles que Gitlab, Bitbucket ou Github, pour ne citer que quelques-unes d'entre elles. Avec celles-ci, de nouvelles fonctionnalités se sont développées, telles des possibilités d'intégration continue (CI) ou de distribution continue (CD). Ceci facilite la vérification du fonctionnement du logiciel ainsi que le déploiement de celui-ci [2]. Par exemple, l'intégration continue permettra de savoir s'il y a des erreurs durant l'exécution du programme. Auquel cas il sera alors possible de corriger ses erreurs avant de fusionner le travail effectué.

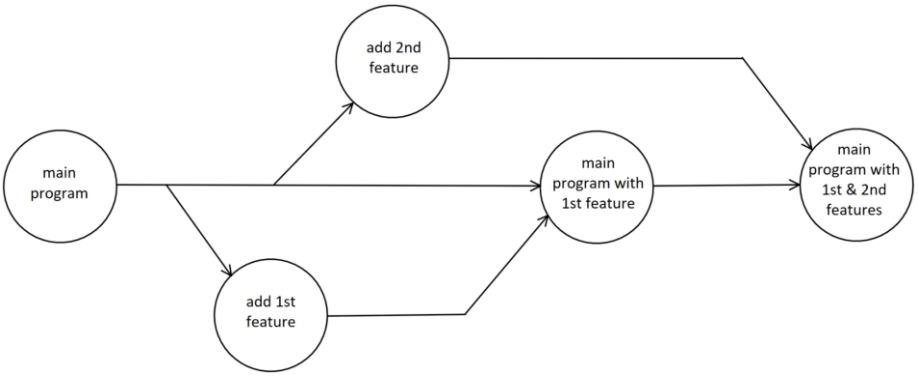


Figure 1: exemple de travail collaboratif à l'aide d'un gestionnaire de sources

Les entreprises s'adaptent de plus en plus à ces propriétés, qui ne sont que bénéfiques pour leur capacité de production. En effet, les tests peuvent être automatisés et donc effectués à n'importe quel moment, il y a donc un évident gain de temps. De plus, les tests ne peuvent être biaisés par la machine sans l'intervention d'une personne, ce qui rend l'exécution de ceux-ci beaucoup plus cadrée.

1.1 Contraintes actuelles

Ces intégrations continues ne sont possibles que pour la vérification d'un comportement logique d'un programme. De ce fait, le résultat d'une opération peut être approuvé à l'aide de ces tests d'intégration. Néanmoins, une hypothèse de base est faite, celle que la cible va se comporter exactement telle qu'il est demandé au niveau physique.

Ceci implique que ces tests sont moins utiles lorsqu'il s'agit d'un développement « hardware ». Certes, il est actuellement possible de vérifier le bon fonctionnement d'un microprocesseur (par exemple, si un Personnal Computer (PC) effectue les tâches correctement), mais pas pour un microcontrôleur (uC).

Il est, pour autant, possible d'évaluer le comportement logique d'un code développé pour un uC, mais le type d'utilisation de celui-ci n'est pas le même que celui d'un ordinateur. Il peut exister des problèmes dans l'exécution dudit code, tels que le temps passé dans une interruption ou la gestion des ressources, qui sont critiques dans ces systèmes embarqués.

1.2 Objectif

Ceci amène donc au but de cette thèse, qui a pour but de lier le serveur présent dans l'école (il s'agit d'un serveur Gitlab), avec une plateforme de tests pour des composants physiques. Ladite plateforme se doit de pouvoir compiler du code émit sur le serveur et l'exécuter sur le composant ciblé. Une fois ceci fait, des signaux pourront alors être émis sur celui-ci ou récupérés depuis celui-ci. Ci-dessous, une représentation graphique du cycle complet exécuté lors d'une mise à jour du logiciel dans le serveur :

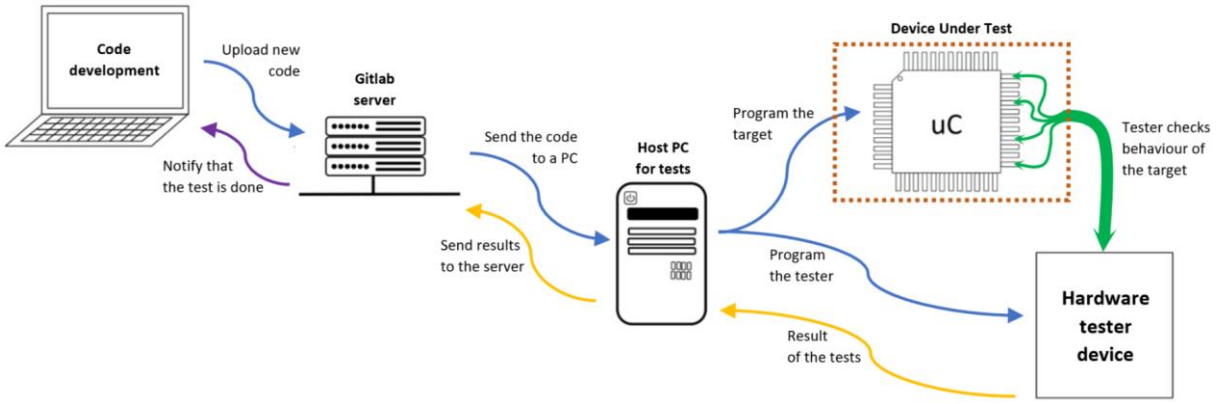


Figure 2: Représentation du cycle exécuté lors d'un git push

1.3 Étapes de la thèse

- Choix de la plateforme d'hébergement
  - Analyses de diverses plateformes afin de justifier le choix du serveur GitLab (ou non)
- Choix de l'outil d'intégration continue
  - Analyses de divers outils pour utiliser celui qui correspond le mieux à la tâche demandée
- Réalisation de l'intégration continue « On Target Testing »
  - Définition d'une cible à tester
  - Installation d'une machine physique dédiée à tester celle-ci
  - Liaison de ladite machine à l'outil d'intégration continu
  - Réalisation d'un projet exemple sur ladite cible
  - Élargissement du type de cible

2 Analyses

2.1 Pipeline de tests

Avant toute chose, il est important de savoir ce qu'est un pipeline de test et son utilité. Ainsi, toutes les futures étapes feront plus de sens dans le développement de celui-ci.

2.1.1 Qu'est-ce qu'un pipeline de tests ?

Afin de s'assurer que le projet en cours de développement fonctionne, un pipeline de test est utile. En effet, il va garantir que les modifications apportées au projet ne comportent aucune erreur, autant au niveau de la norme d'encodage, de la compilation ou encore de tests. Ainsi, le développeur peut rester serein quant au travail effectué. Dans la figure ci-dessous, un exemple d'un pipeline de test.

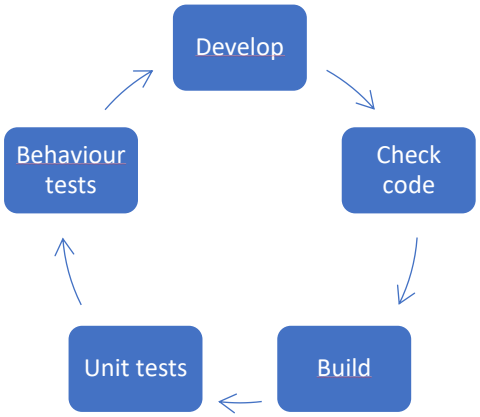


Figure 3 : Exemple d'un pipeline de test

Dans l'exemple ci-dessus, plusieurs étapes sont effectuées. Tout d'abord, le développeur programme. Une fois qu'il est satisfait de son code, il va l'envoyer au serveur. Le serveur va alors lancer une suite de tests. Le premier est communément appelé « Lint ». Il s'agit d'un vérificateur de code, qui va certifier si la syntaxe du code est conforme à une norme définie par les développeurs ou l'entreprise pour laquelle il travaille. Ensuite se trouve la compilation du programme, qui va permettre de s'apercevoir s'il y a des erreurs au sein dudit programme.

Une fois ces étapes « statiques » effectuées, les tests « dynamiques » ont lieux. Il y a alors les tests unitaires (aussi appelés « White Box ») qui vont certifier que les méthodes conçues répondent de la manière souhaitée (par exemple, lorsqu'un « printf » est exécuté, que le String souhaité soit bien affiché comme il le faut). Une fois les tests unitaires en ordre, un test plus global a lieu, celui du comportement, aussi nommé « Black box ». Celui-ci ne s'occupe pas de ce qui se passe comme commandes au sein du logiciel, mais ne vérifie que les résultats en fonction des stimulations définies (par exemple, lors d'un appui sur un bouton, une LED s'allume. Le code intervenant entre deux n'est pas connu).

2.1.2 A quoi cela est utile ?

Avec ceci, il est possible de déterminer à quel endroit du pipeline un problème a lieu. Il est possible qu'un développeur ne s'occupe que d'une partie d'un programme complexe. Afin de savoir si ce qu'il a effectué n'engendre pas d'erreur sur le comportement global du dit logiciel, ces tests s'avèrent indispensables.

Ceci permet également de s'assurer que les fonctionnalités développées et testées dans le passé fonctionnent toujours après la dernière modification du programme (test régressif).

2.2 Choix de la plateforme d'hébergement

Il existe une multitude de plateformes d'hébergements de gestionnaires de source. Étant donné que certaines d'entre elles sont plus utilisées que d'autres, un filtre est déjà effectué en analysant uniquement celles-ci :

- Github[1]
- Gitlab[2]
- Bitbucket[3]

Veuillez prendre en compte que les informations suivantes sont valables au mois de mai 2019, il est possible que des modifications des conditions d'utilisations aient changées d'ici là.

2.2.1 Compte-rendu

L'analyse des différentes plateformes amène au tableau ci-dessous, dans le cas d'un compte gratuit. Pour de plus amples informations concernant chacune des plateformes d'hébergement, veuillez regarder les annexes 1 (Utilisation de Github), 2 (Utilisation de Gitlab) et 3 (Utilisation de Bitbucket).

Caractéristiques	Github	Gitlab	Bitbucket
nombre de répertoires publics	illimité	illimité	illimité
nombre de répertoires privés	illimité	illimité	illimité
nombre de collaborateurs par projets privés	3	illimité	5
suivi des problèmes et bugs	oui	oui	oui
outils CI/CD intégré	non	oui	oui
temps disponible par mois pour CI/CD	-	2000 min	50 min
Possibilité d'avoir son propre serveur	non	oui - gratuit	oui - payant

Figure 4: Comparatif entre les plateformes d'hébergement

Github offre donc le moins de possibilités. En effet, il n'est ni possible d'avoir son propre serveur, ni possible de réaliser du CI directement depuis cette plateforme. Elle offre également le moins de collaborateurs possible parmi les trois plateformes pour des projets privés.

Bitbucket permet en revanche d'effectuer du CI, gratuitement pendant 50 minutes chaque mois. Elle offre également la possibilité d'avoir son propre serveur – contre paiement. Une limite de collaborateurs est toujours de mise pour cette plateforme concernant les projets privés, quoique supérieure à la limite imposée par Github.

Gitlab propose le CI pour 2000 min chaque mois gratuitement, ce qui en fait la plus généreuse des plateformes à ce sujet. Elle offre la possibilité d'avoir son propre serveur gratuitement, et n'impose pas de limite concernant le nombre de collaborateurs par projets.

Ainsi, le choix s'est porté sur Gitlab.

[1] https://github.com/
[2] https://about.gitlab.com/
[3] https://bitbucket.org/product/

2.3 Choix de l'outil d'intégration continue

Plusieurs outils permettant de faire du CI existent. Certains d'entre eux sont plus répandus, et d'autres sont intégrés aux plateformes d'hébergement. Ainsi, un choix arbitraire est fait concernant les outils CI à tester. Voici-en la liste :

- Gitlab-CI[4]
- Travis[5]
- Bitbucket-Pipeline[6]
- Jenkins[7]

Veuillez prendre en compte que les informations suivantes sont valables au mois de mai 2019, il est possible que des modifications des conditions d'utilisations aient changées d'ici là.

2.3.1 Compte-rendu

L'analyse des différents outils amène au tableau ci-dessous, dans le cas d'une utilisation entièrement gratuite. Pour de plus amples informations concernant chacun des outils d'intégration continue, veuillez regarder les annexes 4 (Utilisation de Gitlab CI/CD), 5 (Utilisation de Bitbucket - pipelines) et 6 (Utilisation de Travis CI).

Caractéristiques	Gitlab - CI	Bitbucket -pipelines	Travis	Jenkins
Dédier une machine physique comme exécuteur	oui	non	non	oui
Possibilité de liaison avec Gitlab	oui	non	non	oui
Temps disponible par mois pour le CI	2000 min pour runners de gitlab.com	50 min	pas de limite	pas de limite
Espace mémoire pour les artéfactes	configurable / garde un certain temps	1 GB	aucun	espace de la machine à disposition
Possibilité d'effectuer plusieurs tâches simultanément	oui	oui	oui	oui
Informe en temps réel les étapes du CI	oui	oui	oui	oui

Figure 5: Comparatif entre les outils CI

Travis, l'outil CI typiquement utilisé avec github, possède le plus d'inconvénients. En effet, il ne permet pas de sauvegarder des fichiers entre les diverses étapes de l'intégration, ni de définir une machine physique comme exécuteur du CI. De plus, pour les mêmes étapes, plus de commandes doivent être entrées.

Bitbucket-pipelines ne permet également pas de définir une machine physique en tant qu'exécuteur du CI. Concernant le reste des caractéristiques, toutes sont valides, avec quelques limitations pour certaines d'entre elles. En effet, il faudrait alors utiliser Bitbuket comme plateforme d'hébergement et le temps offert pour le CI est limité à 50 minutes par mois. De plus, une limite est imposée quant à la taille mémoire des fichiers enregistrés durant le CI.

Le choix s'est alors porté sur Gitlab-CI et Jenkins. En effet, ces deux outils permettent de définir des machines physiques fixes (appelées « runners » pour gitlab) sur lesquelles il sera possible de connecter l'appareil à tester et le testeur.
De plus, ils offrent tous les deux la possibilité de lier l'exécution de leur CI à des répertoires désignés sur Gitlab, qui est la plateforme d'hébergement choisie.

Dans un premier temps, le déploiement du CI est réalisé par Gitlab CI, car cet outil nécessite bien moins de configurations que Jenkins et est plus intuitif.

[4] https://docs.gitlab.com/ee/ci/
[5] https://travis-ci.org/
[6] https://bitbucket.org/product/features/pipelines
[7] https://jenkins.io/

2.4 Choix d'une cible

À ce stade, les outils sont prêts à effectuer l'intégration continue. Afin de se rendre compte de la puissance de ceci, un projet type est créé, dont la cible doit être choisie. Dans un premier temps, l'exemple se portera sur un uC. Celui choisi est le STM32F429I (basé sur une architecture ARM Cortex-M4), car il s'agit d'un uC provenant du même fabricant que celui qui est utilisé pour l'apprentissage de la programmation sur systèmes embarqués au sein de la HES. Ainsi, les mêmes outils sont utilisés pour le développement de code sur le uC. Afin de profiter pleinement des capacités offertes par ce uC, la carte d'évaluation STM32F429I-DISC1 est utilisée.

2.5 Choix d'un appareil émulant l'environnement de la cible

Afin d'émuler un environnement physique pour la cible, un appareil de tests est utilisé. Celui-ci va permettre de générer des signaux, tout comme en mesurer. En analysant les signaux émis/reçus, le comportement du système pourra être certifié.

2.5.1 Appareils existants

Peu de fabricants proposent des systèmes conçus à ces fins. Voici quelques appareils qui ont été analysés :

- Saleae Logic Pro 16[8]
- Digilent Analog Discovery 2[9]
- Digilent Analog Discovery Studio[10]

Comme il est constatable, Digilent est l'un des rares fabricants pour ce type d'appareils. Afin de choisir celui qui est le plus avantageux pour une utilisation en tant qu'émulateur pour une cible, voici un comparatif entre les trois appareils susmentionnés :

Caractéristiques	Saleae Logic Pro 16	Digilent Analog Discovery 2	Digilent Analog Discovery Studio
IO digitales configurables en entrées/sorties	non, uniquement lecture	oui	oui
nombre d'IO digitales	16 (configurable analogiques)	16	16
Sortie analogique	0	oui	oui
nombre de sorties analogiques	-	2	2
Entrée analogique	oui	oui	oui
nombe d'entrée analogique	16 (configurable digitale)	2	2
vitesse maximale d'un signal	1MHz en analogique 25 MHz en digital	BNC:30MHz en entrée BNC:12MHz en sortie Digital : 100 MS/s	BNC:30Mhz en entrée MTE: 9MHz en entrée BNC/MTE: 8MHz en sortie Digital: 100MS/s
programmable via un script (C/C++/Python/VB)	oui, C++	oui, Python, C++, Matlab, LabView	oui, Python, C++, Matlab, LabView
peut utiliser des protocoles (USART, I2C,...)	oui	oui	oui
peut utiliser des protocoles propriétaires	oui	oui	oui
peut enregistrer des données temporelles des signaux	oui	oui, avec un script ou matlab	oui, avec un script
coût	$999	$279	$599

Figure 6: Analyse des divers appareils de tests

Au vu des diverses caractéristiques des appareils, le Logic Pro 16 de Saleae n'est pas adéquat. En effet, il sert de data logger mais ne peut pas générer de signaux.

[8] https://www.saleae.com/fr/
[9] https://store.digilentinc.com/analog-discovery-2-100msps-usb-oscilloscope-logic-analyzer-and-variable-power-supply/
[10] https://store.digilentinc.com/analog-discovery-studio-a-portable-circuits-laboratory-for-every-student/

Les deux appareils de Digilent se ressemblent au niveau de leur compétences. Cependant, l'Analog Discovery 2 est environ deux fois moins cher que l'Analog Discovery Studio. La différence entre ces deux appareils réside dans leur fonction physique. En effet, le Discovery 2 est un boitier clos, qui est connecté à l'aide d'une série de câbles sur l'appareil à tester. En revanche, le Discovery Studio est plus orienté vers les plaques d'expérimentations. Voici deux brèves photos qui vont permettre de mieux comprendre ceci :

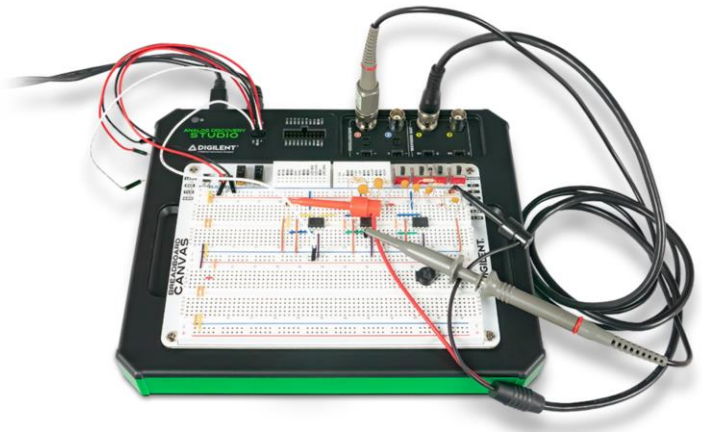


Figure 7: Digilent Analog Discovery Studio

Source : DIGILENT, Analog Discovery Studio, https://store.digilentinc.com/analog-discovery-studio-a-portable-circuits-laboratory-for-every-student/

Figure 8: Digilent Analog Discovery 2

Source : DIGILENT, Analog Discovery 2, https://store.digilentinc.com/analog-discovery-2-100msps-usb-oscilloscope-logic-analyzer-and-variable-power-supply/

Ainsi, pour l'utilisation faite dans cette thèse, l'Analog Discovery 2 est choisi.

3 Méthodologie

3.1 Lier une machine physique à l'outil CI

Étant donné que les outils de développement ont été choisis auparavant (plateforme Gitlab avec Gitlab-CI), il faut maintenant connecter une machine physique à l'outil CI. Pour ce faire, Gitlab offre l'utilistation de « Runners ». Il s'agit d'un environnement dans lequel seront exécutées les commandes du CI. Pour de plus amples informations concernant la démarche pour la liaison entre une machine et la plateforme d'hébergement, veuillez-vous référez à l'annexe 4 Chapitre 5.1.

3.1.1 Relier une machine physique à Gitlab CI

Pour réaliser la liaision, plusieurs méthodes sont proposées par Gitlab CI[11] :

- Se connecter via un protocole de contrôle de transmission (TCP) directement dans l'interface Shell de la machine hôte
- Établir une connexion sécurisée (SSH) entre le CI et l'hôte
- Lancer une machine virtuelle et utiliser celle-ci dans la machine principale
- Utiliser des images docker

Il est cependant important de mentionner que la connexion SSH est déconseillée par Gitlab, car elle est susceptible à des attaques[12]. La plateforme d'hébergement recommande également d'utiliser une image Docker au lieu d'une machine virtuelle, pour plus de découplage et moins d'espace mémoire utilisé sur la machine principale.

Ayant essayé d'utiliser chacune de ces méthodes, les deux seules qui n'ont pas fonctionnées sont celles déconseillées par Gitlab. La cause du disfonctionnement du SSH est que la connexion n'est pas autorisée par Gitlab. Quant à la machine virtuelle, la machine hôte n'arrive pas à la lancer via la commande externe envoyée par la plateforme d'hébergement, qui utilise le SSH. Ce problème lié à l'utilisation d'une connexion SSH avec Giltab CI est connu mais n'est pas encore résolu, ce qui amène à la conclusion qu'actuellement il est préférable d'utiliser des images Docker ou l'interface Shell de la machine hôte.

3.1.2 CI via l'interface shell

Avec cette liaison, la machine exécute le script de Gitlab CI dès que cet outil le demande. Voici plusieurs avantages de cette manière de connexion :

- Simple à utiliser
- Rapide à mettre en place
- Tous les outils installés sur la machine hôte sont accessibles
- Le processus de CI est transparent, et donc la machine peut être utilisée normalement

Cependant, cette liaison possède quelques désavantages :

- La connexion n'est pas sécurisée de base
- Il est possible d'accéder à tous les fichiers de la machine hôte

Il est néanmoins possible de remédier à la connexion non-sécurisée avec des certificats. Ainsi, la machine hôte envoie une requête à l'autorité de certification (CA) du serveur Gitlab, afin qu'il le signe et que la communication devienne ainsi sécurisée. À noter qu'il n'est pas possible de faire ceci si le serveur est la plateforme principale de gitlab, soit gitlab.com.

[11] https://docs.gitlab.com/runner/ dans la partie « Selecting the executor »
[12] https://docs.gitlab.com/runner/executors/ssh.html dans la partie « Sécurité »

3.1.3 CI via une image docker

L'avantage des images docker est qu'elles sont portables. Ainsi, n'importe quelle machine pouvant utiliser Docker pourra effectuer le CI (Une plus grande explication concernant le fonctionnement et l'utilité de Docker se trouve au chapitre 3.3 ci-après). Chaque image est conçue depuis une base définie. Ainsi, un développeur pourra concevoir une image Docker ayant comme base Ubuntu, Windows, ou OSx, le tout dans n'importe laquelle de ses versions, permettant ainsi de tester la portabilité du code. Dans le cadre de cette thèse, l'image conçue servira pour d'éventuelles entreprises ou personnes intéressées aux tests CI pour le hardware. Ainsi, les futurs utilisateurs ne devront pas installer d'autres logiciels que Gitlab-Runner et Docker, lequel utilisera l'image conçue durant ce travail de diplôme (TD).

Cette méthode offre les mêmes avantages que l'interface Shell :

- Simple à utiliser
- Rapide à mettre en place
- Tous les outils installés sur la machine hôte ou de l'image docker sont accessibles
- Le processus de CI est transparent, et donc la machine peut être utilisée normalement

Et palie aux inconvénients de celle-ci :

- Il est impossible d'accéder à tous les fichiers de la machine hôte sauf si on le spécifie (ce qui est fortement déconseillé)
- La connexion n'est toujours pas sécurisée, mais comme l'environnement d'exécution est cloisonné et que l'image est détruite à la fin de l'exécution, il n'y a aucun risque pour la machine physique.

3.2 Environnement de tests CI

Avec les deux options vues précédemment aux chapitres 3.1.2 et 3.1.3, la solution d'une image docker a été choisie. En effet, elle permet de sécuriser l'environnement de test et permet à toute autre personne souhaitant réaliser le même type de CI de le faire de manière très simple, en prenant les images qui seront configurées par la suite.

Il est souhaitable d'utiliser un runner local qui est connecté à gitlab, afin de s'affranchir de la limite des 2000 minutes par mois offertes par Gitlab pour utiliser ses runners, mais également pour pouvoir disposer des outils qui seront définis par la suite, soit la cible à programmer et à tester, ainsi que l'appareil qui va émuler les signaux d'entrées/sorties de la cible.

Le point fort de Gitlab CI est qu'il permet d'utiliser plusieurs images Docker en parallèle, tout en pouvant déterminer un nombre maximal de chaque « instance » d'image (voir annexe 4 chapitre 7).

De plus, si de multiples tests veulent être effectués mais que tous les runners sont utilisés, ceux-ci sont automatiquement mis en attente et seront lancés une fois qu'un runner est utilisable.

Pour lier une machine physique à Gitlab, veuillez-vous référer à l'annexe 4 chapitre 5.1.

3.3 Utilisation de Docker

Docker est un logiciel émulant un environnement. Celui-ci peut être configuré de diverses sortes. L'une d'entre elle est d'utiliser un DockerFile pour concevoir une image. Une autre est d'utiliser une image déjà existante et de la modifier à sa guise pour la sauvegarder par la suite.

3.3.1 Image docker depuis un Dockerfile

Un Dockerfile est un fichier permettant de configurer basiquement une image pour un environnement de travail personnalisé. Ainsi, il est aisé de concevoir une image basée sur Linux, Windows ou encore OSx. Pour de plus amples informations à ce sujet, veuillez-vous référer à l'annexe 7 chapitre 7.1.

3.3.2 Image docker depuis une image déjà existante

Une image docker peut être exécutée sur n'importe quelle machine, du moment qu'elle possède Docker. La plateforme d'hébergement Docker Hub[13] offre de multiples images disponibles au téléchargement (pour de plus amples informations, veuillez-vous référer à l'annexe 7 chapitre 7.2.

Durant le TD, une image est générée depuis un Dockerfile, puis est transmise à la plateforme Docker Hub. Ceci permet d'utiliser l'image directement depuis ladite plateforme, ce qui simplifiera la liaison entre la machine physique et le serveur Gitlab (plus de détails dans l'annexe 4, chapitre 5.1).

[13] https://hub.docker.com/

3.4 Utilisation du compilateur ARM

Pour pouvoir compiler du code sur un ARM, il faut télécharger un compilateur dédié à ladite cible. Celui utilisé est gcc-arm-embedded[14], qui est gratuit.

3.5 Installer gcc-arm-embedded

Pour installer le compilateur sur l'image Docker choisie précédemment (ubuntu), il faut utiliser la commande suivante :

```
add-apt-repository ppa:team-gcc-arm-embedded/ppa
apt-get update
apt-get install gcc-arm-embedded
```

Figure 9: Commandes pour installer gcc-arm-embedded

Ainsi, le compilateur officiel est recherché avec la première commande. Il s'en suit une mise à jour des packages disponibles à l'installation pour finalement installer le dit compilateur.

Une fois le compilateur installé, il est possible de directement compiler du code. Toutefois, ceci implique énormément de paramètres pour un programme conséquent, et la génération du fichier programmable avec toutes ses dépendances compilées prend du temps. Ainsi, un logiciel dédié au déploiement de fichiers exécutables est usuellement utilisé. Dans ce cas il s'agit de Make.

Voici un exemple d'utilisation d'une commande make dans le fichier « gitlab-ci.yaml », qui va permettre d'exécuter l'intégration continue :

```
stages:
  - make

execute the makefile:
  stage: make

  tags:
    - ARM gcc

  script:
    - make clean
    - make BINPATH=/usr/bin
```

Figure 10: Fichier gitlab-ci.yaml utilisant make

Ainsi, l'étape « make » est exécuté sur un Runner ayant comme tag « ARM gcc ». Ceci spécifie que le Runner possède gcc-arm-embedded dans notre cas. Puis, le script effectué est d'abord un make clean, qui va simplement vider le répertoire des fichiers inutiles afin de générer l'exécutable complet. Une fois ceci fait, la commande make est utilisée. À noter qu'ici le paramètre « BINPATH=/usr/bin » est ajouté, sinon les librairies compilées ne se trouvent pas au bon emplacement dans la machine.

[14] https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm

3.6 Make

Make[15] permet de compiler uniquement les parties du programme qui sont nécessaires lors de modifications d'un ou de plusieurs fichiers. La liste définissant ce qui est exécuté se trouve dans un fichier nommé « Makefile ». En voici un exemple simple, dont les commentaires expliquent le contenu de celui-ci :

```
#Default target executed when no arguments are given to make
default_target: all

#Specify that default_target in the makefile
#  isn't a parameter
.PHONY : default_target

#Here regroup the sources in a "Macro" named SRC_FILES
SRC_FILES = main.c operations.c
#Here regroup the headers in a "Macro" named INC_FILES
INC_FILES = operations.h

#When "make all" is called, the command depends on
# the outputRun.o file, it also checks the o
# utputRun.o dependencies
all: outputRun.o

#Here are the outputRun.o dependencies, if SRC_FILES
# or INC_FILES have changed, it will execute the
# gcc command
outputRun.o: $(SRC_FILES) $(INC_FILES)
    #compiles all the sources and the output is named
    # as the main depencie target (outputRun.o) = $@
    gcc $(SRC_FILES) -o $@

#When "make clean" is called, the "rm" commande below
# is executed
clean:
    #And remove all the files that ended with ".o"
    rm -f *.o
```

Figure 11: Exemple d'un Makefile

Le fichier ci-dessus est relativement simple, afin de comprendre la syntaxe basique d'un Makefile. Des commandes bien plus avancées existent et les dépendances peuvent prendre la forme de « tous les fichiers se terminant en .o ».

À l'aide de Make, il est ainsi possible de compiler un programme en appelant uniquement la commande « make ». Ainsi, le fichier gérant l'intégration continue n'a pas besoin d'être réécrit en permanence pour que la suite de tests fonctionne encore. Uniquement le Makefile doit être modifié.

[15] https://www.gnu.org/software/make/

3.7 Installer Make

Pour installer le Make sur l'image Docker choisie précédemment (Ubuntu), il faut utiliser la commande suivante :

```
apt-get install -y make
```

Figure 12: Commande pour installer Make

Ainsi, il est possible d'utiliser Make.

3.8 Configurer un projet avec Make

Comme dit précédemment, un Makefile est nécessaire pour utiliser Make. Divers utilitaires génèrent directement un Makefile sur demande. Dans notre cas, STM23CUBEMx [3] est le logiciel utilisé pour générer un projet sur la carte d'évaluation. Cette option est définie dans la partie « Project Manager ➔ Project » :

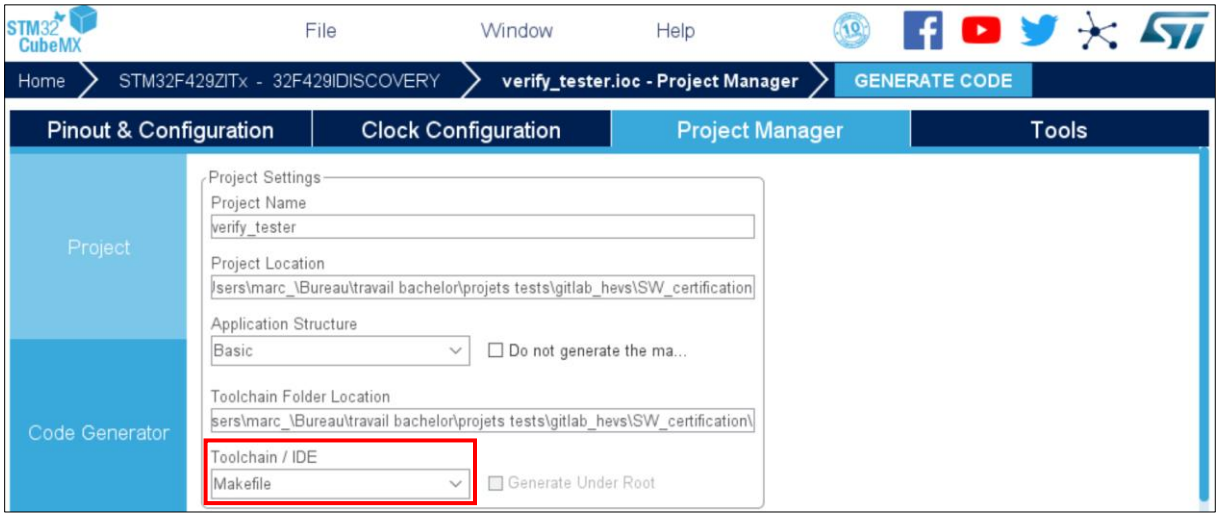


Figure 13: Définir la création d'un projet à base d'un Makefile depuis CubeMX

Afin de s'assurer que toutes les librairies sont disponibles pour le projet, il faut aller sous « Project Manager ➔ Code Generator » puis cocher « Copy all used libraries into the project folder » :

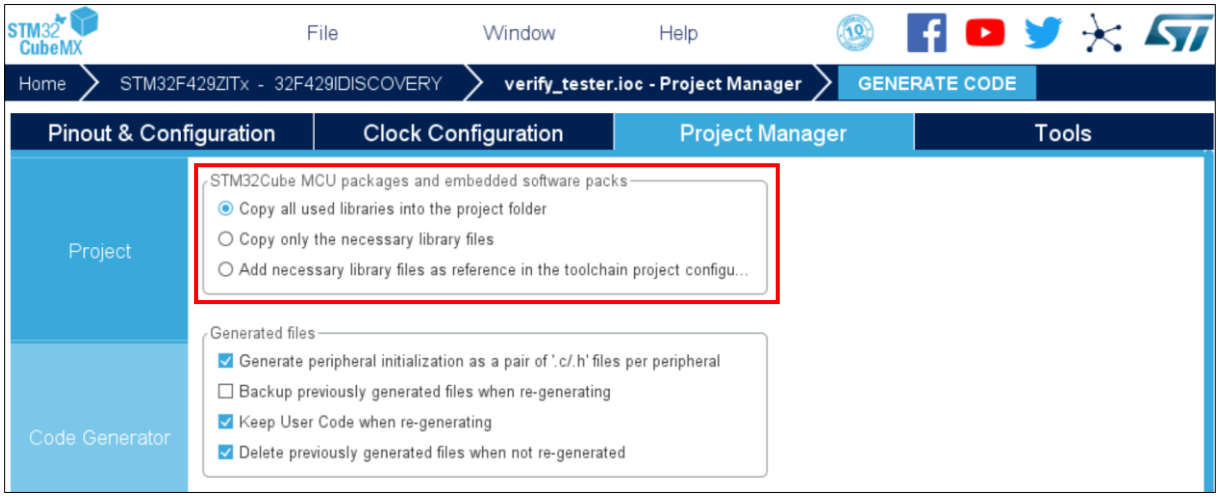


Figure 14: Inclure toutes les librairies dans le project depuis CubeMX

3.9 Configurer une image docker pour ARM

Afin de pouvoir compiler et programmer du code pour le uC, il faut configurer l'image docker réalisant ces étapes. Dans ce but, deux images seront générées. L'une permettra de compiler le code, l'autre de l'insérer au sein du uC.

3.9.1 Image docker pour la compilation

Le compilateur utilisé étant gcc-arm-embedded, il faut le télécharger depuis sa source officielle. Pour ce faire, il faut ajouter un répertoire dans la bibliothèque d'installation d'Ubuntu, ce qui nécessite une commande provenant de software-properties-common. Ainsi, voici les commandes à entrer, qui sont enregistrées dans un script bash, afin de tout exécuter de manière simple par la suite :

Ajout de la librairie permettant d'ajouter des répertoires dans la bibliothèque d'installation d'Ubuntu :

```
apt-get install -y software-properties-common
```

Figure 15: Configuration de l'image pour gcc-arm-embedded i - partie 1

Ajout d'un répertoire dans la bibliothèque d'installation d'Ubuntu :

```
add-apt-repository -y ppa:team-gcc-arm-embedded/ppa
```

Figure 16: Configuration de l'image pour gcc-arm-embedded - partie 2

Mise à jour de la bibliothèque d'Ubuntu :

```
apt-get update
```

Figure 17 : Configuration de l'image pour gcc-arm-embedded - partie 3

Installation de gcc-arm-embedded officiel:

```
apt-get install -y gcc-arm-embedded
```

Figure 18: Configuration de l'image pour gcc-arm-embedded - partie 4

Il faut également installer le logiciel permettant d'exécuter les Makefile, soit make. Celui-ci peut être installé directement à la création de l'image, ne nécessitant pas de modification de configuration au sein de celle-ci :

```
RUN apt-get install -y make
```

Figure 19: Installation de make

3.9.2 Image docker pour l'écriture du programme dans le uC

Le linker-debugger qui va être utilisé est le même que celui utilisé avec System Workbench For STM32[4], soit ST-Link [5].

Celui-ci n'est malheureusement pas disponible au sein de la biliothèque de logiciels de base pour Ubuntu. En revanche, il est possible de l'installer via un répertoire Github[16]. De la sorte, il y a besoin de git. Toutefois, pour l'installer, il y a besoin de Make. Ainsi, voici les commandes à entrer dans le Dockerfile :

```
RUN apt-get install -y git make
RUN git clone https://github.com/texane/stlink stlink
```

Figure 20: Installation de ST-Link – partie 1

Pour communiquer avec la cible, il y a besoin de la librairie suivante : libusb.

Ainsi, pour l'installer, la commande suivante est entrée dans le Dockerfile :

```
RUN apt-get install -y libusb-1.0-0-dev
```

Figure 21: Installation de ST-Link - partie 2

De cette manière, l'image contient tout le nécessaire pour la configurer. Afin de réaliser ceci, un script bash est conçu, de sorte qu'il n'y ait besoin que de le lancer pour que la configuration se fasse.

Pour l'installation de ST-Link, il faut aller dans le répertoire de celui-ci et utiliser le logiciel Make pour générer la version « Release » du linker-debugger. À noter également que, pour installer stlink, CMake (version de Make orientée pour une cible connue) est nécessaire, la librairie étant codée en C++. Voici donc les bases nécessaires :

```
apt-get install -y cmake
cd stlink && make release
```

Figure 22: Configuration de l'image pour ST-Link - partie 1

Ensuite, il faut se déplacer dans le répertoire nouvellement créé « build » puis « Release » et utiliser Make pour l'installation :

```
cd build/Release && make install
```

Figure 23: Configuration de l'image pour ST-Link - partie 2

Avec ceci, l'installation et la configuration sont réalisées. Cependant, la commande suivante est à entrer à chaque utilisation de l'image. En effet, les chemins ajoutés dans les librairies ne sont pas sauvegardés dans l'image docker. Ceci permet d'utiliser les librairies de ST-Link nouvellement installées :

```
export LD_LIBRARY_PATH=/usr/local/lib
```

Figure 24: Configuration de l'image pour ST-Link - partie 3

[16] https://github.com/texane/stlink

Il faut toutefois lancer l'image avec un paramètre particulier. En effet, pour programmer la cible, il faut utiliser un port USB. Comme Docker génère un environnement clos, il n'est, par défaut, pas possible de se connecter à un de ces ports. Pour ce faire, il faut modifier le fichier contenant les paramètres d'exécution des Runners de Gitlab se nommant « config.toml ». Pour plus d'informations sur ce fichier, se référer à l'annexe 4 chapitre 7.

Ainsi, il faut ajouter le paramètre « devices » configuré de la sorte :

```
[runners.docker]
  tls_verify = false
  image = "MarcBerguerand/example_image:v1.0"
  privileged = false
  devices = ["/dev/bus"]
  disable_entrypoint_overwrite = false
  oom_kill_disable = false
  disable_cache = false
  volumes = ["/cache"]
  shm_size = 0
```

Figure 25: Ajout du paramètre "devices" pour lancer l'image Docker

S'il est souhaité de lancer l'image sans utiliser l'intégration continue (en dehors du contexte des Runers), il faut alors utiliser la commande suivante, avec le nom de l'image et la version appropriée :

```
docker run --device=/dev/bus -it Docker_image:v1.0
```

Figure 26: Utilisation de l'image Docker hors contexte du runner

3.10 Programmer l'appareil simulant l'environnement externe

Comme montré dans le récapitulatif des appareils (Figure 6), il est possible de programmer l'appareil à l'aide d'un langage de programmation (C++/Python/VB). Python est choisi, car il s'agit d'un langage très courant et simple à prendre en main, tout en ne nécessitant pas de grandes connaissances dans l'exécution d'un script.

Digilent offre un Software Development Kit (SDK) [6] très complèt pour prendre en main l'appareil. La documentation complète relatant cellui-ci est disponible sur le site du fabricant.

Voici toutefois les commandes de base pour se relier à l'appareil via le script python :

3.10.1 Charger la librairie Digilent Waveforms

Il faut tout d'abord charger la librairie. Celle-ci étant dépendante du système d'exploitation, Digilent fourni le code permettant de le faire :

```
if sys.platform.startswith("win"):
    dwf = cdll.dwf
elif sys.platform.startswith("darwin"):
    dwf = cdll.LoadLibrary("/Library/Frameworks/dwf.framework/dwf")
else:
    dwf = cdll.LoadLibrary("libdwf.so")
```

Figure 27: Chargement de la libraire Waveforms en Python

3.10.2 Connexion à un Analog Discovery 2

Pour se connecter à un Analog Discovery 2 (AD2), il faut utiliser la méthode « FDwfDeviceOpen ». Celle-ci peut être utilisée de deux manières possibles, la première étant de se lier au premier appareil disponible, la seconde de se connecter à un appareil spécifique.

3.10.3 Connexion au premier appareil disponible

Pour ce faire, il faut utiliser le paramètre « c_int(-1) » :

```
hdwf = c_int()

# open device
print("Opening first device...")
dwf.FDwfDeviceOpen(c_int(-1), byref(hdwf))

if hdwf.value == hdwfNone.value:
    raise ValueError("failed to open device")

dwf.FDwfDeviceReset(hdwf)
```

Figure 28: Connexion au premier appareil disponible

3.10.4 Connexion à un appareil spécifique

Pour pouvoir se connecter à un appareil souhaité, il faut d'abord savoir le nombre d'appareils disponibles, à l'aide du code suivant :

```
# enumerate connected devices
dwf.FDwfEnum(c_int(0), byref(cDevice))
print("Number of Devices: "+str(cDevice.value))
```

Figure 29: Obtenir le nombre d'appareils connectés

Il est alors possible de parcourir chacun de ces appareils pour savoir s'il s'agit de celui voulu. Afin de se connecter à un seul appareil, un identifiant (numéro sériel en l'occurrence) est utilisé. Celui-ci se trouve

au dos de l'appareil. À noter qu'au dos de l'appareil, deux lettres supplémentaires sont ajoutées (« DA »), pour stipuler qu'il s'agit d'un produit Digilent Analog. Ces deux caractères ne sont pas contenus dans le numéro de série interne de l'appareil.

Pour obtenir l'identifiant, il faut utiliser la méthode « FDwfEnumSN » :

```
# declare string variables
serialnum = create_string_buffer(16)

# check devices
for iDevice in range(0, cDevice.value):
    dwf.FDwfEnumSN(c_int(iDevice), serialnum)

    if 'A2136' in str(serialnum.value):
        digilentAnalog1 = iDevice

    elif '7FBCB' in str(serialnum.value):
        digilentAnalog2 = iDevice
```

Figure 30: Vérification des appareils connectés

Cette méthode permet d'obtenir le numéro d'identifiant de l'appareil. Celui-ci est utilisé alors pour se lier à l'appareil souhaité, avec la méthode « FDwfDeviceOpen », auquel cas il est défini :

```
#open device 1
if 'digilentAnalog1' in locals():
    print("Opening device 1...")
    dwf.FDwfDeviceOpen(digilentAnalog1, byref(hdwf_analyzer1))

    if hdwf_analyzer1.value == hdwfNone.value:
        print("failed to open device")
        quit()

#open device 2
if 'digilentAnalog2' in locals():
    print("Opening device 2...")
    dwf.FDwfDeviceOpen(digilentAnalog2, byref(hdwf_analyzer2))

    if hdwf_analyzer2.value == hdwfNone.value:
        print("failed to open device")
        quit()
```

Figure 31: Connexion avec un appareil spécifique

Avec ceci, deux appareils sont connectés. En effet, il est possible d'en utiliser plusieurs en parallèle, ce qui permet d'augmenter le nombre de pins pouvant être testées.

3.11 Configurer une image docker pour Analog Discovery 2

Pour pouvoir utiliser l'appareil, les librairies de Digilent sont nécessaires. Le plus simple pour disposer de celles-ci est d'installer le programme complet.

Ainsi, il faut installer Digilent Waveforms, qui est le programme principal permettant d'utiliser l'AD2. Cependant, celui-ci est dépendant d'un autre logiciel : Digilent Adept Runtime, et a besoin d'une librairie pour communiquer avec l'appareil : libusb.

Ainsi, pour installer libusb, la commande suivante est entrée dans le Dockerfile :

```
RUN apt-get install -y libusb-1.0-0-dev
```

Figure 32: Installation des logiciels Digilent - partie 1

Ces deux programmes ne sont pas inclus dans la base de données des logiciels Ubuntu. Ainsi, il faut installer les paquets debian afin de pouvoir les utiliser. En revanche, libusb est incluse dans celle-ci.

Pour simplifier l'installation, les paquets sont déjà téléchargés sur la plateforme qui va générer l'image Docker. Ceux-ci sont à disposition sur le site du fabricant, pour Waveforms[17] [7] et pour Runtime[18] [8].

Ensuite, les deux paquets doivent se retrouver dans le même répertoire que le Dockerfile qui va être utilisé pour générer l'image.

Puis, il faut installer ceux-ci. Pour réaliser ceci, il faut d'abord les copier à l'aide du mot clé COPY (veuillez prêter attention aux XXXX, qui doivent correspondre aux versions des paquets qui seront copiés) :

```
COPY digilent.adept.runtime_XXXX.deb .
COPY digilent.waveforms_XXXXX.deb .
```

Figure 33: Installation des logiciels Digilent - partie 2

Ainsi, les paquets sont disponibles sur l'image qui sera générée. Ils ne seront cependant pas automatiquement installés. Pour ce faire, il faut entrer des commandes au sein même de l'image. Pour rendre ceci plus aisé, un script bash est conçu. Au sein de celui-ci se trouve alors les commandes pour installer les deux paquets (veuillez prêter attention aux XXXX, qui doivent correspondre aux versions des paquets qui seront copiés) :

```
dpkg -i digilent.adept.runtime_XXXX.deb
dpkg -i digilent.waveforms_XXXXX.deb
```

Figure 34: Installation des logiciels Digilent - partie 3

Pour profiter de l'utilisation des scripts python, il faut également installer Python, qui n'est pas inclus dans la base de données standard d'Ubuntu. Il faut alors installer software-properties-common, qui va ensuite permettre l'installation de ceci dans le script bash conçu précédemment :

```
apt-get install -y software-properties-common
apt-get update
apt-get install -y python3
```

Figure 35: Installation de python

Avec ceci, l'image est prête à être utilisée à la suite de l'exécution du script bash dans l'image.

[17] https://store.digilentinc.com/waveforms-previously-waveforms-2015/
[18] https://store.digilentinc.com/digilent-adept-2-download-only/

Il faut toutefois lancer l'image avec un paramètre particulier. En effet, pour programmer l'appareil, il faut utiliser un port USB. Comme Docker génère un environnement clos, il n'est, par défaut, pas possible de se connecter à un de ces ports. Pour ce faire, il faut modifier le fichier contenant les paramètres d'exécution des Runners de Gitlab se nommant « config.toml ». Pour plus d'informations sur ce fichier, se référer à l'annexe 4 chapitre 7.

Ainsi, il faut ajouter le paramètre « devices » configuré de la sorte :

```
[runners.docker]
  tls_verify = false
  image = "MarcBerguerand/example_image:v1.0"
  privileged = false
  devices = ["/dev/bus"]
  disable_entrypoint_overwrite = false
  oom_kill_disable = false
  disable_cache = false
  volumes = ["/cache"]
  shm_size = 0
```

Figure 36: Ajout du paramètre "devices" pour lancer l'image Docker

S'il est souhaité de lancer l'image sans utiliser l'intégration continue (en dehors du contexte des Runers), il faut alors utiliser la commande suivante, avec le nom de l'image et la version appropriée :

```
docker run --device=/dev/bus -it Docker_image:v1.0
```

Figure 37: Utilisation de l'image Docker hors contexte du runner

3.12 Configuration des images docker finales pour ARM

Voici ci-dessous le choix effectué pour les caractéristiques de l'image. À noter qu'il y a deux versions pour cette image. La première est utilisée pour vérifier le fonctionnement de celle-ci avec un code en pure C ou C++. La seconde permet de programmer une cible ARM et l'appareil de tests.

Caractéristique	Version
Système d'exploitation	Ubuntu 18.04
Gestionnaire de source	Git 2.17.1
Compilateur ARM	Arm-gcc-none-eabi 7.3.1
Debugger	ST-Link V2 1.5.1
Interpréteur python	Python 3.6.8
Digilent WaveForms	3.10.9
Digilent Adept RunTime	2.19.2

Figure 38 : Configuration des trois premières images docker

Ainsi, il est aisé de vérifier le bon fonctionnement, étapes par étapes, des images. En effet, au début git est certifié avec un programme C++ standard pour la cible à compiler, qui va par la suite servir à effectuer les tests unitaires. Ensuite, le Debugger (ST-Link) pour programmer une carte d'évaluation et les outils nécessaires à la programmation de l'oscilloscope générateur/récepteur de signaux a lieu à l'aide de Python et des librairies nécessaires à son fonctionnement.

Ainsi, deux images sont configurées de la manière suivante :

Image numéro	Contenu
1	Ubuntu, git, gcc-arm-embedded, C++, Python 3
2	Ubuntu, git, Python 3, Waveforms, ST-Link V2

Figure 39: Configuration des trois dernières versions pour les images Docker

Celles-ci seront utilisées pour des étapes bien précises. L'image 1 servira à compiler le code pour la cible et sauvegardera ensuite les fichiers qui seront utilisés pour programmer la cible (extension .bin) pour les futures étapes. Elle permettra également de vérifier le bon fonctionnement logique du programme à l'aide de tests unitaires. Cette image peut être lancée plusieurs fois sur la même machine, de sorte que plusieurs développeurs puissent compiler leur code en parallèle :

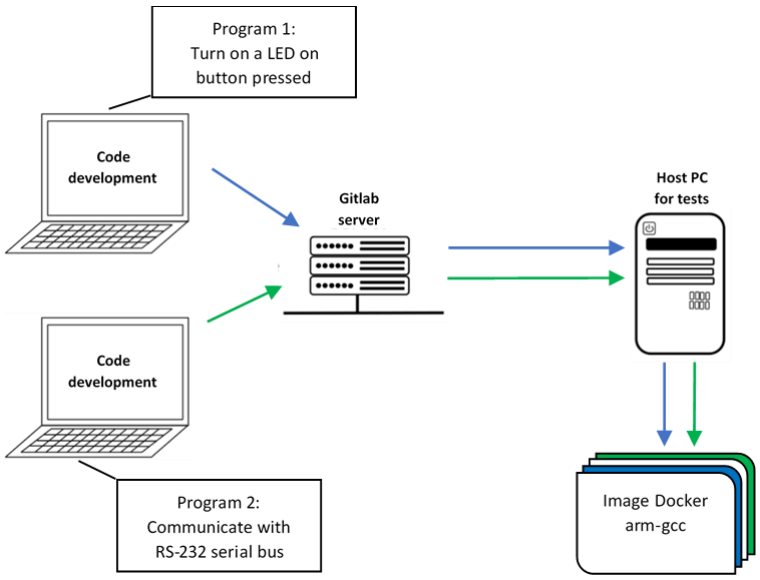


Figure 40: Exemple d'utilisation pour plusieurs développeurs

L'image 2 sera utile pour programmer la cible, à l'aide des fichiers générés et pour programmer l'appareil générant et mesurant les signaux sur la cible. En effet, il n'y a pas une image propre pour chaque appareil au cas où plusieurs développeurs voudraient lancer des tests en même temps. Voici un exemple pour ce mauvais cas d'utilisation :

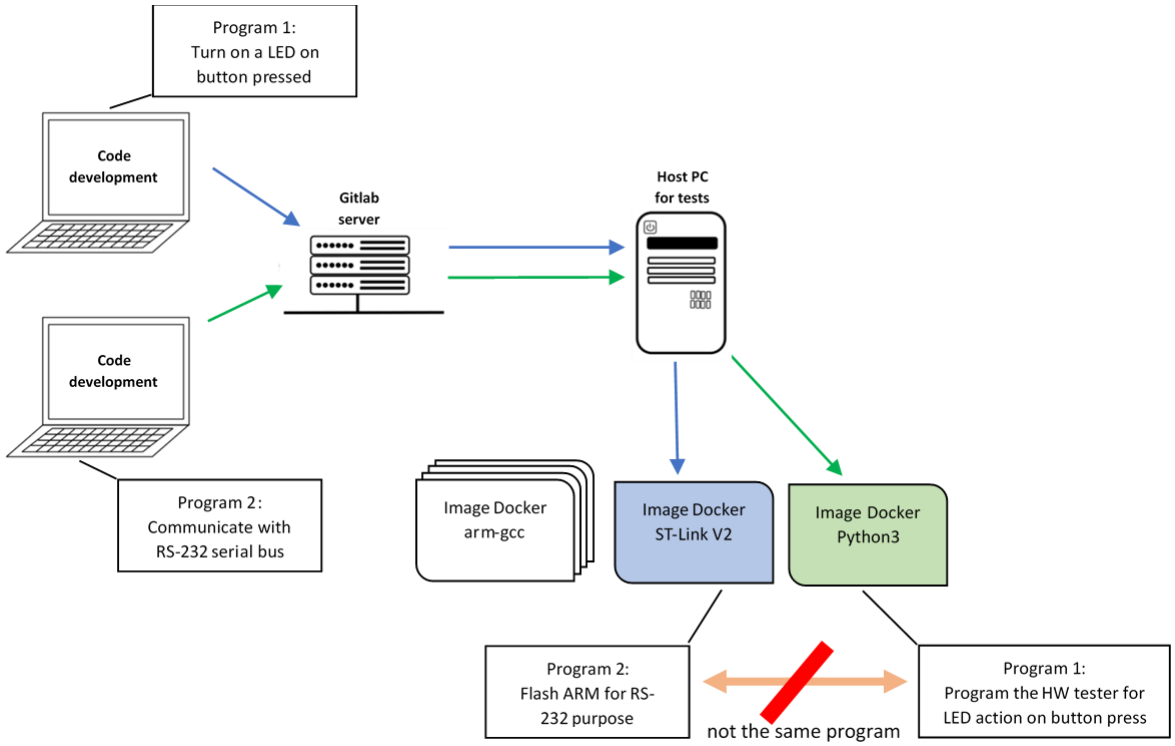


Figure 41: Mauvaise utilisation de multiples images Docker ayant un lien entre elles

Dans l'exemple ci-dessus, les deux développeurs peuvent toujours compiler leur code en parallèle. En revanche, comme les images pour programmer la cible et l'appareil de tests sont différentes, il est possible que le programme numéro 1 aille configurer l'appareil de tests et le programme numéro 2 aille programmer la cible. Ceci est à proscrire.

C'est pourquoi il n'y a que deux images distinctes. Une pour compiler le code (qui peut donc être lancée plusieurs fois en parallèle) et une pour programmer la cible et l'appareil de tests (qui ne peut exister qu'à un seul exemplaire) :

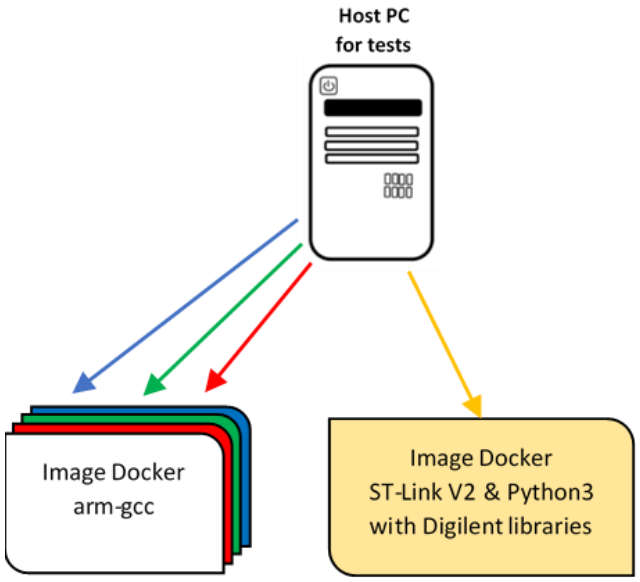


Figure 42: Représentation des images Dockers sur la machine hôte

Ces images sont générées pour permettre à un maximum de développeurs de compiler leur code. En effet, de cette manière il n'y a pas besoin d'attendre que l'un d'entre eux ait fini entièrement son pipeline de tests pour pouvoir compiler le code et se rendre compte qu'il y ait une erreur dans celui-ci. En revanche, si la compilation est un succès, le pipeline de tests sera mis en pause jusqu'à ce que le container exécutant la seconde image soit disponible.

Voici un graphe représentant la suite de tests effectuées et les versions des images utilisées pour chaque étape :

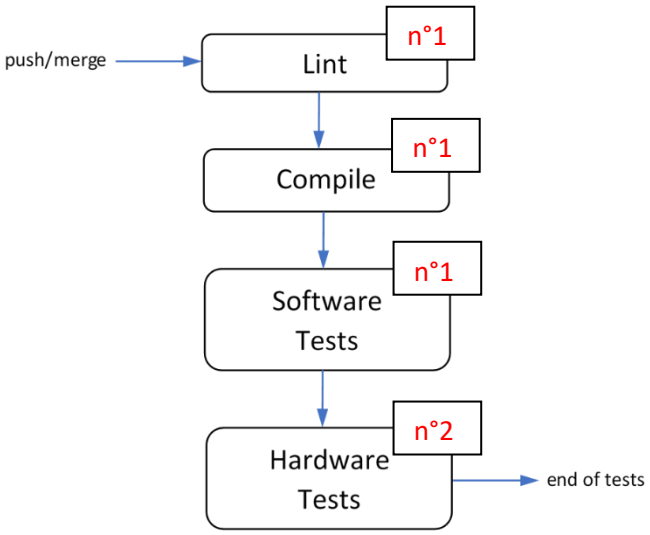


Figure 43: Version des images docker pour chaque étape

3.13 Exemple d'un projet avec CI « On Target Testing »

Maintenant que la configuration est définie, un projet type est réalisé, en utilisant toutes les caractéristiques du gestionnaire de sources. Ainsi, un projet « certification » est créé sur le serveur Gitlab :

Figure 44: Création d'un répertoire sur le serveur Gitlab

3.13.1 Définition d'une fonctionnalité du microcontroleur

La première fonctionnalité concerne des portes logiques, qui permettent de vérifier simplement le comportement du uC. Ainsi, la fonction logique suivante est implémentée :

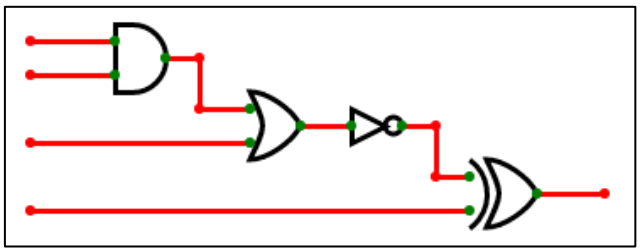


Figure 45: Fonction logique implémentée

Afin de concevoir ceci en représentant un travail d'équipe, une branche a été générée, se nommant « feature/add_logic_gates » :

Figure 46: Création d'une branche pour le développement du code réalisant les fonctions logiques

Partons du principe que la plaque de développement n'est pas encore disponible, soit parce que la commande a du retard, soit parce que l'équipe de développement travaille encore dessus. Ainsi, il n'est pour l'instant pas possible de concevoir du code pour la cible.
Néanmoins, il est possible de développer les fonctionnalités logiques du code. En effet, pour effectuer des tests unitaires, il n'est pas nécessaire d'avoir la cible à disposition, grâce à la méthode du Test-Driven Development (TDD).

3.13.2 Développement de la fonctionnalité logique du code à l'aide du TDD

Le TDD permet de se rendre compte d'éventuels problèmes logiques dans le code. Il permet également d'effectuer des tests récursifs. Ainsi, lorsque de nouvelles fonctionnalités sont implémentées, les anciennes sont constamment testées afin de s'assurer que l'ajout de code ne perturbe pas l'ancien comportement du logiciel. Une manière d'effectuer le TDD pour des systèmes embarqués est décrite dans le livre « Test-Driven Development for Embedded C »[19]. Voici un aperçu du rendu pour les fonctions logiques décrites précédemment :

D'abord, un fichier source et son header son créés :

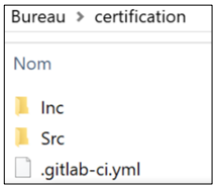


Figure 47: Dossier principale

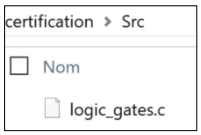


Figure 48: Dossier Src

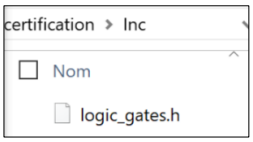


Figure 49: Dossier Inc

Ainsi, les fichier « logic_gates.c » et « logic_gates.h » permettront d'effectuer les modifications logiques en fonctions des différentes portes logiques implémentées.

Afin d'effectuer le TDD, un framework de test est utilisé. Le choix d'Unity est fait, car il s'agit d'un framework simple à utiliser. Voici comment l'implémenter :

Tout d'abord, le télécharger depuis sa source [9]. Puis, copier/coller les répertoires suivants :

Sous « Unity-master », aller dans le répertoire « src » :

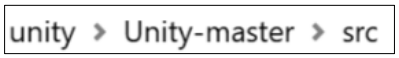


Figure 50: Location du répertoire src dans unity

Au sein de celui-ci se trouvent trois fichiers à copier :

- unity.c
- unity.h
- unity_internals.h

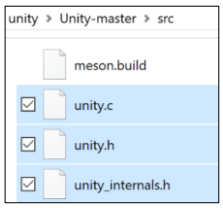


Figure 51: Fichiers source d'Unity à copier

Insérer ces fichiers au sein d'un répertoire « unity » créé dans le répertoire de travail :

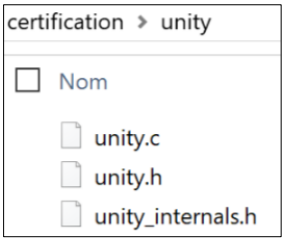


Figure 52: Insertion des fichiers dans le répertoire "unity"

[19] GRENNING, James. *Test-Driven Development for embedded C.* The Pragmatic Programmers, 2011.

Ceci permet d'utiliser des fonctions basiques d'Unity. Cependant, d'autres possibilités sont offertes avec le rajout d'une « fixture ». Ainsi, ceci est rajouté dans le répertoire unity du projet. Voici comment retrouver ces fichiers :

Aller sous « Unity-master » puis dans le répertoire « extras » se trouve un dossier « fixture » contenant les fichiers nécessaires :

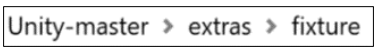


Figure 53: Location des fichiers "fixtures"

Au sein de celui-ci se trouve un répertoire « src ». Tous les fichiers qui s'y trouvent sont alors copiés pour être collés dans le répertoire « unity » du projet :

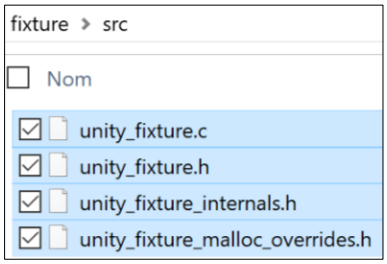


Figure 54: Fichiers devant être copiés dans le répertoire "unity" du projet

Ainsi, voici le contenu du répertoire « unity » du projet :

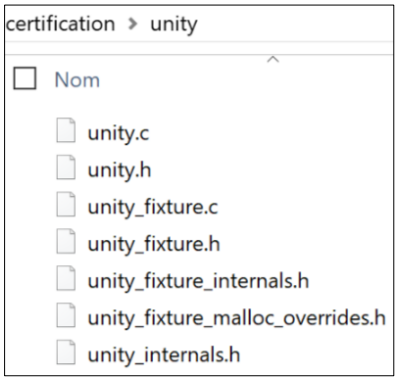


Figure 55: Contenu du répertoire "unity" du projet

Dès lors, le programme de test peut être généré. Afin de garder une architecture propre au sein du projet, un répertoire « tests » est conçu. Étant donné que nous savons d'ores et déjà que plusieurs tests auront lieu pour ce projet, des sous-répertoires sont générés. En effet, l'AD2 est utilisé via un script python, alors qu'Unity est du code en C++. Ainsi, un sous-répertoire « Src » est créé pour Unity et un autre « Py » est généré pour l'AD2 :

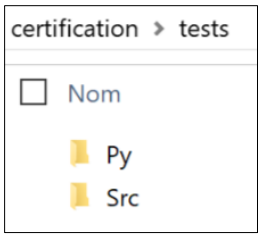


Figure 56: Architecture du projet pour les tests

Ainsi, le programme de tests TDD peut dès lors commencer. Dans le framework Unity, plusieurs possibilités sont offertes :

- Effectuer des tests unitaires dans le programme principal (main)
- Utiliser des groupements de tests exécutant des tests unitaires dans le programme principal (main)
- Utiliser des « runners » pour chaque groupement de tests, permettant de fluidifier la lisibilité des tests

Cette dernière option est choisie, car il s'agit de la manière la plus simple et visuelle possible. Ainsi, un fichier « logicTestRunner.c » est généré. Voici comment procéder :

1) Inclure le fichier « unity_fixture.h »
2) Générer la fonction principale (main)
3) Appeler le main d'Unity au sein de la fonction principale

Ainsi, voici le contenu actuel du fichier :

```
#include "unity_fixture.h"

int main(int argc, const char * argv[])
{
    return UnityMain(argc, argv, RunAllTests);
}
```

Figure 57: Contenu basique du fichier "logicTestRunner.c"

Le main d'Unity va alors rechercher une fonction dont le nom est « RunAllTests ». Il faut alors générer ladite fonction :

```
static void RunAllTests(void)
{
    RUN_TEST_GROUP(LogicGates);
}
```

Figure 58: Méthode appelée par le main d'Unity

Comme il est montré dans la Figure 58, une macro est dès lors appelée. Celle-ci va alors effectuer tous les tests contenus dans le groupe de tests « LogicGates ». Pour les réaliser, il faut utiliser une macro prédéfinie : « TEST_GROUP_RUNNER » :

```
TEST_GROUP_RUNNER(LogicGates)
{
    RUN_TEST_CASE(LogicGates, Inverter_0);
}
```

Figure 59: Réalisation d'un groupe de tests

Dans ce cas-ci, un seul test est effectué au sein du groupe « LogicGates », celui nommé « Inverter_0 ». Plus il y aura de fonctions logiques implémentées dans le code, plus le groupe « LogicGates » se verra rempli.

Afin de découpler au maximum tous les différents tests, chaque groupe aura son propre fichier source dédié. Ainsi, un fichier « logicGatesTest.c » est généré. Voici le contenu de base de celui-ci :

- Inclusion du fichier « unity_fixture.h »
- Inclusion du header dont les méthodes vont être testées (dans ce cas « logic_gates.h »)
- Inclusion des types standars int (« stdint.h »)
- Définition du groupe auquel le fichier est destiné à l'aide de la macro « TEST_GROUP »

Voici donc son contenu actuel :

```
#include "unity_fixture.h"
#include "logic_gates.h"
#include <stdint.h>

TEST_GROUP(LogicGates);
```

Figure 60: Fichier de base pour le groupe de tests logique

Ceci signifie donc que les tests appelés dans le fichier « logicTestRunner.c » seront définis dans ce fichier pour le groupe « LogicGates ».
Dès lors, le premier test peut être réalisé. Il est déjà préconfiguré dans la Figure 59 : Il s'agira de l'inversion d'un 0 logique. Ainsi, voici la description du premier test :

```
TEST(LogicGates, Inverter_0)
{
    value_0 = 0;
    logic_not(&res, &value_0);
    TEST_ASSERT_EQUAL_UINT16(0, res);
}
```

Figure 61: Réalisation du premier test

Veuillez prêter attention au résultat du test. Après l'inversion de la valeur zéro, il est attendu que le résultat soit égal à zéro, or cela est faux. Ceci est normal. Lorsqu'un test est effectué pour la première fois, le résultat attendu doit être faux, afin de s'assurer que ce cas-ci est traité.

La macro « TEST » est appelée pour en générer un, dans le contexte du groupe « LogicGates » et plus particulièrement pour le test « Inverter_0 » (en fonction des paramètres suivant la macro).

À ce stade, les variables, tout comme la fonction « logic_not » ne sont pas définies, ni configurées de base. Il faut alors les déclarer. Pour le fichier « logicGatesTest.c », les variables sont définies comme globales, et sont paramétrées au lancement de chaque test. Voici comment réaliser ceci :

```
static uint8_t res;
static uint8_t value_0;

TEST_SETUP(LogicGates)
{
    res = 0;
}

TEST_TEAR_DOWN(LogicGates)
{
}
```

Figure 62: Définition des variables et initialisation de celles-ci

De cette manière, les variables sont définies et initialisées avant chaque tests, grâce à la macro « TEST_SETUP ». Il est également possible d'effectuer des actions lors de la fin des tests, à l'aide de la macro « TEST_TEAR_DOWN ».

Il est important de noter la définition des variables. L'idéal est d'avoir un format se rapprochant le plus de celui interprété par la cible. Dans ce cas-ci, les variables sont définies comme entiers sur huit bits.

Dès lors, la conception de la méthode logique « logic_not » peut commencer. Ainsi, il faut écrire le prototype de la fonction dans le header « logic_gates.h » puis l'implémentation de celle-ci dans le fichier source « logic_gates.c » :

```
#ifndef __LOGIC_GATES_H
#define __LOGIC_GATES_H

#ifdef __cplusplus
 extern "C" {
#endif

#include <stdint.h>

void logic_not(uint8_t* res, uint8_t* b0);

#ifdef __cplusplus
}
#endif

#endif
```

Figure 63: Fichier "logic_gates.h" après le prototype "logic_not"

```
#include "logic_gates.h"
#include <stdint.h>

void logic_not(uint8_t* res, uint8_t* b0)
{
    uint8_t bit_value = (*b0) & 1;

    *(res) = (!bit_value) & 1;
}
```

Figure 64: Fichier"logic_gates.c" après l'implémentation de la méthode "logic_not"

Ainsi, tous les fichiers sont prêts pour effectuer le premier test. Il faut alors compiler ceux-ci et les exécuter. Pour ce faire, un Makefile est utilisé. Voici le contenu de celui-ci :

```
GCC_SOURCES =  \
Src/logic_gates.c \
tests/Src/logicGatesTest.c \
tests/Src/logicTestRunner.c \
unity/unity.c \
unity/unity_fixture.c


GCC_H_INCLUDES = \
unity/unity.h \
unity/unity_internals.h \
unity/unity_fixture.h

GCC_INCLUDES = \
-IInc \
-Iunity

run_test:
    gcc $(GCC_H_INCLUDES) $(GCC_SOURCES) $(GCC_INCLUDES) -o check_software_behaviour
```

Figure 65: Contenu du Makefile pour les tests unitaires

Avec ce makefile, il est aisé d'ajouter des fichiers sources et headers au cas ou plusieurs fonctionnalités sont ajoutées par la suite.

Puis, le fichier « .gitlab-ci.yml » est complété. À ce stade, trois étapes peuvent déjà être réalisées :

- Lint
- Compilation
- Tests unitaires

Ainsi, le fichier « lint.py », qui va compter le nombre de caractères en commentaires est ajouté au projet.

Voici donc le contenu du fichier yaml :

```yaml
stages:
  - lint
  - compile
  - soft_unit_test

lint:
  tags:
    - python3
  stage: lint
  script:
    - python3 lint.py
  allow_failure : true

compilation:
  stage: compile
  tags:
    - gcc
  script:
    - make run_test
  artifacts:
    paths:
      - check_software_behaviour

software unit tests:
  stage: soft_unit_test
  tags:
    - gcc
  script:
    - ./check_software_behaviour
```

Figure 66: Contenu du fichier ".gitlab-ci.yml" pour les tests unitaires

Ainsi, un premier add, commit puis push pour la branche « feature/add_logic_gates » est fait. Voici le résultat du CI :

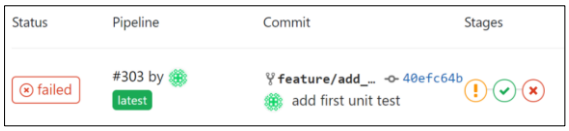


Figure 67: Résultat du CI pour le premier test unitaire

Comme montré dans la figure ci-dessus, le lint n'est pas validé. En effet, aucun commentaire n'a été entré dans le code, ce qui est d'ailleurs visible en cliquant sur cette étape (voir Figure 68 à la page suivante).

La compilation à l'aide du Makefile ne possède pas d'erreur. En revanche, l'exécution des tests unitaires s'avère fausse, ce qui est normal (voir le descriptif de la Figure 61). Ainsi, le premier CI est correct, il ne reste plus qu'à modifier le test unitaire pour avoir le vrai résultat attendu.

```
$ python3 lint.py
0
290
taux de commentaire : 0.0
Traceback (most recent call last):
  File "lint.py", line 98, in <module>
    raise ValueError("taux de commentaire trop faible");
ValueError: taux de commentaire trop faible
Authenticating with credentials from /home/marcberguera/.docker/config.json
ERROR: Job failed: exit code 1
```

Figure 68: Résultat du Lint pour le premier test unitaire

Ainsi, il y a en tout 290 caractères, dont aucun n'est en commentaire.

L'étape suivante est donc de modifier la condition du test unitaire à la bonne valeur souhaitée :

```
TEST(LogicGates, Inverter_0)
{
    value_0 = 0;
    logic_not(&res, &value_0);
    TEST_ASSERT_EQUAL_UINT16(1, res);
}
```

Figure 69: Test unitaire avec le bon résultat attendu

Une fois un nouvel envoi vers le serveur effectué, et le CI terminé, le résultat est le suivant :

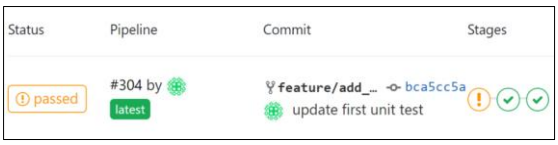


Figure 70: Résultat du CI après la mise à jour du premier test unitaire

Évidemment, aucun commentaire n'a été rajouté, et donc, le Lint est encore en échec. Néanmoins, le reste du CI est valide. Il faut dès lors compléter le fichier de tests en ajoutant un test après l'autre, afin d'avoir une certitude du bon fonctionnement du programme. Voici le fichier « logicGatesTest.c » une fois tous les tests passés :

```
TEST_GROUP_RUNNER(LogicGates)
{
    RUN_TEST_CASE(LogicGates, Inverter_0);
    RUN_TEST_CASE(LogicGates, Inverter_1);
    RUN_TEST_CASE(LogicGates, And_0);
    RUN_TEST_CASE(LogicGates, And_1);
    RUN_TEST_CASE(LogicGates, And_2);
    RUN_TEST_CASE(LogicGates, And_3);
    RUN_TEST_CASE(LogicGates, Or_0);
    RUN_TEST_CASE(LogicGates, Or_1);
    RUN_TEST_CASE(LogicGates, Or_2);
    RUN_TEST_CASE(LogicGates, Or_3);
    RUN_TEST_CASE(LogicGates, Xor_0);
    RUN_TEST_CASE(LogicGates, Xor_1);
    RUN_TEST_CASE(LogicGates, Xor_2);
    RUN_TEST_CASE(LogicGates, Xor_3);
}
```

Figure 71: Fichier "logicGatesTest.c" complété

Et voici le header « logic_gates.h » afin de s'apercevoir des fonctions implémentées dans le fichier source :

```
void logic_not(uint8_t* res, uint8_t* b0);
void logic_and(uint8_t* res, uint8_t* b0, uint8_t* b1);
void logic_or(uint8_t* res, uint8_t* b0, uint8_t* b1);
void logic_xor(uint8_t* res, uint8_t* b0, uint8_t* b1);
```

Figure 72: Prototypes des méthodes logiques

3.13.3 Ajout du hardware

Les tests unitaires sont dès lors approuvés. Il faut cependant relier le matériel physique au comportement logique. Pour ce faire, un projet est généré à l'aide de STM32CubeMX. Ainsi, les pins d'entrées et la pin de sortie sont configurées comme suit :

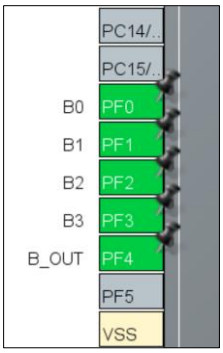


Figure 73: Utilisation des pins en entrées et en sortie

Avec les caractéristiques suivantes pour chaques pins :

Group By Peripherals

GPIO | SYS | NVIC

Search Signals

Search (Crtl+F) — Show only Modified Pins

Pin Name	Signal on Pin	GPIO output level	GPIO mode	GPIO Pull-up/Pu...	Maximum output speed	User Label	Modified
PF0	n/a	n/a	External Interrupt Mode with Rising/Falling edge trigger detection	No pull-up and n...	n/a	B0	☑
PF1	n/a	n/a	External Interrupt Mode with Rising/Falling edge trigger detection	No pull-up and n...	n/a	B1	☑
PF2	n/a	n/a	External Interrupt Mode with Rising/Falling edge trigger detection	No pull-up and n...	n/a	B2	☑
PF3	n/a	n/a	External Interrupt Mode with Rising/Falling edge trigger detection	No pull-up and n...	n/a	B3	☑
PF4	n/a	Low	Output Push Pull	No pull-up and n...	Very High	B_OUT	☑

Figure 74: Caractéristiques des GPIOs du uC

Tout en activant les interruptions pour les pins définies en entrée :

Group By Peripherals

GPIO | SYS | NVIC

NVIC Interrupt Table	Enabled
EXTI line0 interrupt	☑
EXTI line1 interrupt	☑
EXTI line2 interrupt	☑
EXTI line3 interrupt	☑

Figure 75: Activation des interruptions

Puis générer le projet comme indiqué précédemment au point 3.8. Attention toutefois lors de la génération de celui-ci. En effet, STM23CubeMX a de la peine à régénérer le Makefile. Il est préférable de le supprimer en concervant son contenu pour la partie des tests unitaires puis de générer à nouveau le projet.

3.13.4 Implémentation du code liant le hardware au comportement logique

Pour cet exemple, toutes les modifications se dérouleront au sein du fichier « main.c ».

Tout d'abord, il faut inclure le header permettant d'utiliser les fonctions logiques :

```
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "logic_gates.h"
/* USER CODE END Includes */
```

Figure 76: Ajout du header "logic_gates.h"

Ensuite, comme les entrées sont utilisées en mode interruption, la méthode callback ci-dessous est appelée.

```
/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{

}
/* USER CODE END 0 */
```

Figure 77: Méthode appelée lors d'une interruption

Puis, le code permettant d'effectuer le comportement souhaité est conçu. Dans un premier temps, la valeur de la pin qui a été modifiée est mise à jour, puis il est défini à quel niveau cette valeur a été changée, de manière à ne pas devoir effectuer toutes les fonctions logiques s'il n'y a que la dernière porte logique qui a vu une entrée indépendante se modifier. Ainsi, voici le code développé :

```
step_to_do = 0;
switch(GPIO_Pin)
{
    case B0_Pin:
        val_b0 = HAL_GPIO_ReadPin(B0_GPIO_Port, B0_Pin);
        step_to_do = 1;
        break;
    case B1_Pin:
        val_b1 = HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin);
        step_to_do = 1;
        break;
    case B2_Pin:
        val_b2 = HAL_GPIO_ReadPin(B2_GPIO_Port, B2_Pin);
        step_to_do = 2;
        break;
    case B3_Pin:
        val_b3 = HAL_GPIO_ReadPin(B3_GPIO_Port, B3_Pin);
        step_to_do = 3;
        break;
    default:
        break;
}
```

Figure 78: Définition de quelle pin a vu une interruption

Puis le code appelant les méthodes logiques et mettant à jour la pin de sortie est développé :

```
switch(step_to_do)
{
    case 1:
        logic_and(&res_and, &val_b0, &val_b1);
    case 2:
        logic_or(&res_or, &res_and, &val_b2);
        logic_not(&res_not, &res_or);
    case 3:
        logic_xor(&res_xor, &res_not, &val_b3);
        HAL_GPIO_WritePin(B_OUT_GPIO_Port, B_OUT_Pin, res_xor);
    default:
        break;
}
```

Figure 79: Utilisation des fonctions logiques et mise à jour de la sortie

Une fois ceci réalisé, les variables sont définies :

```
/* Private variables ---------------------------------------------------------*/
/* USER CODE BEGIN PV */
uint8_t step_to_do;

uint8_t val_b0;
uint8_t val_b1;
uint8_t val_b2;
uint8_t val_b3;

uint8_t res_and = 0;
uint8_t res_or = 0;
uint8_t res_not = 0;
uint8_t res_xor = 0;
/* USER CODE END PV */
```

Figure 80: Définition des variables utilisées

Ainsi, le code hardware est prêt. Toutefois, le Makefile permettant de compiler le code pour l'ARM ne contient pas le fichier source « test_gates.c ».

Il faut alors le rajouter (surligné dans la figure ci-dessous) :

```
#####################################
# source
#####################################
# C sources
C_SOURCES =  \
Src/main.c \
Src/gpio.c \
Src/usart.c \
Src/stm32f4xx_it.c \
Src/stm32f4xx_hal_msp.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_tim.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_tim_ex.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_rcc.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_rcc_ex.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_flash.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_flash_ex.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_flash_ramfunc.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_gpio.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_dma_ex.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_dma.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_pwr.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_pwr_ex.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_cortex.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_exti.c \
Src/logic_gates.c \
Src/system_stm32f4xx.c
```

Figure 81: Modification du Makefile

Avant de concevoir le fichier de tests hardware programmant l'AD2, le CI est effectué pour certifier que le code compile et qu'il est possible de programmer le uC. Ainsi, le fichier « .gitlab-ci.yml » est modifié de la manière suivante.

Un « stage » est ajouté :

```
stages:
  - lint
  - compile
  - soft_unit_test
  - hw_unit_test
```

Figure 82: Insertion d'une étape pour la programmation du uC

L'étape « compilation » voit son script modifié, de manière à exécuter autant la compilation du code pour l'ARM que pour les tests unitaires. Attention ici si le Makefile vient d'être généré par STM32CubeMX. Il est possible que le contenu de l'ancien fichier ait disparu. Dans ce cas, le rajouter à la fin dudit fichier.

```
compilation:
  stage: compile
  tags:
    - ARM make, gcc
  script:
    - make all BINPATH=/usr/bin
    - make run_test
  artifacts:
    paths:
      - build/verify_tester.bin
      - check_software_behaviour
```

Figure 83: Modification de l'étape pour la compilation

Puis l'étape rajoutée est définie :

```
HW unit test:
  stage: hw_unit_test
  tags:
    - ST-Link

  script:
    - st-flash write build/verify_tester.bin 0x8000000
```

Figure 84: définition de la nouvelle étape

Ainsi, le CI peut être effectué. S'il y a des erreurs pour la compilation, les observer et les corriger. S'il y a des erreurs à l'écriture du programme dans l'ARM, alors vérifier que la plaque de développement soit fonctionnelle et connectée à la machine physique. Si cela est le cas et qu'il y a toujours un problème, alors essayer de brancher directement la carte à un ordinateur pouvant lancer le programme (tel que Keil [10] ou SW4STM32 [4] par exemple). Si le problème persiste, alors vérifier si le Makefile généré est correct, ainsi que la commande « st-flash write ».

Voici le résultat du CI :

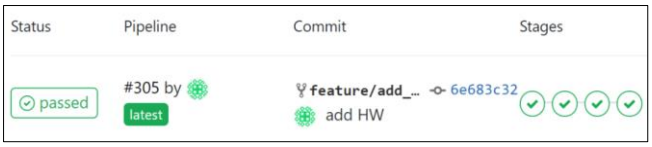


Figure 85: Résultat du CI avec le hardware

Cette fois-ci, les quatre étapes sont passées avec succès. En effet, STM32CubeMX génère un code très complet, et qui contient énormément de commentaires. Voilà pourquoi le Lint a passé alors qu'un commentaire supplémentaire n'a été rajouté. Quant à l'étape concernant le hardware, le résultat est le suivant, qui démontre que l'écriture du programme est un succès :

```
Flash page at addr: 0x08000000 erased
2019-08-13T10:58:16 INFO flash_loader.c: Successfully loaded flash loader in sram
enabling 32-bit flash writes
size: 6720
2019-08-13T10:58:16 INFO common.c: Starting verification of write complete
2019-08-13T10:58:16 INFO common.c: Flash written and verified! jolly good!
```

Figure 86: Résultat du CI pour le hardware

3.13.5 Ajout de l'appareil effectuant les tests

Il faut dès lors certifier que le uC réagit correctement dans son environnement. Ceci est analysé à l'aide de l'AD2.

Afin de commander l'AD2, un choix arbitraire a été fait : chaque fonctionnalité correspond à un module contenant une classe (dont le nom est encadré en rouge dans la figure ci-dessous), qui est exécuté en tant que thread. Ceci permet de fortement découpler les différentes options de l'AD2. Voici un schéma représentant les liens possibles :

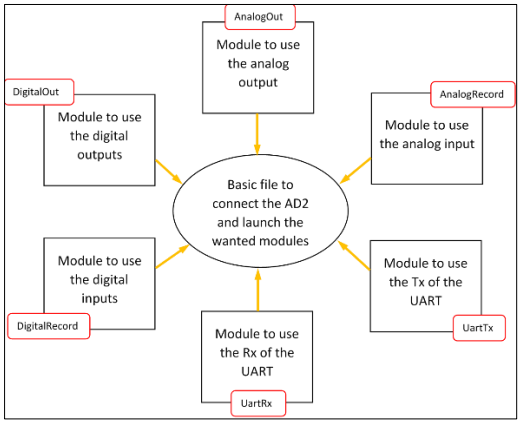


Figure 87: Modules actuellement disponibles permettant d'utiliser l'AD2

Ainsi, le fichier principal (nommé « digilent_analyzer_tester.py ») va uniquement importer les différents modules, établir une connexion avec l'appareil, puis va exécuter les divers modules en leur donnant accès à ladite connexion. Une fois les tests effectués, il va couper la connexion.

Voici donc le fichier python de base, sans utilisation des modules importés :

```
# basic import to work with Digilent libraries ----------
from ctypes import *
import time
from dwfconstants import *
import sys
#-----------------------------------------------------------
# specific import to work with the modules of the AD2 ----
from thread_analog_out import *
from thread_analog_record import *
from thread_digital_out import *
from thread_digital_record import *
from thread_uart_tx import *
from thread_uart_rx import *
#-----------------------------------------------------------
```

Figure 88: Importation des modules pour l'AD2

```
# load the Digilent libraries ---------------------------
if sys.platform.startswith("win"):
    dwf = cdll.dwf
elif sys.platform.startswith("darwin"):
    dwf = cdll.LoadLibrary("/Library/Frameworks/dwf.framework/dwf")
else:
    dwf = cdll.LoadLibrary("libdwf.so")
#-----------------------------------------------------------
```

Figure 89: Chargement de la librairie Digilent

C'est dans la partie ci-dessous que des modifications auront lieu. Les modules voulant être utilisés seront ajoutés dans la partie « here use the wanted modules » :

```
dwf.FDwfParamSet(DwfParamOnClose, c_int(0))  # 0 = run, 1 = stop, 2 = shutdown

hdwf = c_int()

# try to establish a connection ------------------------
print("Opening first device...")
dwf.FDwfDeviceOpen(c_int(-1), byref(hdwf))

if hdwf.value == hdwfNone.value:
    raise ValueError("failed to open device")
#----------------------------------------------------------

dwf.FDwfDeviceReset(hdwf)

# here use the wanted modules ---------------------------

#----------------------------------------------------------

dwf.FDwfDeviceClose(hdwf)
```

Figure 90: Code principal pour l'AD2

Ainsi, pour utiliser les entrées et sorties digitales, l'ajout suivant est effectué :

```
# here use the wanted modules ---------------------------
digital_out = DigitalOut(hdwf, dwf)
digital_record = DigitalRecord(hdwf, dwf)

digital_record.start()
digital_out.start()
#----------------------------------------------------------
```

Figure 91: Utilisation des modules entrée et sortie digitale

Comme il est constatable, la référence de l'AD2 (« hdwf ») et la librairie de Digilent (« dwf ») sont les deux paramètres nécessaires à l'utilisation de chaque module. Avec ceci, le code principal de l'AD2 est fonctionnel.

Pour mettre à jour les sorties digitales, le code ci-dessous est utilisé (dans le module « DigitalRecord »). L'adaptation des sorties se faite grâce à la méthode « FDwfDigitalIOOutputSet » :

```
for i in range(16):
    time.sleep(0.01)
    self.dwf.FDwfDigitalIOOutputSet(self.hdwf, c_int((i)<<4))
```

Figure 92: Code de l'AD2 pour les sorties digitales

Ainsi, les seize possibilités binaires sont testées.

Concernant la pin d'entrée de l'AD2 (en utilisant donc le module « DigitalRecord »), un fichier CSV est généré ayant les informations du temps relatif au lancement du code et de l'état des pins de l'appareil. L'acquisition des informations des pins de l'AD2 est basée sur le code fournit dans les exemples de Digilent. La ligne surlignée est celle qui va entrer les valeurs dans la liste nommée « rgwSamples ».

```
while cAvailable.value > 0:
    cSamples = cAvailable.value
    # we are using circular sample buffer, prevent overflow
    if iSample + cAvailable.value > self.nSamples:
        cSamples = self.nSamples - iSample
        self.dwf.FDwfDigitalInStatusData2(
            self.hdwf, byref(rgwSamples, 2*iSample), c_int(iBuffer), c_int(2*cSamples)
        )
        iBuffer += cSamples
        cAvailable.value -= cSamples
        iSample += cSamples
        iSample %= self.nSamples
```

Figure 93: Acquisition des données sur les pins digitales de l'AD2

Ici se trouve la création du fichier CSV :

```
f = open("record_digital.csv", "w")
f.write("time of acquisition[s], value[V]\n")
time_spent = 0
for v in rgwSamples:
    f.write("%.3f, %s\n" % (time_spent, v))
    time_spent += 1/self.freq_acq
f.close()
```

Figure 94: Création du fichier CSV pour les pins digitales de l'AD2

Ainsi, il est possible de vérifier que le fichier « record_digital.csv » existe en tant qu'artéfact dans le CI et qu'il a un format valide. Pour ce faire, il faut mettre à jour le fichier « .gitlab-ci.yml » de la sorte :

```
HW unit test:
  stage: hw_unit_test
  tags:
    - ST-Link
  script:
    - st-flash write build/verify_tester.bin 0x8000000
    - cd tests/Py
    - python3 digilent_analyzer_tester.py
  artifacts:
    paths:
      - tests/Py/record_digital.csv
```

Figure 95: Modification du fichier ".gitlab-ci.yml" pour inclure l'AD2

Et voici le résultat du CI :

Figure 96: Résultat du CI avec l'ajout de l'AD2

Avec comme artéfact le fichier « record_digital.csv » :

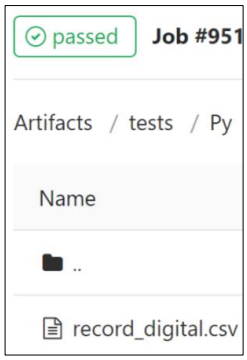


Figure 97: Artéfact sauvegardé dans le CI

```
time of acquisition[s], value[V]
0, 3
0.001, 3
0.002, 3
0.003, 3
0.004, 3
0.005, 3
0.006, 3
0.007, 3
0.008, 83
0.009, 83
0.010, 83
0.011, 83
0.012, 83
0.013, 83
0.014, 83
0.015, 83
0.016, 83
0.017, 83
0.018, 83
0.019, 83
0.020, 32771
0.021, 32771
0.022, 32771
0.023, 32771
0.024, 32771
0.025, 32771
```

Figure 98: Fichier CSV généré par l'AD2

3.13.6 Vérification du bon fonctionnement du uC

Une fois le fichier « record_digital.csv » généré, il faut l'analyser afin de certifier le bon fonctionnement de la cible. Pour ce faire, un autre fichier CSV est conçu. Ceci simplifiera la portabilité du programme vérifiant le bon fonctionnement de l'appareil. En effet, le nouveau CSV, nommé « in_out_digital.csv » est créé de la sorte :

```
mask output active:, 32768, end_of_line
0.025, 32771, 32768
0.036, 32787, 32768
0.044, 32803, 32768
```

Figure 99: CSV permettant de certifier le bon fonctionnement de la cible

Ainsi, la première ligne du fichier impose la base de travail de comparaison. En effet, le « mask output active » permettra de ne vérifier uniquement les valeurs des pins indiquées. Dans le cas ci-dessus, uniquement la seizième pine sera vérifiée, car 32768 équivaut à 2 à la puissance 16. À noter que la numérotation des pins de l'AD2 commence à partir de zéro, et donc il s'agit de la pin quinze.

Ensuite, les colonnes du fichier sont représentées de la sorte :

1e colonne	2e colonne	3e colonne
Temps relatif depuis le lancement du code où le changement de pin a lieu	Valeur des pins comprenant autant celles en entrées qu'en sorties	Quelle pin est censée être modifiée durant cette opération

Figure 100: Représentation du contenu du fichier CSV

Avec un fichier ayant cette configuration, il est facile de modifier les résultats attendus pour de futures développements.

Ainsi, le code permettant de certifier le bon fonctionnement des IO digitales se trouve dans le fichier « verification_digital.py ». Voici les explications de celui-ci ci-dessous.

Tout d'abord se fait la lecture des données au sein des deux fichiers CSV :

```python
import csv

# get CSV datas from "record_digital.csv" -------------
time_digital_mes = []
value_digital_mes = []

with open('record_digital.csv') as csvDataFile:
    csvReader = csv.reader(csvDataFile)
    for row in csvReader:
        time_digital_mes.append(row[0])
        value_digital_mes.append(row[1])
# -----------------------------------------------------
# get CSV datas from "in_out_digital.csv" -------------
time_digital_want = []
value_digital_want = []
channel_changed_want = []

with open('in_out_digital.csv') as csvDataFile:
    csvReader = csv.reader(csvDataFile)
    for row in csvReader:
        time_digital_want.append(row[0])
        value_digital_want.append(row[1])
        channel_changed_want.append(row[2])
# -----------------------------------------------------
```

Figure 101: Lecture des données concernant les fichiers CSV

Puis s'effectue la vérification des valeurs mesurées en fonction de celles souhaitées :

```
# get the mask value for the wanted pins
mask = int(value_digital_want[0])
# set the index of the wanted values at 1 to skip the first line
index_mes = 1

# check the content of the in_out_digital without the first line
for index_want in range(1, len(value_digital_want)):
    # only check the content of a pin with a wanted pin
    if (int(channel_changed_want[index_want]) & mask) == 0:
        index_want += 1
    else:
        # go to the next time value that's equal or greater of the wanted one
        while float(time_digital_mes[index_mes]) < float(time_digital_want[index_want]):
            index_mes += 1
        # then check if the measured value is the same as the wanted one
        if (int(value_digital_mes[index_mes]) & mask) ==
                                        (int(value_digital_want[index_want]) & mask):
            print('-----------success-----------')
        else:
            print('-----------error-----------')
            raise ValueError("measure wasn't expected")

        index_want += 1
```

Figure 102: Vérification des valeurs mesurées en fonction de celles souhaitées

Ainsi, le projet est prêt à être certifié dans son fonctionnement intégral, d'une fois que le fichier « .gitlab-ci.yml » est mis à jour de la sorte :

La dernière étape est rajoutée, celle analysant les résultats mesurés précédemment :

```
stages:
  - lint
  - compile
  - soft_unit_test
  - hw_unit_test
  - analyze_hw_unit_test
```

Figure 103: Rajout d'une étape dans le pipeline de tests

Puis le descriptif de l'étape est indiqué ci-dessous :

```
analyze HW unit test results:
  stage: analyze_hw_unit_test
  tags:
    - python3
  script:
    - cd tests/Py
    - python3 verification_digital.py
```

Figure 104: Description de l'étape supplémentaire pour vérifier le bon fonctionnement du uC

Ainsi, le résultat du pipeline de tests est le suivant :

Figure 105: Résultat du CI avec toutes les étapes

Avec comme indication pour la dernière étape :

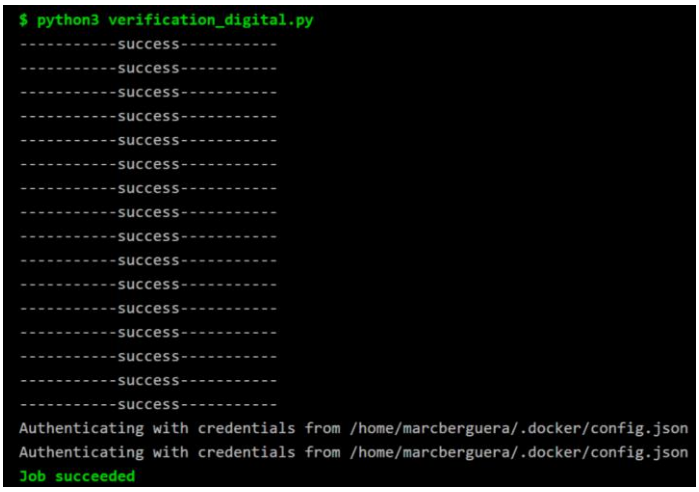


Figure 106: Résultat de la dernière étape, attestant le bon fonctionnement du uC

3.13.7 Fusion de la fonctionnalité et du code principal

Une fois le CI effectué complètement avec succès, un « merge request » est souhaité. Ceci se passe sur le serveur Gitlab. Il faut aller sur la branche avec la nouvelle fonctionnalité (« feature/add_logic_gates », en vert dans la figure ci-dessous), puis cliquer sur « Create merge request », en rouge ci-dessous :

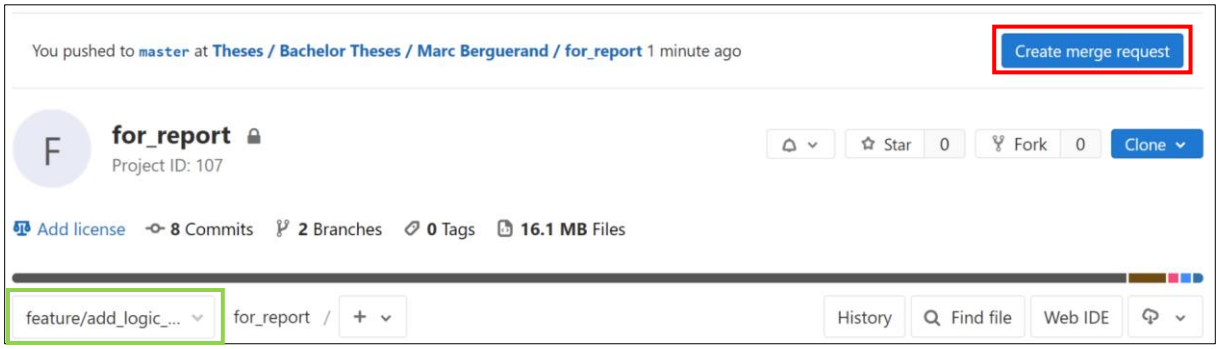


Figure 107: Création d'une merge request

Puis modifier les informations pour que cela soit compréhensible :

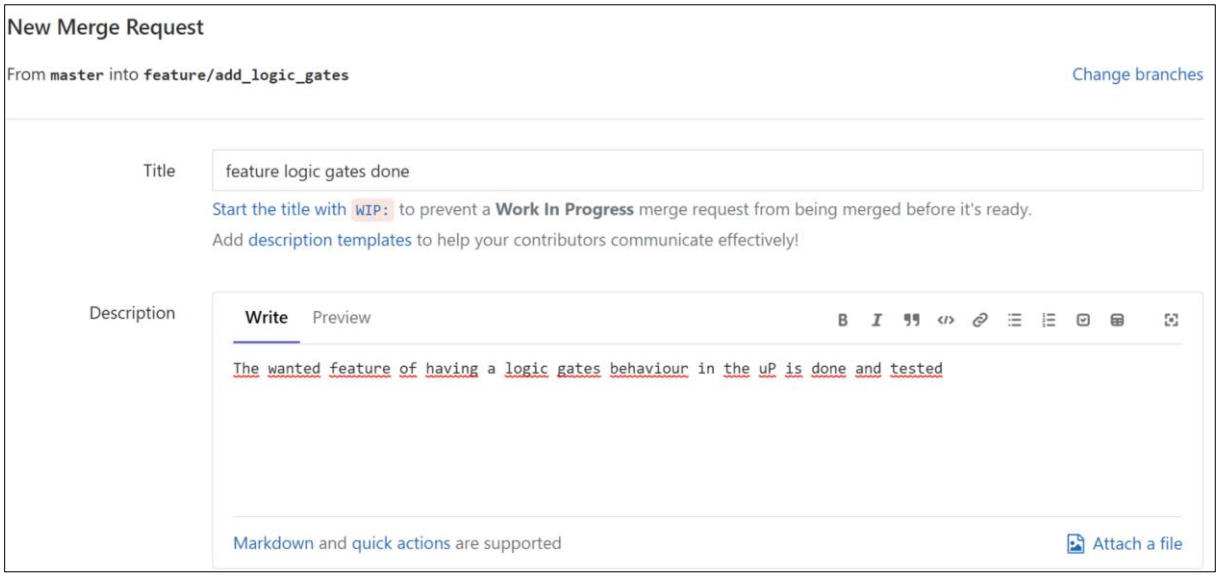


Figure 108: Modifications des champs remplis de base pour le merge request

Il s'en suit les informations concernant la personne requérant le merge request :

Figure 109: Informations sur le quémandaire du merge request

Il est également possible de demander à un collaborateur d'approuver le travail effectué :

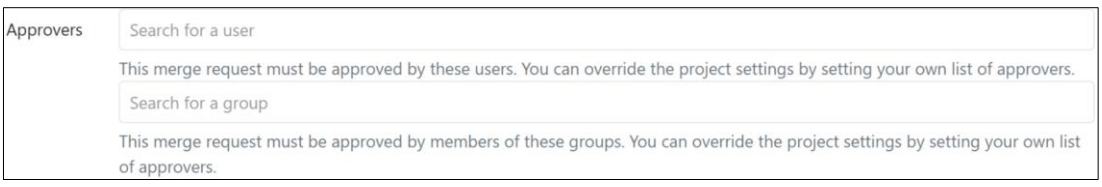


Figure 110: Approbation du travail par des collaborateurs

Puis se trouvent les informations propres au merge (la branche source, la branche cible et les options possible pour ce merge) :

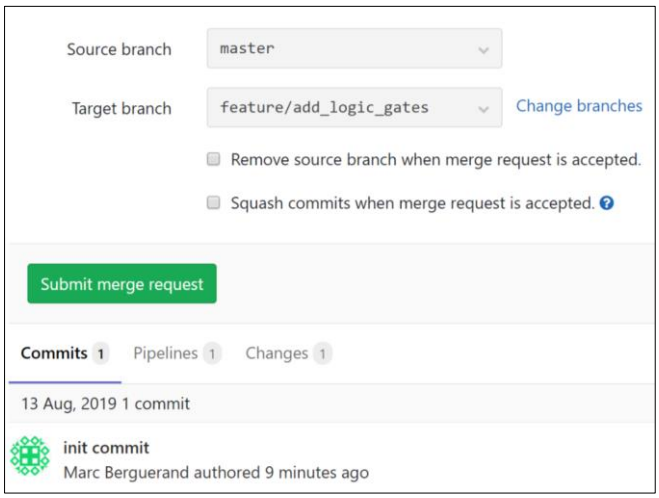


Figure 111: Gestion du merge

Auquel cas la branche source ou cible est incorrect, il faut cliquer sur « Change branches » qui affiche ceci :

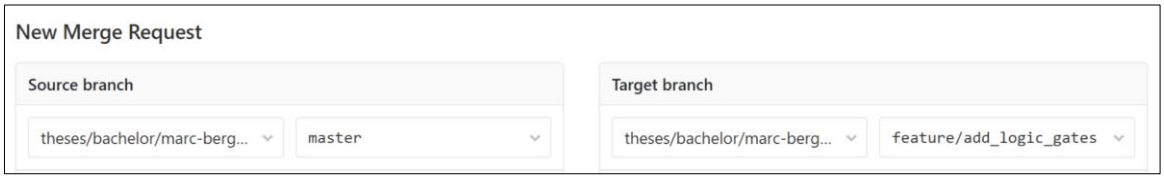


Figure 112: Modification des branches pour le merge

Ainsi, les branches peuvent être modifiées. Ensuite, un appui sur « Submit merge request » est fait dans la ci-dessusFigure 111, puis la fenêtre suivante apparait :

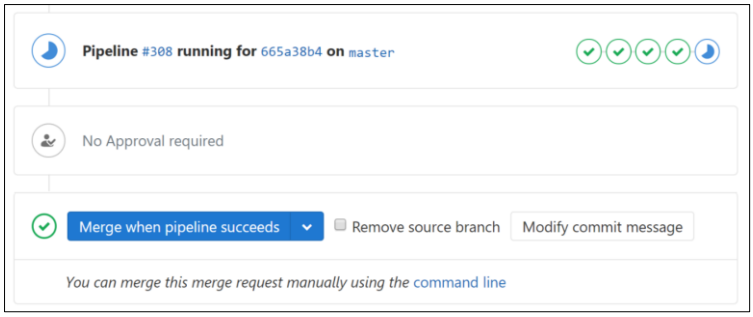


Figure 113: Options durant le merge

Le pipeline de test s'effectue alors. Une fois celui-ci réalisé, le merge s'effectue s'il est réussi. Il est également possible d'effectuer un merge sans attendre que le pipeline soit réalisé, en déroulant le menu entouré en rouge dans la figure ci-dessus :

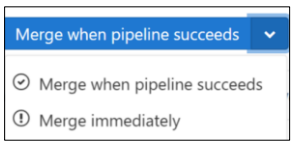


Figure 114: Option pour faire un merge immédiatement

3.13.8 Définition d'une fonctionnalité du microcontroleur

Cette fois-ci, une communication sérielle de type UART souhaite être utilisée. Son fonctionnement logique doit être le suivant :

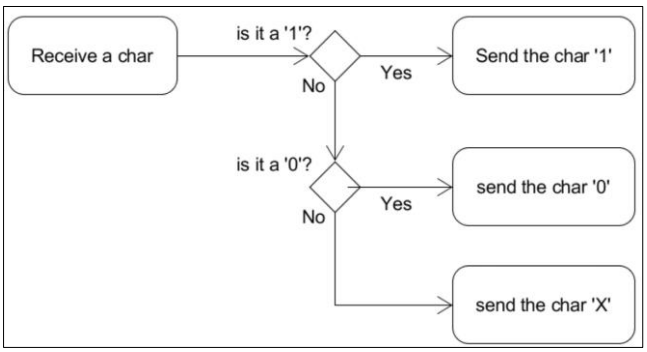


Figure 115: Comportement logique de l'UART

Afin de réaliser ceci, une nouvelle branche est générée : « feature/add_uart » :

Figure 116: Création de la nouvelle branche

Puis, les tests unitaires sont générés.

3.13.9 Développement des tests unitaires pour l'UART

Un nouveau fichier est généré dans le répertoire « tests/Src », se nommant « logicUartTest.c » :

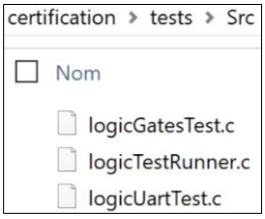


Figure 117:. Création d'un nouveau fichier de test

Tout d'abord, il faut modifier le fichier « logicTestRunner.c », afin de rajouter un groupe de tests pour la logique de l'UART :

```
static void RunAllTests(void)
{
    RUN_TEST_GROUP(LogicGates);
    RUN_TEST_GROUP(LogicUart);
}
```

Figure 118: Ajout d'un groupe de tests

Et ainsi définir le groupe de tests, avec, tout comme pour le premier groupe créé, un premier test concernant l'envoi d'une trame UART portant le caractère '1' :

```
TEST_GROUP_RUNNER(LogicUart)
{
    RUN_TEST_CASE(LogicUart, Send_1);
}
```

Figure 119: Définition du nouveau groupe de test

Puis le fichier « logicUartTest.c » est complété, de la même manière que pour « logicGatesTest.c » au point 3.13.2. À noter qu'ici également la vérification se fait avec une fausse valeur, afin de s'assurer de passer par ce test. En effet, lorsqu'un '1' est reçu, un '1' devrait être envoyé :

```c
#include "unity_fixture.h"
#include "logic_uart.h"
#include <stdint.h>

TEST_GROUP(LogicUart);
static char res;
static char value;

TEST_SETUP(LogicUart)
{
    res = ' ';
}

TEST_TEAR_DOWN(LogicUart)
{
}

TEST(LogicUart, Send_1)
{
    value = '1';
    check_reception(&res, &value);
    TEST_ASSERT_EQUAL_UINT8((uint8_t)'0', res);
}
```

Figure 120: Contenu du fichier "logicUartTest.c" pour le premier test unitaire

Ainsi, les fichiers « logic_uart.c » et « logic_uart.h » sont créés dans le répertoire « Src » du projet, avec comme contenu :

```c
#ifndef __LOGIC_UART_H
#define __LOGIC_UART_H

#ifdef __cplusplus
 extern "C" {
#endif

#include <stdint.h>

void check_reception(char * res, char * to_check);

#ifdef __cplusplus
}
#endif

#endif
```

Figure 121: Contenu du fichier "logic_uart.h"

```c
#include "logic_uart.h"

void check_reception(char * res, char * to_check)
{
    if ((*to_check) == '1')
    {
        *(res) = '1';
    }
    else
    {
        if ((*to_check) == '0')
        {
            (*res) = '0';
        }
        else
        {
            (*res) = 'X';
        }
    }
}
```

Figure 122: Contenu du fichier "logic_uart.c"

Ainsi les fichiers sources sont prêts à être testés. Il faut alors modifier le Makefile pour compiler le code. Comme il est possible de le constater, très peu de changements sont effectués par rapport au Makefile précédent. Il n'y a eu que l'insertion de « Src/logic_uart.c \ » et de « tests/Src/logicUartTest.c \ » dans le groupe « GCC_SOURCES », Le reste est identique.

```
GCC_SOURCES =  \
Src/logic_gates.c \
Src/logic_uart.c \
tests/Src/logicGatesTest.c \
tests/Src/logicUartTest.c \
tests/Src/logicTestRunner.c \
unity/unity.c \
unity/unity_fixture.c

GCC_H_INCLUDES = \
unity/unity.h \
unity/unity_internals.h \
unity/unity_fixture.h

GCC_INCLUDES = \
-IInc \
-Iunity

run_test:
    gcc $(GCC_H_INCLUDES) $(GCC_SOURCES) $(GCC_INCLUDES) -o check_software_behaviour
```

Figure 123: Contenu du Makefile

Ainsi, le projet peut être mis à jour sur le serveur, afin qu'un cycle de CI s'effectue et qu'il n'y ait un problème qu'avec les tests unitaires, étant donné que le résultat attendu est faux (ce qui est ce que l'on souhaite au premier passage dans un test). Ce qui est intéressant est qu'il n'y a, cette fois, pas d'erreur vis-à-vis du Lint. En effet, le code généré pour l'ARM étant toujours valable, les commentaires liés au code de celui-ci sont toujours comptés et donc le taux de commentaires est au-delà des cinquante pourcents. Voici un aperçu du pipeline de tests :

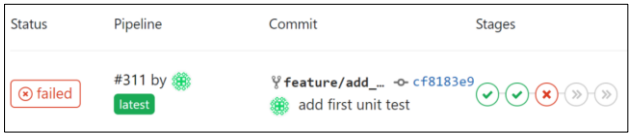


Figure 124: Aperçu du CI

Une fois tous les tests réalisés, le fichier « logicUartTest.c » contient les trois tests suivant :

```
TEST_GROUP_RUNNER(LogicUart)
{
    RUN_TEST_CASE(LogicUart, Send_1);
    RUN_TEST_CASE(LogicUart, Send_0);
    RUN_TEST_CASE(LogicUart, Send_Other);
}
```

Figure 125: Définition de tous les tests unitaires UART

3.13.10 Adaptation des configurations du uC

Le projet doit être régénéré afin de pouvoir profiter de l'UART sur l'ARM. Ceci est réalisé à l'aide du programme STM32CubeMX. Tout d'abord, il faut aller sélectionner la configuration UART5 et la définir en mode asynchrone. Ensuite il faut vérifier les paramètres pour les pins de sorties et configurer les options souhaitées pour la communication UART :

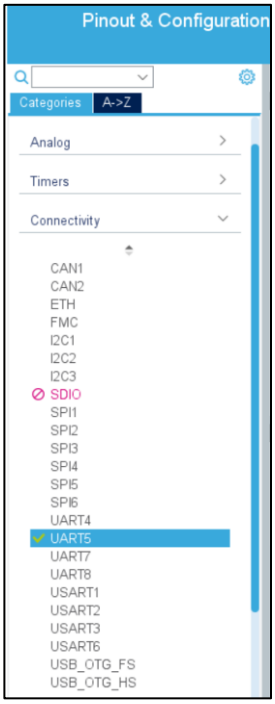


Figure 126: Sélection de l'UART

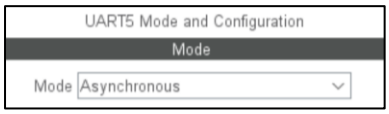


Figure 127: Définir en mode asychrone

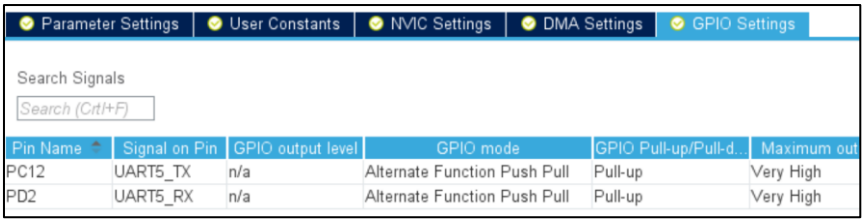


Figure 128: Vérifier les configurations des pins

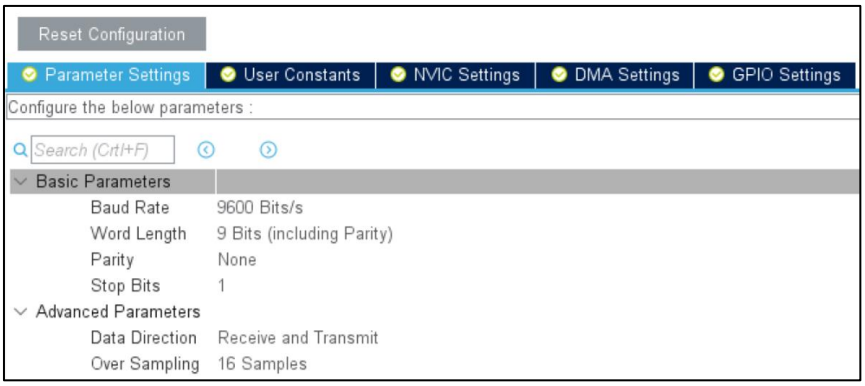


Figure 129: Configurer les options relatives à la communication UART

Il est souhaitable d'utilise une interruption pour la réception d'une trame UART. Il faut donc l'activer :

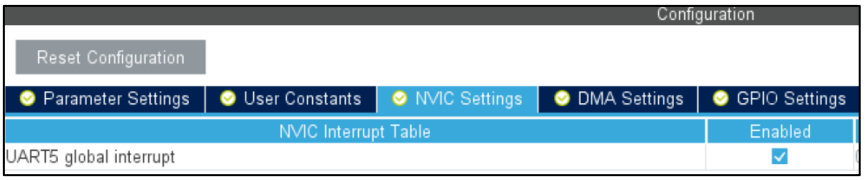


Figure 130: Utilisation de l'UART en mode interruption

Il faut alors générer le projet, attention encore au Makefile, il est préférable de sauvegarder la partie s'occupant des tests unitaires puis de supprimer le fichier avant de le régénérer et de rajouter la partie des tests unitaires, afin de s'assurer de partir sur de bonnes bases.

Comme précédemment, il faut rajouter dans le Makefile généré les fichiers sources que nous avons créés (« logic_gates.c » et « logic_uart.c » dans ce cas-ci), pour que le compilateur ARM y ait accès :

```
######################################
# source
######################################
# C sources
C_SOURCES =  \
Src/main.c \
Src/gpio.c \
Src/usart.c \
Src/stm32f4xx_it.c \
Src/stm32f4xx_hal_msp.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_tim.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_tim_ex.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_uart.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_rcc.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_rcc_ex.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_flash.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_flash_ex.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_flash_ramfunc.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_gpio.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_dma_ex.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_dma.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_pwr.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_pwr_ex.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_cortex.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal.c \
Drivers/STM32F4xx_HAL_Driver/Src/stm32f4xx_hal_exti.c \
Src/logic_gates.c \
Src/logic_uart.c \
Src/system_stm32f4xx.c
```

Figure 131: Mise à jour du Makefile

Dès lors, il faut modifier le fichier principal, « main.c » pour qu'il réagisse à l'UART.

3.13.11 Implémentation du code liant le hardware au fonctionnement logique

Tout d'abord, il faut inclure les headers :

```
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "logic_gates.h"
#include "logic_uart.h"
/* USER CODE END Includes */
```

Figure 132: Inclusion des headers dans le "main.c"

Lorsqu'une interruption intervient pour l'UART, la fonction suivante est appelée :

```
/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{

}
```

Figure 133: Méthode appelée lors d'une interruption UART

Il est cependant important de mentionner qu'une interruption sur l'UART ne se fait uniquement si la fonction suivante est appelée auparavant. Le premier paramètre correspond à l'interface UART utilisée (dans ce cas-ci il s'agit de l'UART5), le second est un pointeur vers la destination où seront enregistrées les caractères reçus par l'UART et le dernier correspond au nombre de caractères attendus lors d'une trame UART :

```
HAL_UART_Receive_IT(&huart5, &rx_uart_value, 1);
```

Figure 134: Fonction à appeler pour qu'une interruption ait lieu

Cette fonction doit être appelée à chaque fois qu'une interruption a eu lieu, sinon aucune réception n'a lieu. Ainsi, voici le code entré dans la fonction d'interruption :

```
/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    HAL_UART_Receive_IT(&huart5, (uint8_t *)&rx_uart_value, 1);

    check_reception(&val_to_send, &rx_uart_value);

    HAL_UART_Transmit_IT(&huart5, (uint8_t *) &val_to_send, 1);
}
```

Figure 135: Code de l'interruption

À noter qu'il faut également ajouter la ligne de code suivante dans le main, afin de recevoir la première interruption UART :

```
/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_UART5_Init();

/* USER CODE BEGIN 2 */
HAL_UART_Receive_IT(&huart5, &rx_uart_value, 1);
/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
  /* USER CODE END WHILE */

  /* USER CODE BEGIN 3 */
}
```

Figure 136: Insertion d'une ligne de code dans le main

Et voici donc les variables nécessaires au bon déroulement du programme :

```
/* Private variables ---------------------------------------------------------*/
/* USER CODE BEGIN PV */
extern UART_HandleTypeDef huart5;
char rx_uart_value;
char val_to_send;
/* USER CODE END PV */
```

Figure 137: Variable du main

Une fois ceci réalisé, le pipeline de test s'exécute en entier, sans erreur, car le code pour l'AD2 n'a pas été modifié, ni pour la vérification des résultats :

Status	Pipeline	Commit	Stages
passed	#312 by latest	feature/add_... 83e47a77 update code for ARM	✓ ✓ ✓ ✓ ✓

Figure 138: Résultat du CI

3.13.12 Mise à jour du code de l'AD2

Pour utiliser l'UART de l'AD2, deux modules sont conçus. Le premier concernera la gestion du Tx et le second celle du Rx.

Dans un premier temps, il faut configurer l'UART. Cela se passe dans le fichier principal, soit « digilent_analyzer_tester.py » :

```
# digital tests
digital_record.start()
digital_out.start()
# let the tests run for 5 seconds
time.sleep(5)
# end of previous tests ----------------------------------------------
# reset the AD2
dwf.FDwfDeviceReset(hdwf)
# let do the reset
time.sleep(0.1)

# configure the I2C/TWI, default settings
dwf.FDwfDigitalUartRateSet(hdwf, c_double(9600)) # 9.6kHz
dwf.FDwfDigitalUartTxSet(hdwf, c_int(0))  # TX = DIO-0
dwf.FDwfDigitalUartRxSet(hdwf, c_int(1))  # RX = DIO-1
dwf.FDwfDigitalUartBitsSet(hdwf, c_int(8))  # 8 bits
dwf.FDwfDigitalUartParitySet(hdwf, c_int(0))  # 0 none, 1 odd, 2 even
dwf.FDwfDigitalUartStopSet(hdwf, c_double(1))  # 1 bit stop length
```

Figure 139: Configuration de l'UART pour l'AD2

Il s'en suit la création des threads pour le Rx et le Tx, s'exécutant puis rompt la connexion avec l'AD2 à la fin des tests :

```
# create the threads
uart_rx = UartRx(hdwf, dwf)
uart_tx = UartTx(hdwf, dwf)

# let them be create
time.sleep(0.1)

# launch the threads
uart_rx.start()
uart_tx.start()

# wait until the end of the test
time.sleep(40)

# and then close the device
dwf.FDwfDeviceClose(hdwf)
```

Figure 140: Exécution des threads pour Rx et Tx de l'UART

Concernant le thread qui gère la transmission (« UartTx »), son code est le suivant. Pour envoyer une trame UART, il faut utiliser la méthode « FDwfDigitalUartTx » :

```
tsec = time.clock() + 5 # wait 5 seconds
while tsec > time.clock():
    time.sleep(0.2)

txt_to_send = "0"   # here send the 1st char, a '0' ---------------
rgTX = create_string_buffer(bytes(txt_to_send, encoding="ascii"))

self.dwf.FDwfDigitalUartTx(self.hdwf, rgTX, c_int(sizeof(rgTX)-1))
#-------------------------------------------------------------------
# wait to send another char
tsec = time.clock() + 10 # wait 10 seconds
while tsec > time.clock():
    time.sleep(0.2)

txt_to_send = "1"   # here send the 2nd char, a '1' ---------------
rgTX = create_string_buffer(bytes(txt_to_send, encoding="ascii"))

self.dwf.FDwfDigitalUartTx(self.hdwf, rgTX, c_int(sizeof(rgTX)-1))
#-------------------------------------------------------------------
# wait to send another char
tsec = time.clock() + 10 # wait 10 seconds
while tsec > time.clock():
    time.sleep(0.2)

txt_to_send = "3"   # here send the 3rd char, a '3' ---------------
rgTX = create_string_buffer(bytes(txt_to_send, encoding="ascii"))

self.dwf.FDwfDigitalUartTx(self.hdwf, rgTX, c_int(sizeof(rgTX)-1))
#-------------------------------------------------------------------
```

Figure 141:code pour le Tx de l'UART

Ensuite vient le thread pour la partie réception (« UartRx »). Dans un premier temps, voici la partie acquisition des données :

```
# create a string buffer of 100 char
rgRX = create_string_buffer(100)

tsec = time.clock() + 30 # receive for 30 seconds
while time.clock() < tsec:
    # method to acquire the received chars
    self.dwf.FDwfDigitalUartRx(self.hdwf, rgRX, c_int(sizeof(rgRX) - 1),
                               byref(self.cRX),
                               byref(self.fParity))  # read up to 8k chars at once
    # here it checks if its a valid reception
    if self.cRX.value > 0:
        rgRX[self.cRX.value] = 0  # add zero ending
        # decode it
        sz = rgRX.value.decode()
        # put it in a list
        self.saved_values.append(sz)
        # save the relative time to another list
        self.time_saved.append(time.clock() - tbegin)
```

Figure 142: Acquisition des valeurs de l'UART

La seconde partie pour le Rx concerne la création d'un fichier CSV « record_uart.csv ». En effet, cela facilitera la tâche de la vérification du bon fonctionnement et permettra d'avoir une trace des résultats mesurés en tant qu'artéfact :

```
f = open("record_uart.csv", "w")
f.write("time of acquisition[s], character[-]\n")
for index in range(len(self.saved_values)):
    f.write("%.3f, %s\n" % (self.time_saved[index], self.saved_values[index]))
    f.close()

    print(self.saved_values)
    print(self.time_saved)
```

Figure 143: Création d'un fichier CSV

À la suite de ceci, le fichier « .gitlab-ci.yml » doit être mis à jour, afin de profiter du nouvel artéfact. Ainsi, l'étape nommée « HW unit test » se voit modifié comme suit :

```
HW unit test:
  stage: hw_unit_test
  tags:
    - ST-Link
  script:
    - st-flash write build/verify_tester.bin 0x8000000
    - cd tests/Py
    - python3 digilent_analyzer_tester.py
  artifacts:
    paths:
      - tests/Py/record_digital.csv
      - tests/Py/record_uart.csv
```

Figure 144: Modification du fichier ".gitlab-ci.yml"

Ainsi, lors de l'exécution du CI, les tests restent tous réussis, et le nouvel artéfact est disponible :

Figure 145: Apparition du nouvel artéfact

3.13.13 Vérification du bon fonctionnement du uC

Le fichier « record_uart.csv » a donc été généré. Voici sa forme :

```
time of acquisition[s], character[-]
5.044, 0
15.201, 1
25.326, X
```

Figure 146: Forme du fichier "record_uart.csv"

Ainsi, la première colonne correspond au temps où le caractère a été reçu, et la seconde colonne correspond au caractère reçu.

Afin de simplifier la vérification du fonctionnement de la cible, un autre fichier CSV (« in_out_uart.csv ») est conçu. Celui-ci a la même forme que le fichier généré par l'AD2 :

```
time sent, character value
5, 0
15, 1
25, X
```

Figure 147: Aperçu du fichier "in_out_uart.csv"

Avec ceci, le code pour vérifier le bon fonctionnement de l'UART (« verification_uart.py ») est portable pour d'autres tests, en modifiant uniquement le fichier CSV « in_out_uart.csv ». Son fonctionnement est basé sur le même principe que pour celui présenté au point 3.13.6.

Il faut dès lors modifier dans le fichier « .gitlab-ci.yml » l'étape « analyze HW unit test results », pour ajouter la vérification de l'UART :

```
analyze HW unit test results:
  stage: analyze_hw_unit_test
  tags:
    - python3
  script:
    - cd tests/Py
    - python3 verification_digital.py
    - python3 verification_uart.py
```

Figure 148: Modification du fichier ".gitlab-ci.yml"

Ceci amène donc à la réalisation complète du CI ayant comme le résultat escompté :

```
$ python3 verification_uart.py
----------success----------
----------success----------
----------success----------
Authenticating with credentials from /home/marcberguera/.docker/config.json
Authenticating with credentials from /home/marcberguera/.docker/config.json
Job succeeded
```

Figure 149: Résultat des analyses concernant le fonctionnement de l'UART

3.13.14 Fusion de la fonctionnalité et du code principale

Pour finaliser l'implémentation de la nouvelle fonctionnalité, un merge request est réalisé vers la branche master, en suivant les instructions données au point 3.13.7.

4 Résultats

4.1 Connexion sécurisée entre la machine physique et le serveur Gitlab

Voici un aperçu de ce qu'indique le serveur Gitlab lors de l'établissement de la connexion, plus particulièrement lors d'essayer d'établir une connexion sécurisée :

```
* SSL connection using TLSv1.2 / ECDHE-RSA-AES256-GCM-SHA384
* ALPN, server accepted to use h2
* Server certificate:
*  subject: CN=*.hevs.ch
*  start date: Mar 21 00:00:00 2018 GMT
*  expire date: Jun 19 12:00:00 2020 GMT
*  issuer: C=US; O=DigiCert Inc; OU=www.digicert.com; CN=RapidSSL RSA CA 2018
*  SSL certificate verify ok.
```

Figure 150: Établissement d'une connexion sécurisée

Comme indiqué dans la figure ci-dessus, la connexion entre le serveur et la machine physique réalisant le CI se fait de manière sécurisée, grâce au SSL [11].

4.2 Exécution du lint

Au moment de la vérification du code, le taux de commentaire en pourcent est affiché :

```
taux de commentaire : 63.62
```

Figure 151: Aperçu du taux de commentaires

Le vérificateur de syntaxe est dédié aux commentaires. Ainsi, il va calculer un pourcentage du nombre de caractères en commentaire par rapport au nombre de caractères au total dans le programme. Si celui-ci est supérieur à 50 pourcents le code est considéré comme bien commenté et donc le CI passe à l'étape suivante.

4.3 Compilation du code

Lors de la compilation du code pour le uC et pour les tests unitaires, énormément de commandes sont affichées. Voici uniquement les dernières lignes :

```
arm-none-eabi-size build/verify_tester.elf
   text    data     bss     dec     hex filename
   6956      20    1748    8724    2214 build/verify_tester.elf
arm-none-eabi-objcopy -O ihex build/verify_tester.elf build/verify_tester.hex
arm-none-eabi-objcopy -O binary -S build/verify_tester.elf build/verify_tester.bin
$ make run_test
gcc Inc/logic_gates.h unity/unity.h unity/unity_internals.h unity/unity_fixture.h Src/logic_gates.c
Src/logic_uart.c tests/Src/logicGatesTest.c tests/Src/logicUartTest.c tests/Src/logicTestRunner.c
unity/unity.c unity/unity_fixture.c -IInc -IInc -Iunity -o check_software_behaviour
```

Figure 152: Compilation du code

4.4 Exécution des tests unitaires

Lors de la troisième étape du pipeline, les premières vérifications du bon fonctionnement du code sont réalisées, à l'aide du Framework de test Unity :

```
$ ./check_software_behaviour
Unity test run 1 of 1
.................

----------------------
17 Tests 0 Failures 0 Ignored
OK
```

Figure 153: Exécution des tests unitaires

4.5 Chargement du code sur l'ARM

Une fois le comportement logique validé, la réalisation physique des tests a lieu. Ainsi, le uC est programmé :

```
2019-08-07T14:37:48 INFO common.c: Finished erasing 1 pages of 16384 (0x4000) bytes
2019-08-07T14:37:48 INFO common.c: Starting Flash write for F2/F4/L4
2019-08-07T14:37:48 INFO flash_loader.c: Successfully loaded flash loader in sram
enabling 32-bit flash writes
size: 6976
2019-08-07T14:37:48 INFO common.c: Starting verification of write complete
2019-08-07T14:37:48 INFO common.c: Flash written and verified! jolly good!
```

Figure 154: Chargement du code sur l'ARM

4.6 Programmation de l'AD2

Une fois la commande lançant le script python pour programmer l'AD2 afin de réaliser les tests hardware exécutée, le résultat suivant apparait :

```
$ python3 digilent_analyzer_tester.py
Opening first device...
Configuring Digital Out / In...
DigitanIn base freq: 100000000.0
configure output
Digital Input Pins:  0000000000000011
Recording...
Digital Input Pins:  0000000000000011
  done
send a 0
0
send a 1
1
send a 3
X
['0', '1', 'X']
[5.0277829999999994, 15.165728, 25.192476]
end of the tests
```

Figure 155: Programmation de l'AD2

4.7 Vérification d'un exemple

Finalement, les mesures ayant été réalisées, il faut alors les analyser. Ceci permet alors de certifier le bon fonctionnement physique de l'appareil sous tests (DUT) :

```
$ python3 verification_digital.py
----------success----------
----------success----------
----------success----------
----------success----------
----------success----------
----------success----------
----------success----------
----------success----------
----------success----------
----------success----------
----------success----------
----------success----------
----------success----------
----------success----------
----------success----------
$ python3 verification_uart.py
----------success----------
----------success----------
----------success----------
```

Figure 156: Vérification de l'exemple

4.8 Aperçu du pipeline de tests

Voici la visualisation du pipeline de tests effectué par le CI, lorsque toutes les étapes sont passées avec succès :

Figure 157: Visualisation du pipeline de tests

4.9 En cas d'erreur

4.9.1 Lint

Si le taux de commentaire est inférieur à 50%, le pipeline signale une erreur mais continue les tests suivants, de sorte à ne pas attendre que les commentaires soient rajoutés pour vérifier le bon fonctionnement du code:

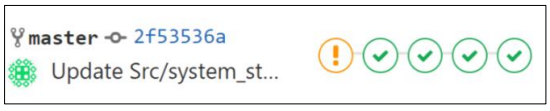


Figure 158: Avertissement pour le pipeline de tests

```
taux de commentaire : 40.85
Traceback (most recent call last):
  File "lint.py", line 98, in <module>
    raise ValueError("taux de commentaire trop faible");
ValueError: taux de commentaire trop faible
Authenticating with credentials from /home/marcberguera/.docker/config.json
ERROR: Job failed: exit code 1
```

Figure 159: Erreur lors du lint

4.9.2 Compilation

Les deux compilateurs (arm-gcc-embedded et gcc) compilent le code. S'il y a une erreur dans l'implémentation de celui-ci, le pipeline est arrêté et l'erreur est affichée :

Figure 160: Interruption du pipeline de tests

```
Src/system_stm32f4xx.c:13:4: error: expected identifier or '(' before '/' token
   */
    ^
make: *** [build/system_stm32f4xx.o] Error 1
Makefile:169: recipe for target 'build/system_stm32f4xx.o' failed
Authenticating with credentials from /home/marcberguera/.docker/config.json
ERROR: Job failed: exit code 1
```

Figure 161: Erreur lors de la compilation

4.9.3 Tests unitaires

Ayant utilisé le framework de tests d'Unity, les résultats sont affichés auquel cas une erreur a lieu, et le pipeline de tests s'arrête alors :

Figure 162: Interruption du pipeline de tests

```
$ ./check_software_behaviour
Unity test run 1 of 1
..............tests/Src/logicUartTest.c:22:TEST(LogicUart, Send_1):FAIL: Expected 48 Was 49
.tests/Src/logicUartTest.c:29:TEST(LogicUart, Send_0):FAIL: Expected 49 Was 48
.tests/Src/logicUartTest.c:36:TEST(LogicUart, Send_Other):FAIL: Expected 49 Was 88


----------------------
17 Tests 3 Failures 0 Ignored
FAIL
Authenticating with credentials from /home/marcberguera/.docker/config.json
ERROR: Job failed: exit code 1
```

Figure 163: Erreur lors des tests unitaires

4.9.4 Tests hardware

Auquel cas une erreur aurait lieu durant l'étape de réalisation des tests physiques (connexion impossible/interrompue ou soucis de programmations), le pipeline est interrompu, autant pour le uC :

Figure 164: Interruption du pipeline de tests

```
$ st-flash write build/verify_tester.bin 0x8000000
st-flash 1.5.1-30-g84f63d2
2019-08-12T11:37:51 WARN usb.c: Couldn't find any ST-Link/V2 devices
Authenticating with credentials from /home/marcberguera/.docker/config.json
ERROR: Job failed: exit code 1
```

Figure 165: Erreur de communication avec le uC

Que pour l'AD2 :

Figure 166: Interruption du pipeline de tests

```
$ python3 digilent_analyzer_tester.py
Opening first device...
Traceback (most recent call last):
  File "digilent_analyzer_tester.py", line 39, in <module>
    raise ValueError("failed to open device")
ValueError: failed to open device
Authenticating with credentials from /home/marcberguera/.docker/config.json
ERROR: Job failed: exit code 1
```

Figure 167: Erreur de communication avec l'AD2

4.9.5 Vérification des résultats

À ce moment les données enregistrées par l'AD2 sont analysées. S'il y a une erreur, le pipeline s'arrête immédiatement et indique uniquement qu'il y a eu une erreur :

Figure 168: Interruption du pipeline de tests

```
$ python3 verification_digital.py
----------success----------
----------success----------
----------success----------
----------success----------
----------success----------
----------success----------
----------success----------
----------success----------
----------success----------
----------success----------
----------success----------
----------error----------
Traceback (most recent call last):
  File "verification_digital.py", line 38, in <module>
    raise ValueError("measure wasn't expected")
ValueError: measure wasn't expected
Authenticating with credentials from /home/marcberguera/.docker/config.json
ERROR: Job failed: exit code 1
```

Figure 169: Erreur lors de la validation des mesures

5 Conclusion

5.1 Analyse des résultats

À la suite d'une analyse entre les différents gestionnaires de sources et les outils permettant de réaliser l'intégration continue, Gitlab s'est avéré être le choix le plus prolifique pour une utilisation gratuite. Ainsi, le serveur Gitlab interne de la HES a été utilisé afin de profiter de l'outil proposé nativement par Gitlab pour l'intégration continue : Gitlab CI.

Il s'en est suivi une décision concernant une cible à analyser. Celle-ci fut prise de manière arbitraire, un ARM Cortex-M4 de ST, car plusieurs travaux ont déjà été réalisés sur les ARM de ST et le fabricant propose beaucoup d'outils pour configurer le uC.

À la suite de cela, a eu lieu le choix de l'appareil allant effectuer les tests. C'est-à-dire émulant un environnement de travail pour la cible, le uC. L'appareil choisi fut l'Analog Discovery 2 de Digilent. Il n'y a pas énormément de fabricants qui proposent des outils de ce type et Digilent est l'un d'entre eux. Néanmoins, cet appareil offre de multitudes de possibilités avec la mesure de deux signaux analogiques, la génération de deux sorties analogiques et seize IO digitales, pouvant être utilisées comme pins pour des protocoles tels que l'UART, le SPI, l'i2c, et bien d'autres encore.

Pour pouvoir utiliser le uC et l'AD2, il a fallu concevoir des environnements sur une machine physique. Ceux-ci ont été réalisés par des images Docker, de manière à pouvoir toucher le plus de monde possible. En effet, ces images sont reprises de manière très simple, et permettent ainsi de faire profiter au plus grand nombre ces environnements préconfigurés pour l'AD2 et l'ARM.

Une connexion entre une machine physique utilisant des images docker a ensuite été réalisée, afin de pouvoir exécuter le CI directement depuis le serveur Gitlab, sans devoir entrer aucune commande autre. Il fut assez aisé de le réaliser grâce à la documentation de Gitlab-runner. Par la suite, cette connexion s'est vue sécurisée à l'aide du protocole SSL [11].

C'est alors que le pipeline de tests a vu ses étapes se construire. D'abord avec la compilation du code pour l'ARM, puis avec la programmation dudit code sur le uC, pour s'en suivre avec la programmation de l'AD2. Ensuite, des fonctions ont été rajoutées pour l'ARM, et donc la méthode du Test Drivent Développement a été utilisée, permettant d'effectuer des tests unitaires. Une fois ceci réalisé, la vérification du bon fonctionnement hardware a été rajoutée, pour terminer avec le vérificateur de syntaxe du code. Ainsi, cinq étapes ont lieu durant le pipeline de tests :

- Vérification syntaxique du code (vérifie le taux de commentaire)
- Compilation du code pour l'ARM et pour les tests unitaires
- Réalisation des tests unitaires
- Programmation du uC et de l'AD2 afin d'effectuer les tests hardware, enregistre les données du test
- Analyse des données enregistrées précédemment pour attester le bon fonctionnement physique du uC

Ce pipeline est opérationnel et les tests hardware effectués concernent une fonction binaire logique de quatres entrées à l'aide des portes AND, OR, NOT et XOR ainsi qu'une communication sérielle UART.

Ainsi, les objectifs proposés ont été atteints dans le temps imparti.

5.2 Améliorations futures

Les fonctionnalités importantes du CI sont opérationnelles. Afin d'étoffer la capacité de celui-ci, quelques points peuvent être améliorés. En effet, actuellement il n'y a que la programmation d'un uC de type ARM qui puisse être testé. Il serait souhaitable d'élargir le type de cibles pouvant être soumises à un pipeline de tests, comme un circuit logique programmable (FPGA) ou un uC Microchip PIC par exemple.

Un autre point augmentant l'efficacité d'utilisation d'un pipeline de tests pour une cible hardware concerne une modification de l'appareil effectuant les tests. En effet, l'Analog Discovery 2 s'est avéré très pratique pour démontrer qu'il est possible de réaliser ce type de tests hardware depuis Gitlab CI, mais très vite ses limites s'imposent. Il n'est, par exemple, pas possible de communiquer via l'UART et de modifier l'état des autres IO de l'appareil simultanément, malgré le fait qu'il s'agisse d'une FPGA qui gère le tout. Il serait intéressant de reprendre l'appareil mais de le programmer directement soi-même, de sorte à pouvoir réaliser tout ce que l'on souhaite, à l'aide d'une interface utilisateur (UI) très simple par exemple, comme le logiciel fourni par Digilent (Waveforms) mais, au lieu de programmer l'appareil, le logiciel générerait le code pour qu'il soit utilisé durant le CI.

L'AD2 amène une autre contrainte : la fréquence à laquelle l'appareil met à jour ses sorties. En effet, celui-ci adapte ses IO lorsqu'une communication USB est réalisée. Le problème apparait alors : la machine qui va programmer l'AD2 tourne sous Linux. Ceci implique qu'il y a une durée temporelle durant laquelle les tâches de l'ordinateur sont effectuées, le tick. Celui-ci est environ égal à onze millisecondes. Ainsi, les IO de l'AD2 sont mises à jour à chaque tick. Cette cadence de rafraichissement des IO n'est clairement pas suffisante pour tester un système embarqué, comme par exemple une interruption ayant lieu à des périodes d'une milliseconde. Une solution serait éventuellement de développer son propre outils (comme mentionné au paragraphe précédent) qui aurait déjà le code effectuant tous les tests implantés en son sein. Ainsi, la gestion des IO ne dépendrait plus d'une communication USB.
Une autre solution pourrait être d'avoir un appareil supplémentaire entre la machine physique et l'AD2, qui enregistrerait toutes les communications USB qui se feraient entre le PC et l'AD2. Comme l'appareil intermédiaire n'aurait pas d'autre tâches que de transmettre au bon moment une trame USB à l'AD2, les temps de mise à jour des IO de l'AD2 se verraient raccourcis.

Finalement, il serait préférable d'avoir un programme qui permettrait de générer le code pour programmer l'AD2, tout comme pour générer le code pour vérifier le bon fonctionnement du uC. Celui-ci pourrait avoir le même aspect que le programme Waveforms, qui est simple d'utilisation.

5.3 Conclusion

Il est donc possible d'effectuer un pipeline de tests capables de vérifier le bon fonctionnement physique d'une cible. Un autre avantage quant à cette manière de procéder est qu'elle ne demande pas d'avoir une plaque de développement complète pour tester si le code pour la cible fonctionne dans l'environnement spécifique lié à chaque projet développé, car l'appareil effectuant les tests émule ledit environnement. Ainsi, il n'y a plus d'attente sur la réalisation d'une plaque de développement pour chaque projet, et donc une rentabilité augmentée pour chacun de ceux-ci.

De plus, une seule cible peut être partagée entre plusieurs utilisateurs. Il n'y a plus besoin pour chaque développeur d'avoir son propre environnement de développement, car le CI va compiler et programmer la cible par lui-même.

En utilisant la méthode du TDD, les erreurs logiques sont traitées avant qu'elles ne soient constatées physiquement. Ainsi, un gain de temps est réalisé. De plus, à chaque exécution du CI, il est possible de vérifier que le nouveau code développé n'amène pas d'erreur sur le comportement précédent dudit code.

Un autre avantage de cette manière de procéder est qu'à la base de tout ceci, il s'agit d'un gestionnaire de sources et donc, le développement en équipe n'est que renforcé par cette méthode.

Chaque entreprise ou particulier peut avoir son propre serveur Gitlab gratuit. Ceci est un énorme avantage s'il n'est pas souhaité de sauvegarder ses projets sur une grande plateforme en ligne.

Un autre avantage est que le type de cible n'est pas limité. En effet, pour l'instant il n'est possible uniquement de tester un ARM mais il est facilement envisageable de charger un programme sur une FPGA du moment qu'une image docker, par exemple, le permette.

De plus, à l'aide de l'AD2 il est possible de vérifier plusieurs fonctionnalités, telles que des entrées/sorties analogiques ou digitales, tout comme des protocoles de communication. Malheureusement il n'est actuellement pas possible de faire simultanément des vérifications pour les protocoles et pour les IOs.

Néanmoins, il est encore complexe de programmer l'appareil effectuant les tests hardware. Un logiciel aidant les utilisateurs à générer le code pour ledit appareil, ainsi que le code vérifiant si les résultats sont corrects, amènerait un énorme surplus pour effectuer de l'intégration continue sur une cible.

Ainsi, cette manière de procéder est très puissante, car les tests sont effectués de manière autonome, sans besoin d'intervention pour démontrer que le programme fonctionne ou non.

Date et Signature

Berguerand Marc

Sion, le 16 août 2019

6 Références

[1] Git, « Why do you use Git », https://git-scm.com/about

[2] Docker, « Why do you use CI/CD », https://www.docker.com/solutions/cicd

[3] STM32CubeMX, STMicroelectronics, https://www.st.com/en/development-tools/stm32cubemx.html

[4] SW4STM32, STMicroelectronics, https://www.st.com/en/development-tools/sw4stm32.html

[5] TEXANE, Github, « ST-Link », https://github.com/texane/stlink

[6] Digilent, SDK, https://reference.digilentinc.com/reference/software/waveforms/waveforms-sdk/reference-manual

[7] Digilent, Waveforms, https://store.digilentinc.com/waveforms-previously-waveforms-2015/

[8] Digilent, Runtime, https://store.digilentinc.com/digilent-adept-2-download-only/

[9] Throwtheswitch, « Unity », http://www.throwtheswitch.org/unity

[10] Keil, Arm Limited, http://www.keil.com/

[11] Dominique Gabioud, documentation pour le cours Systèmes Distribués OpenSSL « OpenSSL_certificates », HES-SO Valais-Wallis, 2017

Inspirations pour la mise en forme du rapport :

J. Udressy, « Travail de diplôme : Transformateur MHz pour plasma atmosphérique», HES-SO Valais-Wallis, 2018

L. Pfefferlé, « Travail de diplôme : Bass loudspeaker control», HES-SO Valais-Wallis, 2018

J. Michel, « Travail de diplôme : Système embarqué pour drone matrice 210», HES-SO Valais-Wallis, 2018

7 Bibliographie

GRENNING, James. *Test-Driven Development for embedded C.* The Pragmatic Programmers, 2011.

KERNIGHAN, Briand et RITCHIE, Dennis. *The C Programming Language, Second Edition*. Prentice Hall, 1988.

SWINNEN, Gérard. *Apprendre à programmer avec Python 3*. Editions Eyrolles, 2012.

ARM, *Arm Compiler User Guide*. Arm Limited, 2016.

8 Annexes

Les documents suivants ont été annexés :

- Annexe n°1 : Utilisation de Github
- Annexe n°2 : Utilisation de Gitlab
- Annexe n°3 : Utilisation de Bitbucket
- Annexe n°4 : Utilisation de Gitlab CI/CD
- Annexe n°5 : Utilisation de Bitbucket-pipeline
- Annexe n°6 : Utilisation de Travic CI
- Annexe n°7 : Utilisation de Docker

Annexe n°1

Utilisation de Github

Table des matières

Liste des figures

Préambule

Toutes les images apparaissant dans cette annexe proviennent de la plateforme Github. Cette annexe a été réalisée au mois de mai 2019, veuillez prendre en compte que des modifications aient pu avoir lieu au moment où vous lisez ceci.

1 Différentes possibilités des comptes utilisateurs

Cette plateforme possède une communauté élevée de développeurs, plus de 36 millions en Avril 2019. Tout utilisateur peut y avoir accès gratuitement avec les propriétés de gestion suivantes :

- Nombre de répertoires publics illimités
- Nombre de répertoires privés illimités
- Suivi de problèmes et de bugs
- Management de projet

Il y a deux limitations au compte gratuit, qui sont les suivantes :

- Maximum de 3 collaborateurs pour les répertoires privés
- 1GB disponible au maximum pour stocker les fichiers

Il est toutefois possible de s'affranchir de cette limitation avec un compte payant. À noter également que Github propose des comptes « équipes », qui sont payants mais qui offrent des outils supplémentaires spécifiques aux entreprises et groupes de travaux, pour n'en citer que quelques-uns :

- Contrôle d'accès
- Facturation et management des utilisateurs
- Management de projet

1.1 Tarifs

Github propose diverses options en fonction du type de compte de l'utilisateurs. En voici la liste avec les principales caractéristiques et tarifs pour chacun d'entre eux :

	Free	Pro	Team	Enterprise
coût [$/mois]	0	7	9/utilisateurs	contacter pour savoir le prix
nombre de répertoires publiques	illimité	illimité	illimité	illimité
nombre de répertoires privés	illimité	illimité	illimité	illimité
nombres de collaborateurs par répertoires privés	3	illimité	illimité	illimité
suivi des bugs et erreurs	oui	oui	oui	oui
gestion de projet	oui	oui	oui	oui
outils avancés	-	page github Wikis branches protégées propriété du code graphes d'utilisations des répertoires	page github Wikis branches protégées propriété du code graphes d'utilisations des répertoires draft pull requests	page github Wikis branches protégées propriété du code graphes d'utilisations des répertoires draft pull requests support prioritaire
gestion des utilisateurs et des coûts	-	-	oui	oui
contrôle des accès des collaborateurs	-	-	oui	oui

Figure 1: Liste des tarifs proposés par Github

2 Identification

Pour s'identifier, il suffit de cliquer sur « Sign in » (indications en rouge sur la figure ci-dessous). Si aucun compte n'a été créer auparavant, il n'y a qu'à remplir le rectangle jaune sur la figure ci-dessous:

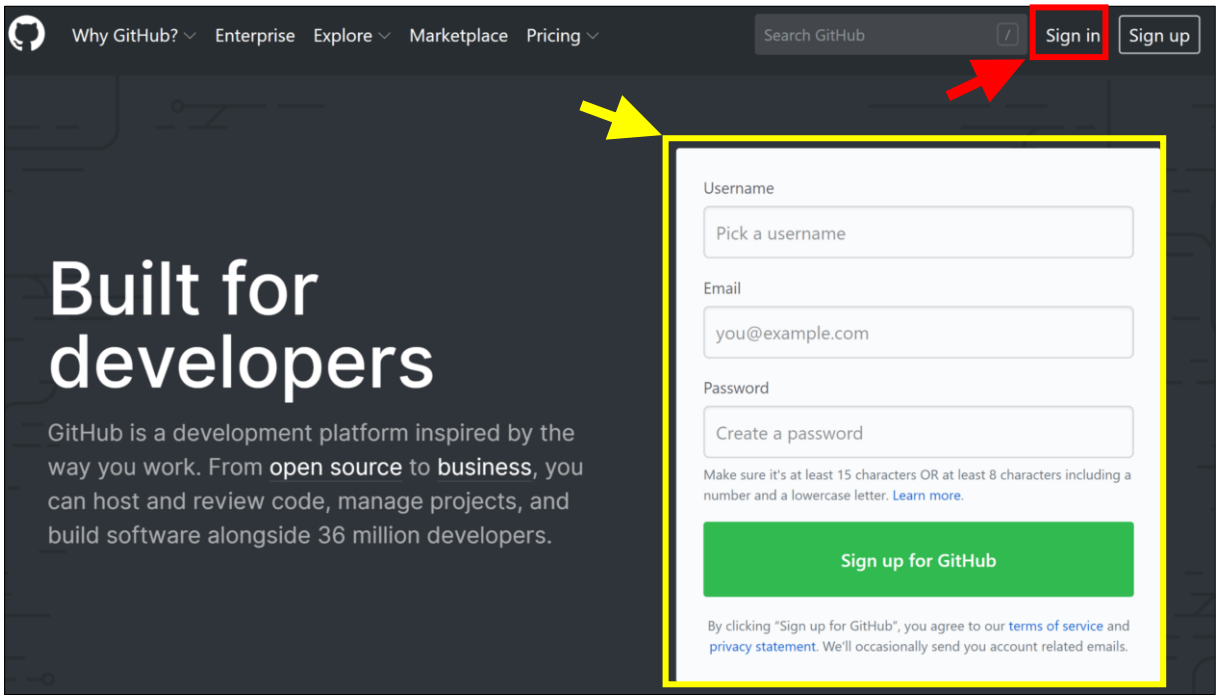


Figure 2: Fenêtre principale de Github

Une fois l'identification faite, la fenêtre Figure 3 s'affiche :

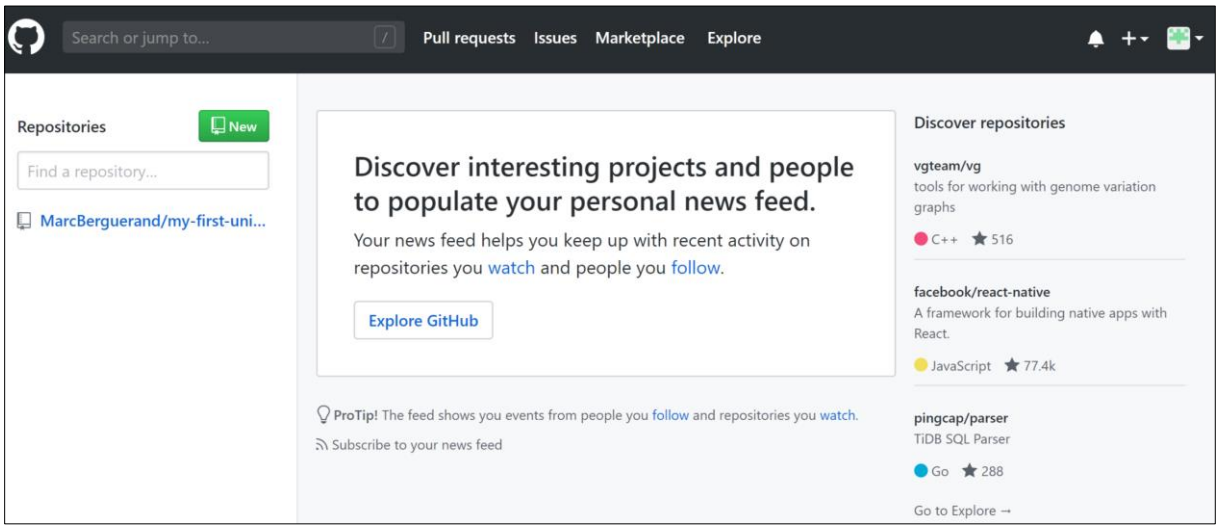


Figure 3: Fenêtre principale d'un utilisateur sur Github

3 Déconnexion

Pour se déconnecter, il faut aller sur l'onglet « utilisateur », puis cliquer sur « Sign out » :

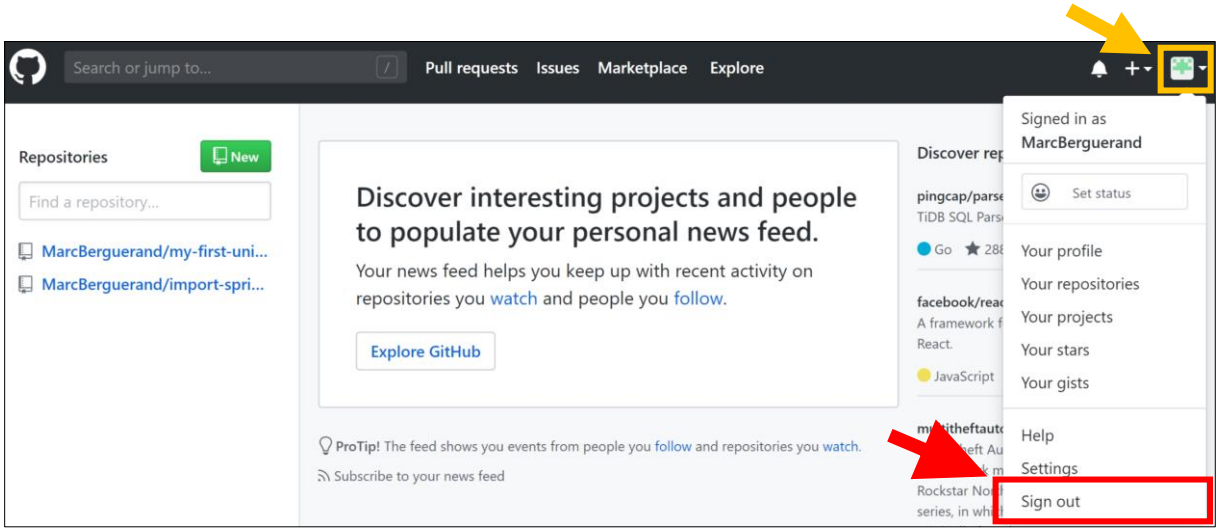


Figure 4: Déconnexion de Github

4 Héberger un répertoire

Voici les étapes basiques pour pouvoir héberger un projet sur Github :

4.1 Créer un nouveau répertoire

Tout d'abord, il faut aller sur l'onglet « + », juste à côté de l'onglet « utilisateur »:

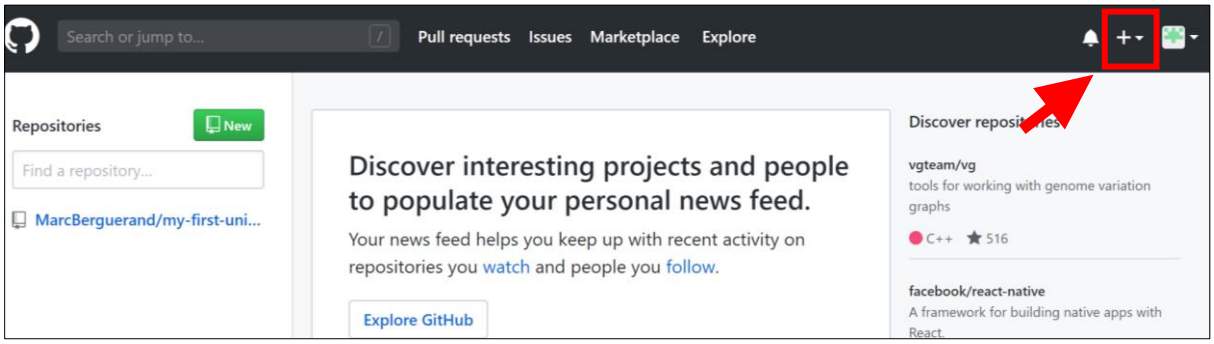


Figure 5: Situation de l'onglet « + »

Puis cliquer sur « New repository » :

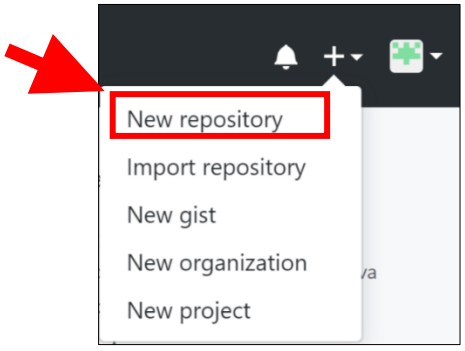


Figure 6: Onglet pour créer un nouveau répertoire Github

4.2 Lier le projet au répertoire Github

À ce moment-là, il est possible de choisir entre deux options : importer un répertoire déjà existant (cliquer là où se trouve le rectangle rouge) où en créer un (option par défaut, il suffit de remplir les champs affichés à l'écran).

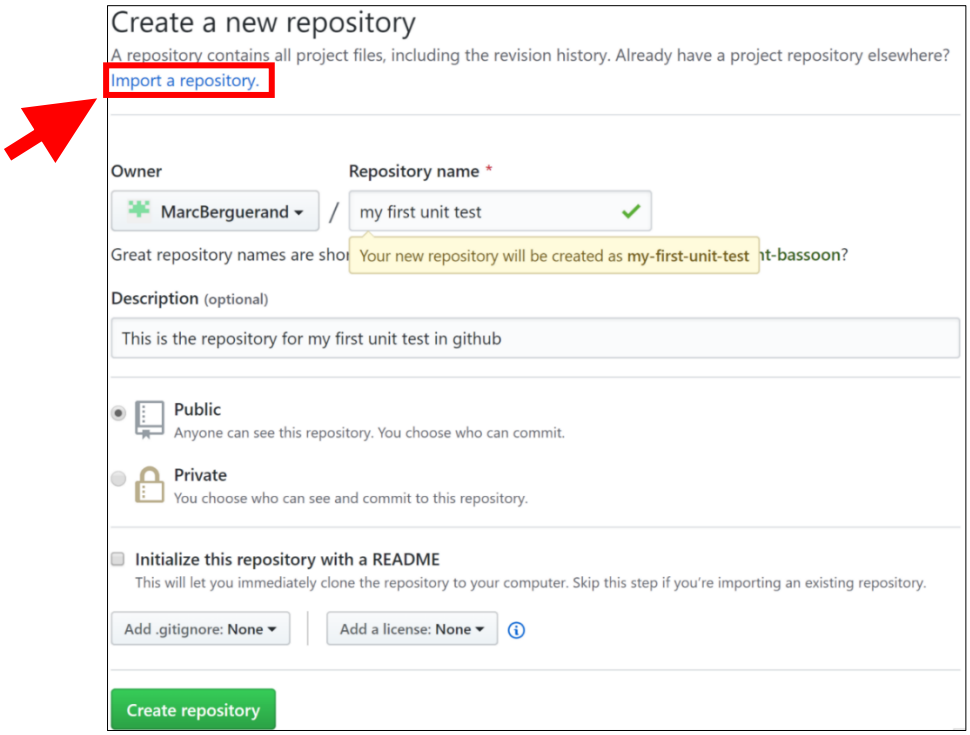


Figure 7: Choix possibles lors de la création du répertoire Github

4.2.1 Importer un répertoire déjà existant

Dans le cas où l'on souhaite importer un projet déjà existant, voici la fenêtre qui apparait. Ici est importé un répertoire depuis Gitlab :

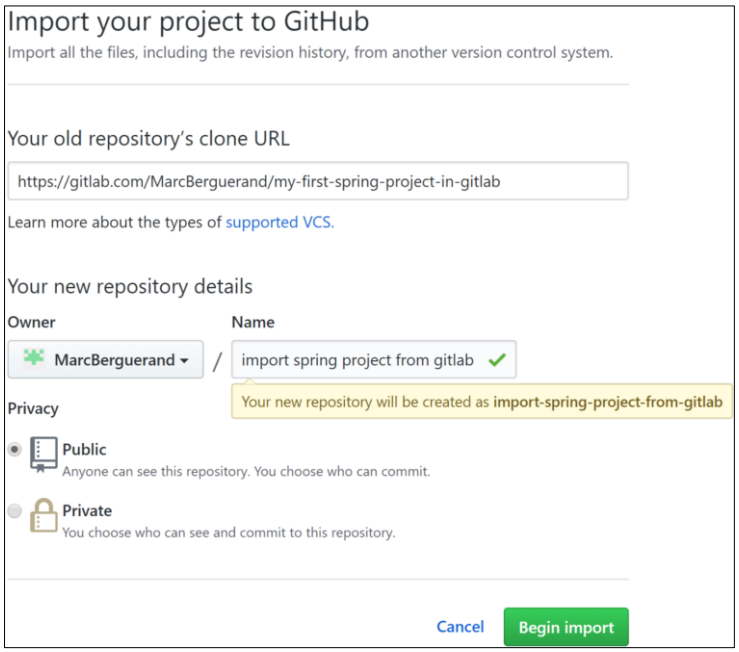


Figure 8: Fenêtre d'importation d'un répertoire dans Github

4.2.2 Créer un nouveau répertoire

Comme dit au point 4.2, il suffit de remplir les champs de la fenêtre puis de cliquer sur « Create repository ».

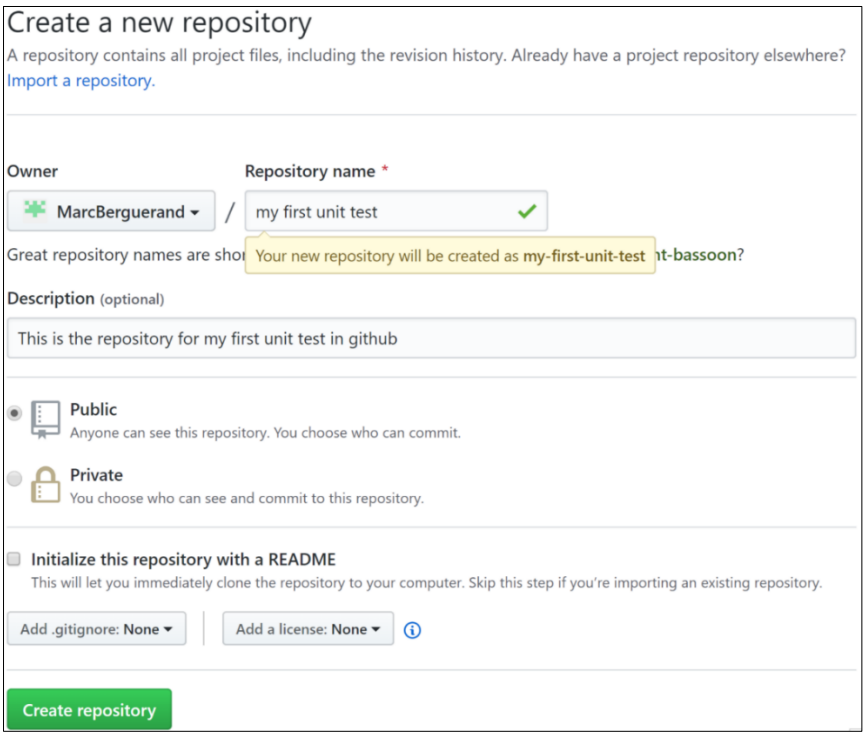


Figure 9: Choix possibles lors de la création du répertoire Github

Comme le montre la Figure 10, il est possible de faire le lien de diverses manières : créer directement les fichiers, télécharger des fichiers déjà existants ou importer via une invite de commande. L'utilisateur a libre choix d'utiliser la méthode qu'il préfère.

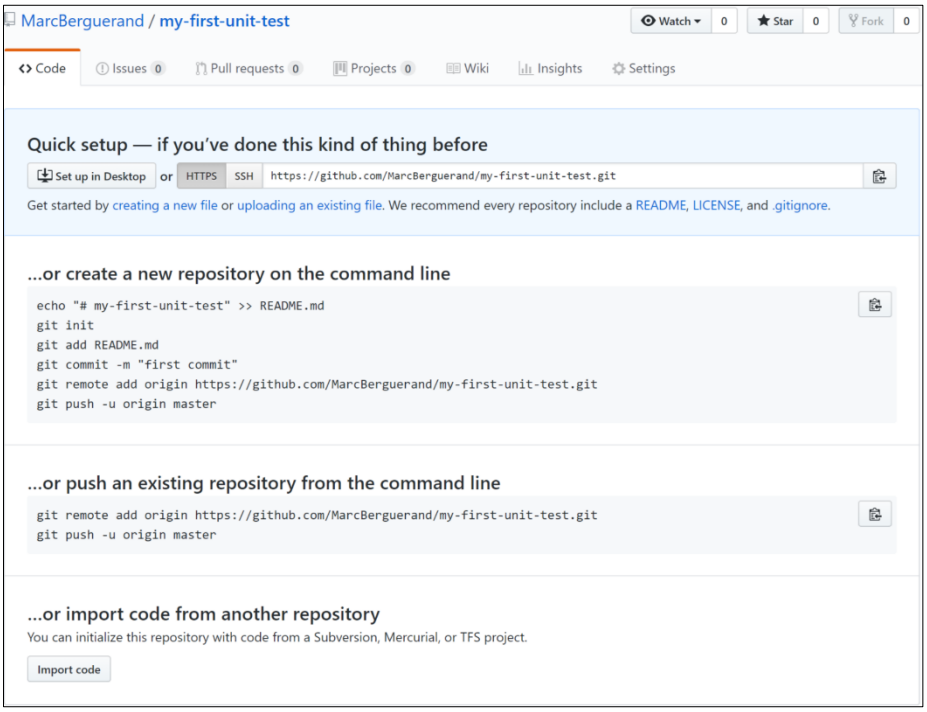


Figure 10: Aperçu de l'interface utilisateur une fois le répertoire Github créé

Pour cet exemple, les commandes git ont été utilisées. Voici le résultat final :

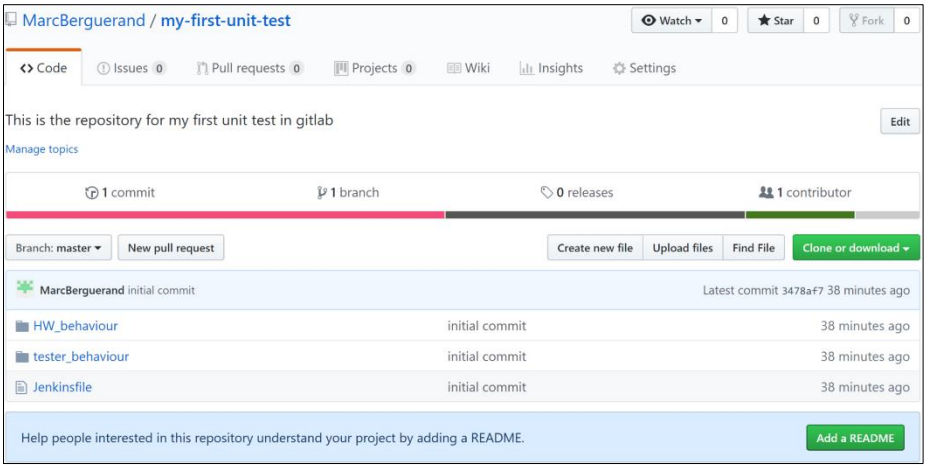


Figure 11: Aperçu de l'interface utilisateur avec le projet associé au répertoire Github

Annexe n°2

Utilisation de Gitlab

Table des matières

Liste des figures

Préambule

Toutes les images apparaissant dans cette annexe proviennent de la plateforme Gitlab. Cette annexe a été réalisée au mois de mai 2019, veuillez prendre en compte que des modifications aient pu avoir lieu au moment où vous lisez ceci.

1 Différentes possibilités des comptes utilisateurs

Plus de 100'000 organisations dans le monde utilisent cette plateforme. Tout développeur peut y avoir un compte gratuit avec les possibilités suivante :

- Nombre de répertoires publics illimités
- Nombre de répertoires privés illimités
- Nombre de collaborateurs pour les répertoires privés illimités
- Support communautaire
- Tableau de tâches intégré permettant de classer le type de travail et dans quel état il se trouve
- Outil d'intégration continue et de déploiement continu intégré

La principale limitation du compte gratuit est la suivante :

- Utilisation des machines fournies par Gitlab pour l'intégration continue limitées à 2000 minutes par mois

Il est cependant possible de définir sa propre machine qui effectuera l'intégration continue. Ainsi, cette limitation n'est plus valable. Autrement, un compte payant peur être fait, permettant d'acquérir plus d'outils et de support pour les utilisateurs, comme un vérificateur de qualité du code entre autres.

Gitlab offre également la possibilité d'avoir sa propre plateforme. Ainsi, les répertoires ne se trouvent plus sur la plateforme gitlab.com mais sur gitlab.hevs.ch par exemple, ce qui offre plus de contrôle sur le flux de données et les autorisations d'accès aux répertoires.

Cette plateforme auto-gérée est disponible gratuitement avec les mêmes conditions que le compte gratuit de gitlab.com, et il est également possible d'obtenir une version payante aux mêmes conditions que la plateforme de base.

1.1 Tarifs

Gitlab propose diverses options en fonction du type de compte de l'utilisateur. La particularité de Gitlab est qu'elle permet d'avoir son propre serveur. Il y a quatre types de comptes, autant pour le serveur principal que pour celui qui est interne au réseau, et les tarifs sont les mêmes pour l'un comme pour l'autre. En voici la liste :

	Free	Bronze	Silver	Gold
coût [$/mois]	0	4/utilisateurs	19/utilisateurs	99/utilisateurs
nombre de répertoires publiques	illimité	illimité	illimité	illimité
nombre de répertoires privés	illimité	illimité	illimité	illimité
nombres de collaborateurs par répertoires privés	illimité	illimité	illimité	illimité
type de support	communautaire	au jour ouvré suivant	prioritaire	prioritaire
temps de CI/CD [min/mois]	2000	2000	10000	50000
tableau des résultats de projet	oui	oui	oui	oui
approbation pour les merges	-	oui	oui	oui
inspecteur de qualité du code	-	oui	oui	oui
graphe de pipeline multi-projet	-	-	oui	oui
tableau de déploiement	-	-	oui	oui
gestion du portfolio	-	-	-	oui
scanneur de containeur	-	-	-	oui
gestion de license	-	-	-	oui

Figure 1: Liste des tarifs proposés par Gitlab avec leur plateforme d'hébergement

2 Identification

Pour s'identifier, il suffit de cliquer sur « Sign in » :

Figure 2: Fenêtre principale de Gitlab

Il s'en suit plusieurs cas possibles :

Si le compte a déjà été généré :

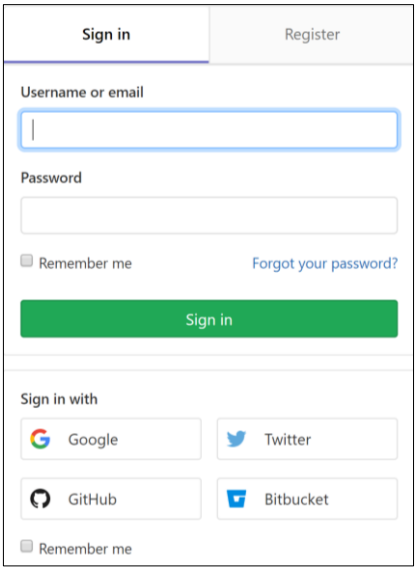


Figure 3: Identification sur Gitlab

S'il faut créer un compte :

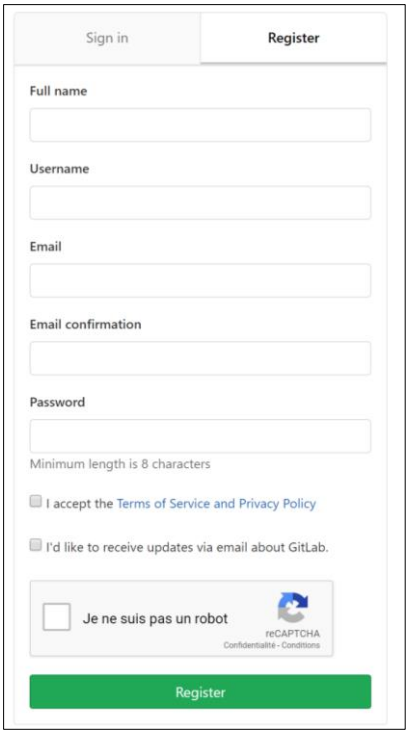


Figure 4: Création d'un compte Gitlab

Une fois l'identification faite, la fenêtre suivante apparait dans le navigateur :

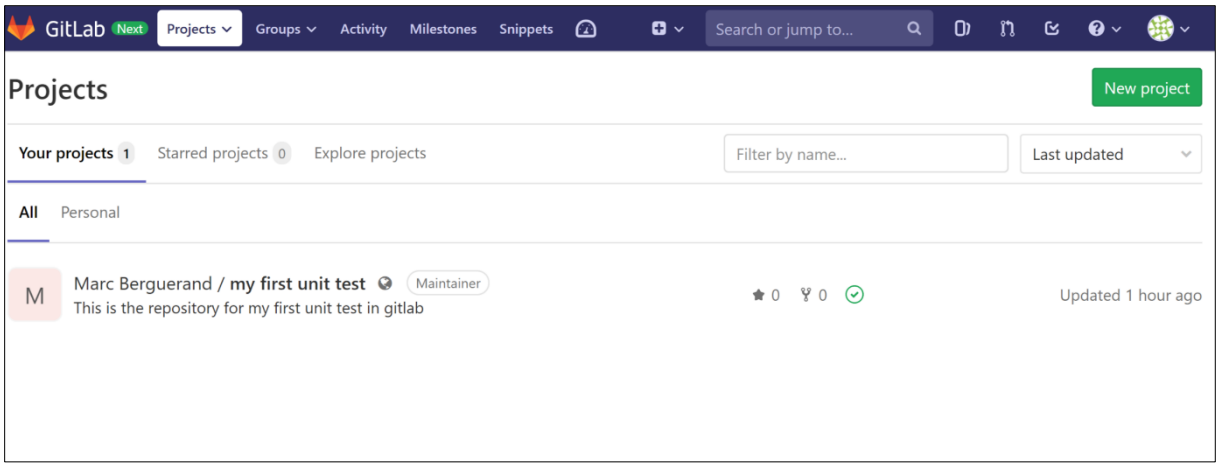


Figure 5: Fenêtre principale d'un utilisateur sur Gitlab

3 Déconnexion

Pour se déconnecter il suffit de cliquer sur l'icône de l'utilisateur et puis sur « Sign out » :

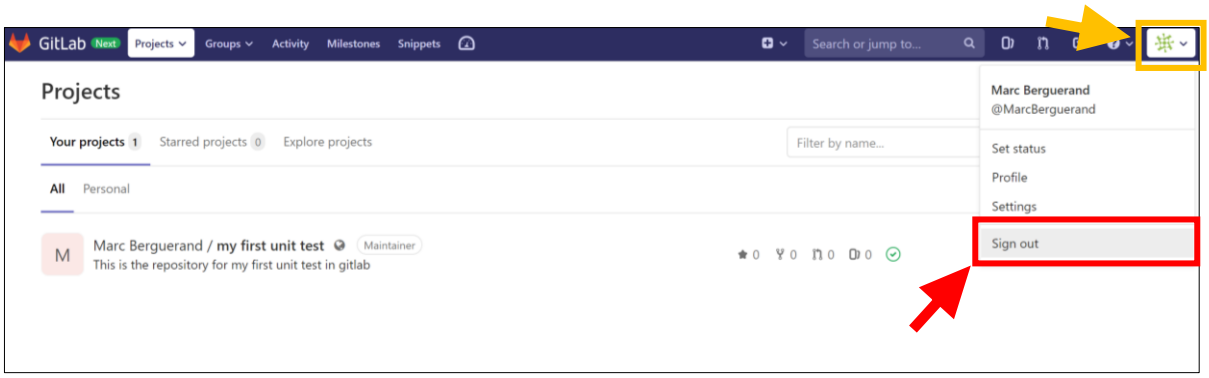


Figure 6: Déconnexion de Gitlab

4 Héberger un répertoire

Voici les étapes basiques pour pouvoir héberger un répertoire sur Gitlab :

4.1 Créer un nouveau répertoire

Tout d'abord, il faut aller sur l'onglet « + » :

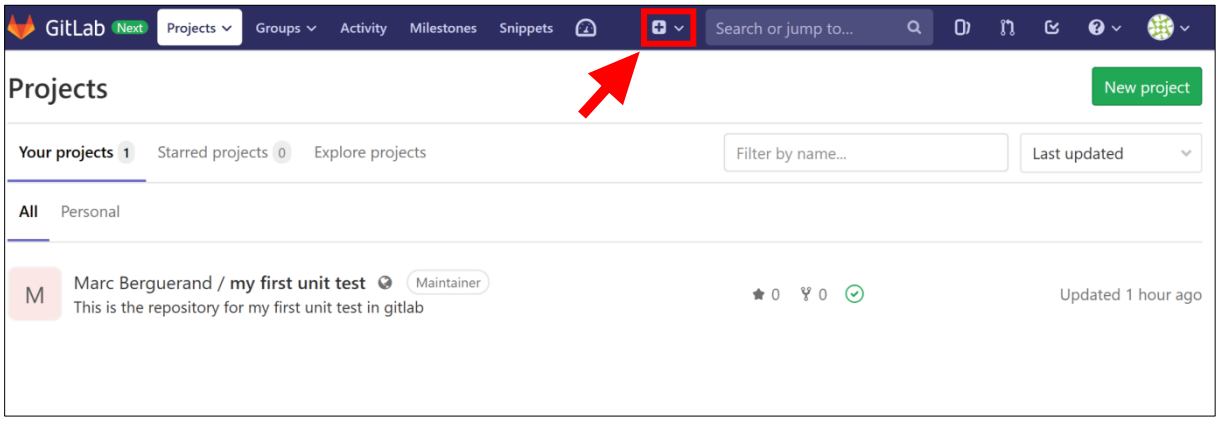


Figure 7: Situation de l'onglet «+»

Puis cliquer sur « New project » :

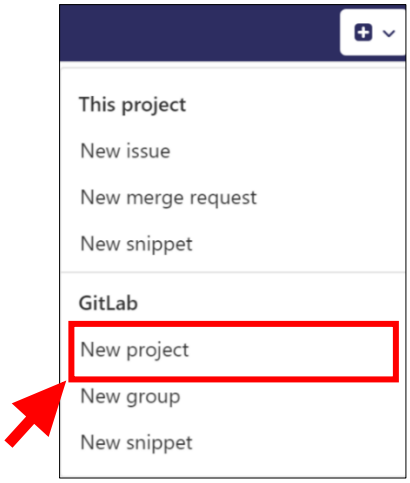


Figure 8: Onglet pour créer un nouveau répertoire Gitlab

4.2 Lier le projet au répertoire Gitlab

À ce moment-là, il est possible de choisir diverses options pour générer le répertoire Gitlab.

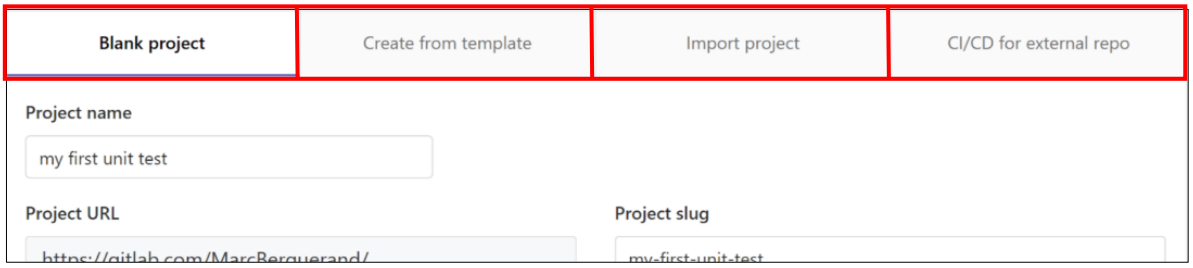


Figure 9: Configurations pour la création des répertoires

4.2.1 Création d'un répertoire vide

Cette option est à choisir si un projet est déjà existant. À ce moment, le nom du projet et une éventuelle définition sont à rentrer, et qu'il faut choisir s'il s'agit d'un projet privé, public ou interne. Gitlab offre la possibilité d'initialiser le répertoire avec un fichier « Readme » :

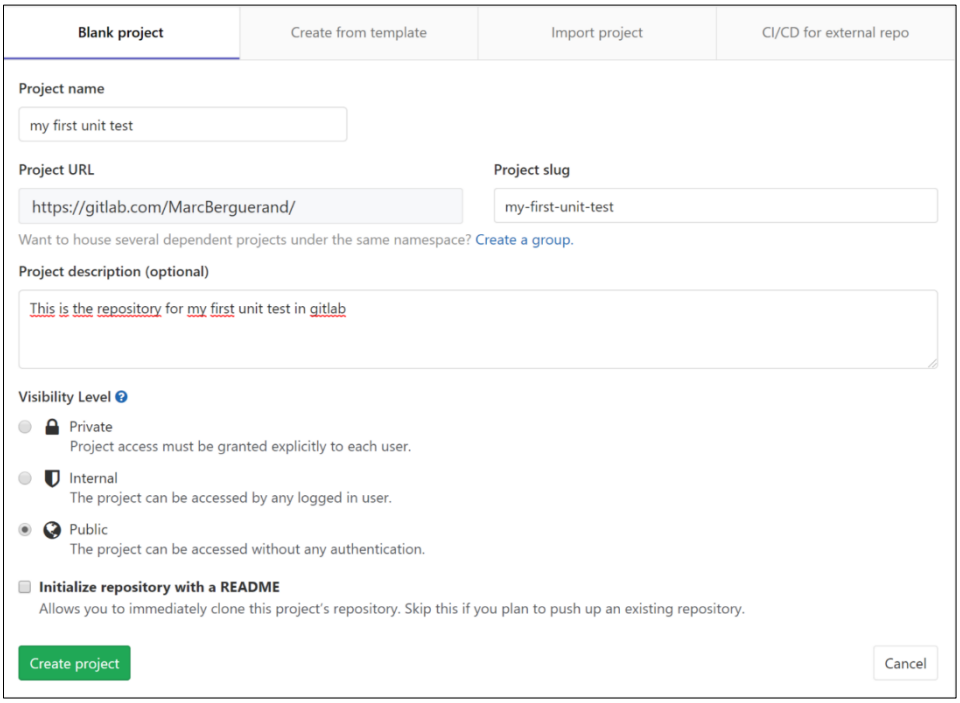


Figure 10: Options possibles lors de la création du répertoire vide sur Gitlab

À ce moment, cette fenêtre apparait lorsque le répertoire nouvellement créé est sélectionné :

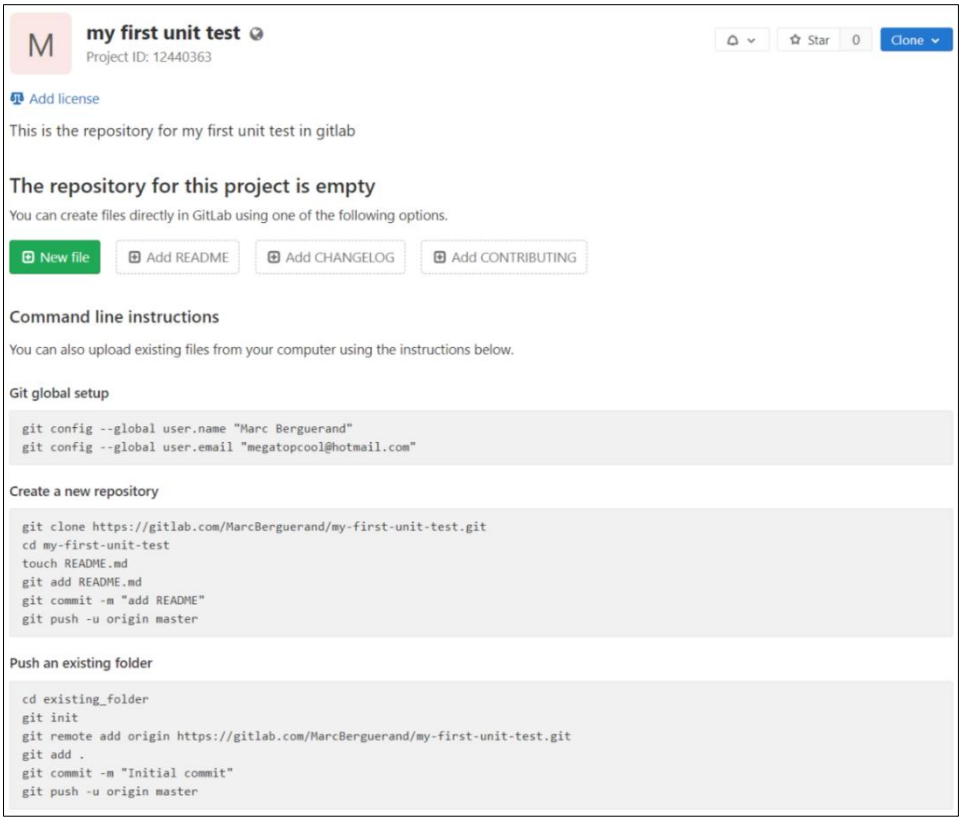


Figure 11: Aperçu de l'interface utilisateur une fois le répertoire Gitlab créé

Il faut alors lier le projet au répertoire à l'aide des commandes git proposées. Le résultat est le suivant :

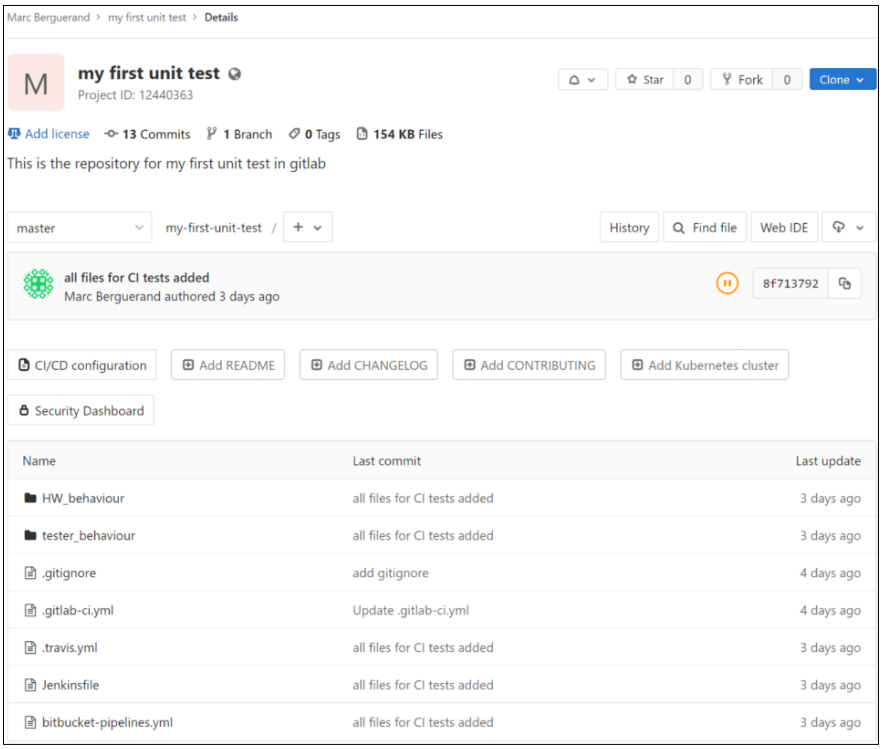


Figure 12: Aperçu de l'interface utilisateur avec le projet associé au répertoire Gitlab

4.2.2 Création depuis un modèle

Cette option permet de générer une architecture avec des configurations pour l'intégration continue, le déploiement continu et le gitignore déjà effectuées.

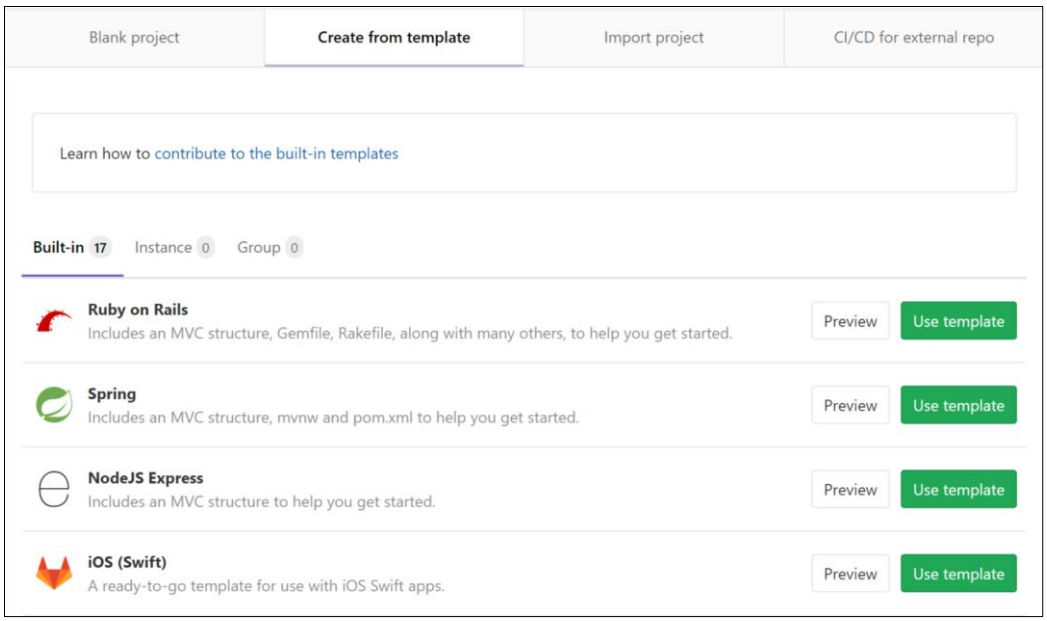


Figure 13: Création d'un répertoire Gitlab depuis un modèle

Ainsi, pour une modèle « Spring » par exemple, l'on retrouve les choix de la Figure 10 :

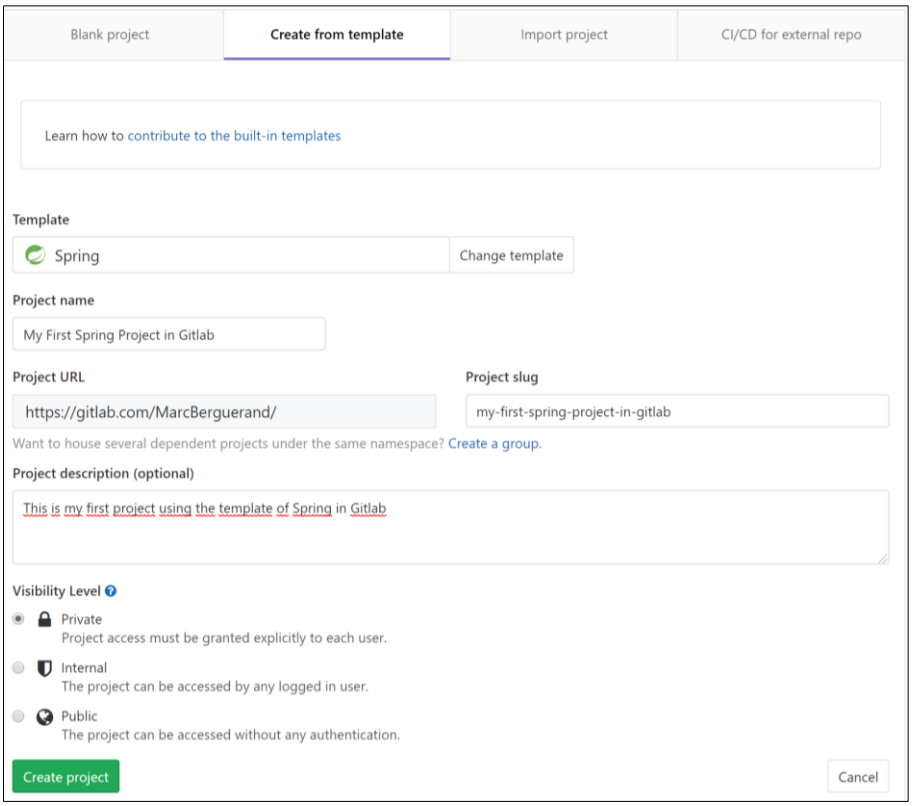


Figure 14: Options pour la création d'un projet depuis un modèle Gitlab

Et voici le résultat du projet créer à partir du modèle Spring. Tous les fichiers de bases sont insérés, ce qui fait des bases saines pour le départ d'un projet.

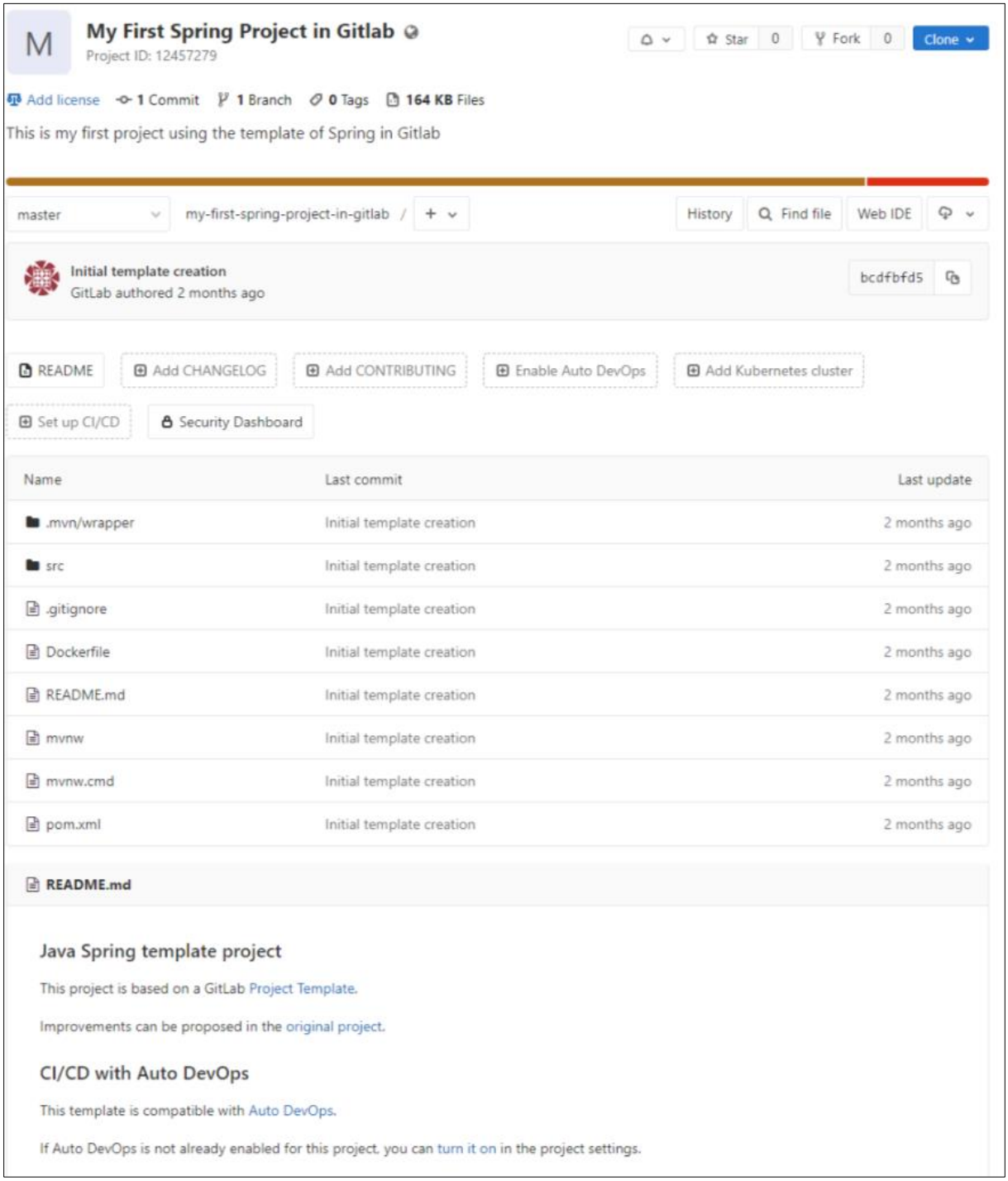


Figure 15: Résultat de la création à partir d'un modèle

4.2.3 Importation d'un répertoire déjà existant

Ceci permet de reprendre un répertoire depuis une autre plateforme d'hébergement :

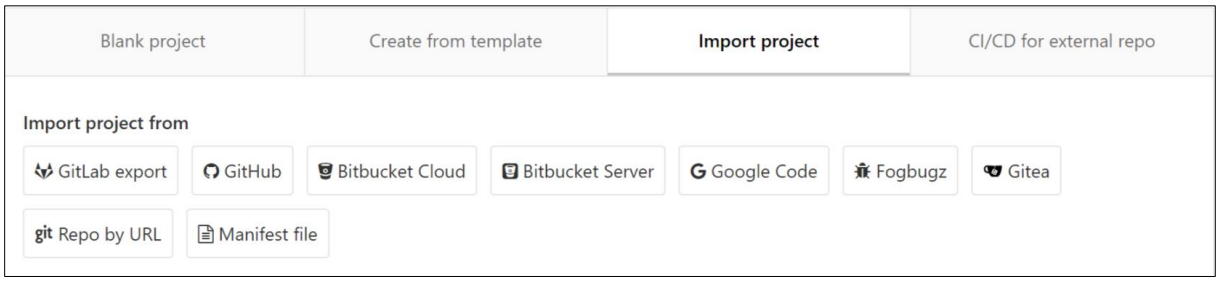


Figure 16: Création d'un répertoire Gitlab depuis un répertoire déjà existant

4.2.4 Associer les outils d'intégration continue et de déploiement continue

Cette option permet de lier les outils CI/CD de Gitlab à des répertoires qui ne sont pas hébergés sur Gitlab mais, par exemple, sur Github :

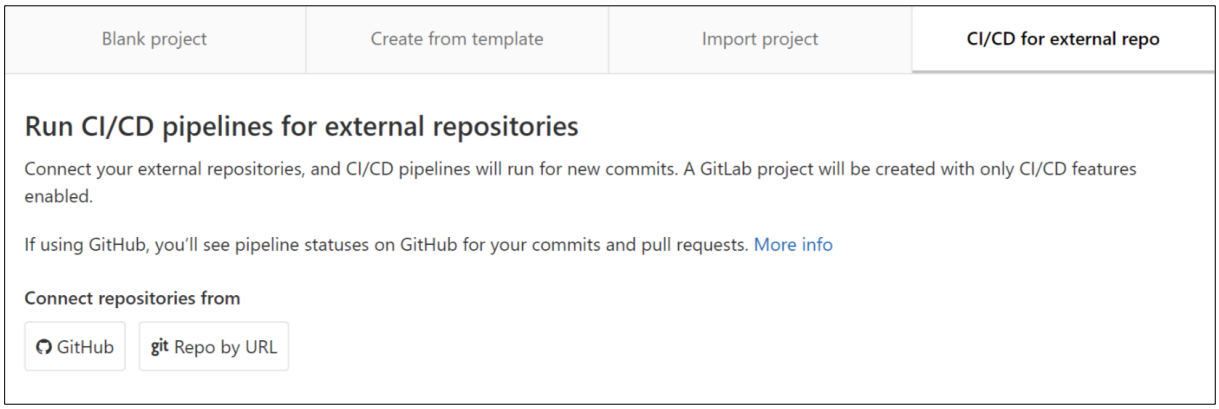


Figure 17: Association des outils d'intégration et déploiement continus de Gitlab à un autre répertoire

Ainsi, pour un projet hébergé sur Github, la fenêtre suivante apparait, demandant si un profil entier veut être lié ou si un accès à un répertoire seulement est utilisé.

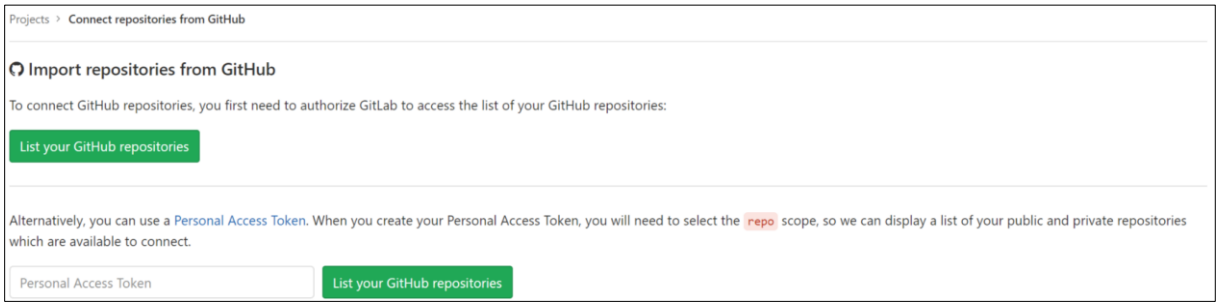


Figure 18: Choisir entre lier un compte ou un projet

À ce moment, une liaison entre un profil et le Gitlab CI/CD souhaitent être liés. Les accès sont donc demandés :

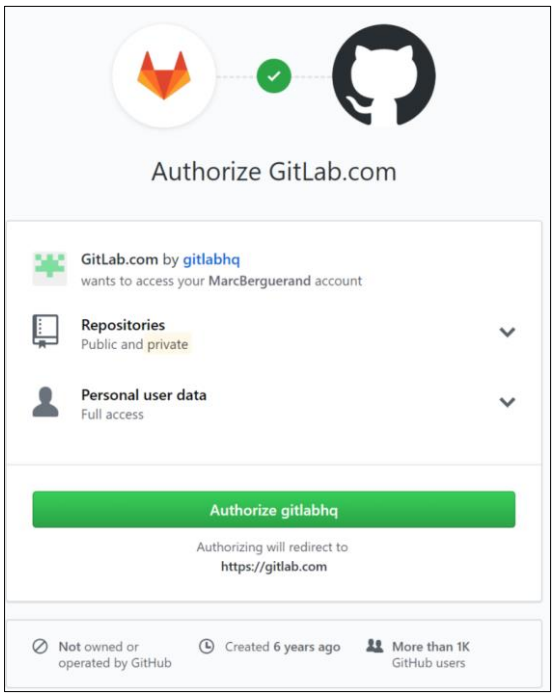


Figure 19: Autorisation de liaison du compte

Puis il est possible de lier tous les répertoires ou uniquement certains d'entre eux :

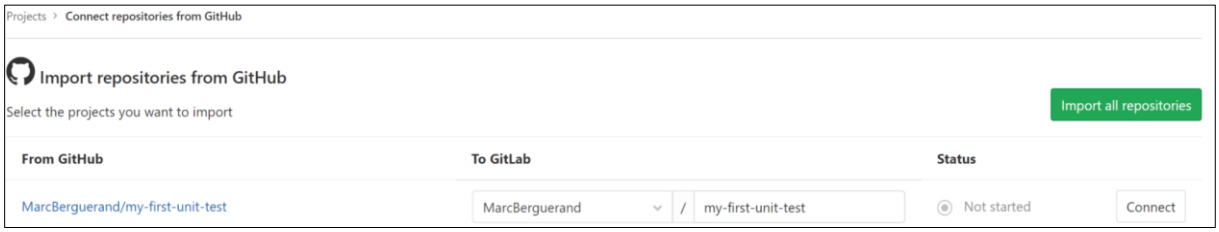


Figure 20: Choisir quel répertoire lier

Annexe n°3

Utilisation de Bitbucket

Table des matières

Liste des figures

Préambule

Toutes les images apparaissant dans cette annexe proviennent de la plateforme Bitbucket. Cette annexe a été réalisée au mois de mai 2019, veuillez prendre en compte que des modifications aient pu avoir lieu au moment où vous lisez ceci.

1 Différentes possibilités des comptes utilisateurs

Cette dernière plateforme est utilisée par plus de 10 millions de développeurs. Chacun peut avoir un compte gratuit offrant les propriétés suivantes :

- Nombre de répertoires publics illimités
- Nombre de répertoires privés illimités
- Logiciel Jira intégré de base avec les projets
- Outil d'intégration continue et de déploiement continu intégré

Les limitations liées au compte gratuit sont les suivantes :

- Utilisation des machines Bitbucket pour l'intégration continue limitées à 50 minutes par mois
- Sauvegarde de fichiers volumineux jusqu'à 1GB/mois
- Nombre de collaborateurs limités à 5 pour les répertoires privés

Il est cependant possible d'avoir un compte payant donnant l'accès à divers outils (vérificateur de merge, vérification en deux étapes par exemple) et d'augmenter les restrictions liées au compte gratuit.

Bitbucket offre également la possibilité d'avoir sa propre plateforme, contre un certain coût dépendant de l'utilisation (serveur ou centre de données) et du nombre d'utilisateurs.

1.1 Tarifs

Bitbucket propose diverses options en fonction du type de compte de l'utilisateurs. En voici la liste avec les principales caractéristiques et tarifs pour chacun d'entre eux, dans le cas d'utilisation de la plateforme principale :

	Free	Standard	Premium
coût [$/mois]	0	2/utilisateurs	5/utilisateurs
nombre de répertoires publiques	illimité	illimité	illimité
nombre de répertoires privés	illimité	illimité	illimité
nombres de collaborateurs par répertoires privés	5	illimité	illimité
temps disponible pour le CI [min/mois]	50	500	1000
capacité de stockage des fichiers [GB]	1	5	10

Figure 1: Liste des tarifs de Bitbucket pour l'utilisation de sa plateforme

S'il est souhaité utiliser un serveur interne, alors voici les tarifs varient en fonction du nombre d'utilisateurs et du type de serveur souhaité. Voici les caractéristiques des deux types de serveurs :

Serveur	Data Center
nombre de répertoire privé illimités	nombre de répertoire privé illimités
nombre de répertoire publique illimités	nombre de répertoire publique illimités
contrôle complet de l'environnement	contrôle complet de l'environnement
déploiement d'un seul serveur	active-active clustering
license perpétuelle et maintenance gratuite d'un an	license annuelle avec maintenance comprise
	Smart mirroring
	SAML 2.0

Figure 2: Caractéristiques des différents serveurs proposés

Et voici une liste réduite des tarifs pour les deux types de serveur (la liste complète se trouve sur le site de bitbucket : https://bitbucket.org/product/pricing?tab=self-hosted) :

nombre d'utilisateurs	25	100	500	1000
Paiement unique pour Serveur [$]	2500	8300	22000	30400
Paiement annueal pour Data Center [$/an]	1800	6000	16000	24000

Figure 3: Liste des tarifs raccourcies pour l'utilisation des différents serveurs

2 Identification

Pour s'identifier, il suffit de cliquer sur « Log in ». S'il faut créer un compte, alors cliquer sur « Get started » et suivre la procédure.

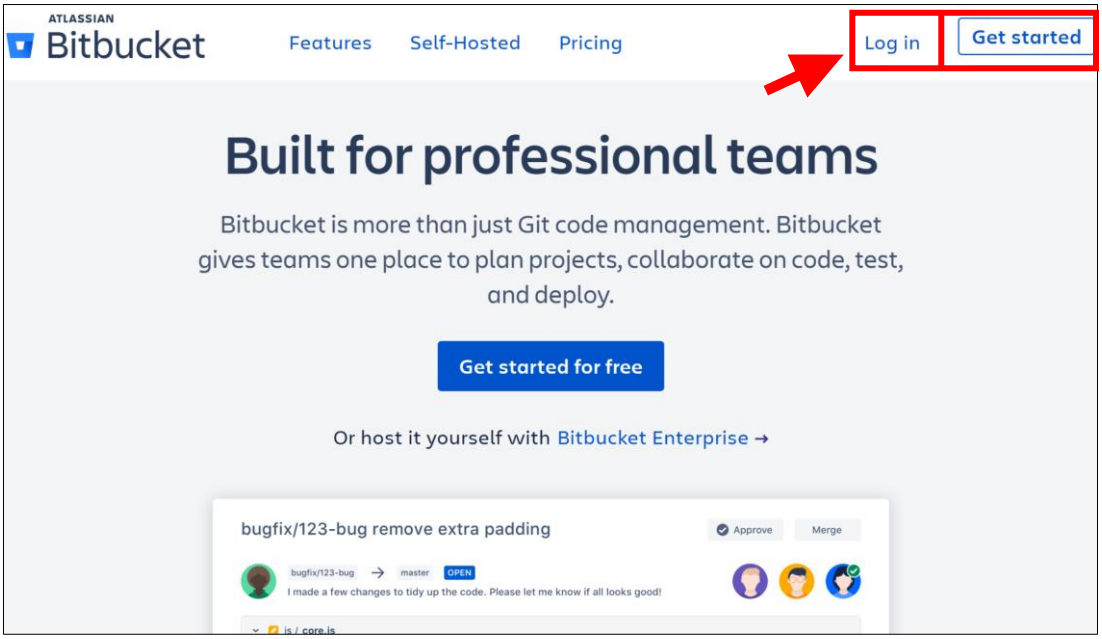


Figure 4: Fenêtre principale de Bitbucket

Une fois l'identification faite, la fenêtre ci-dessous est affichée :

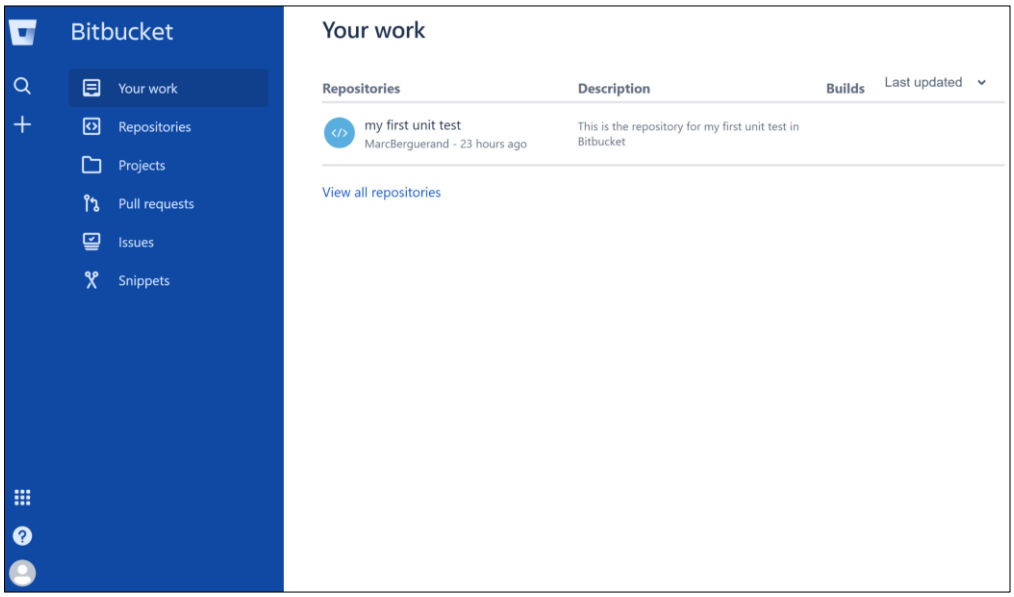


Figure 5: Fenêtre principale d'un utilisateur sur Bitbucket

3 Déconnexion

Pour se déconnecter, il faut cliquer sur l'onglet « utilisateur » (en vert sur la figure ci-dessous), puis sur « Log out » (en rouge sur la figure ci-dessous) :

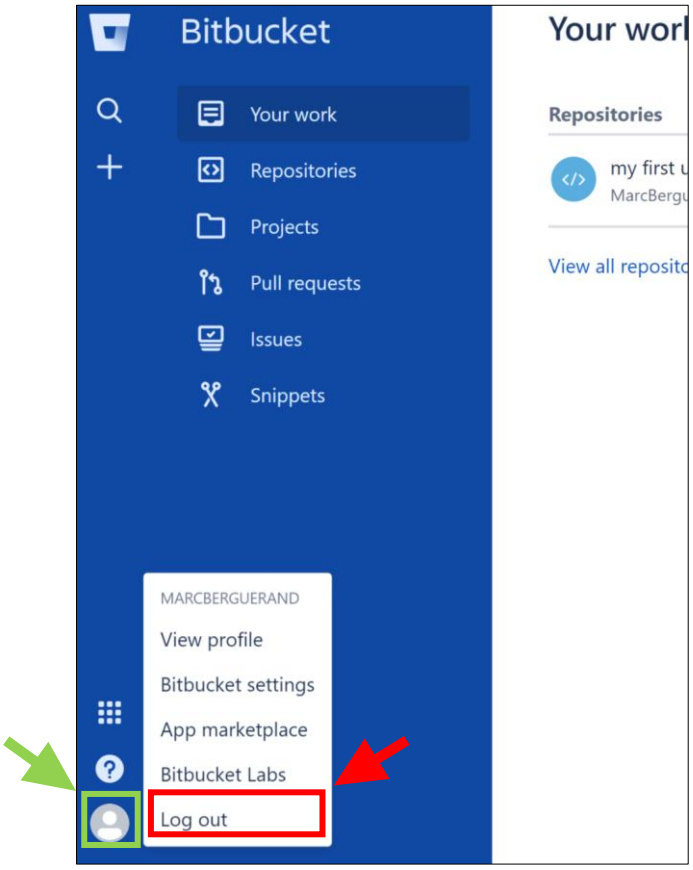


Figure 6: Situation de l'onglet « utilisateur »

4 Héberger un répertoire

Voici les étapes basiques pour pouvoir héberger un répertoire sur Bitbucket :

4.1 Créer un nouveau répertoire

Pour créer un nouveau répertoire, il faut d'abord cliquer sur l'onglet « + » :

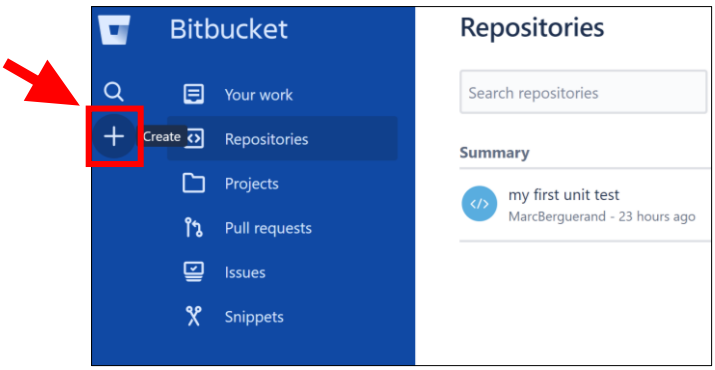


Figure 7: Situation de l'onglet « + »

Dès lors, le menu suivant apparait et il est possible de créer un nouveau répertoire (en vert) où d'en importer un (en rouge).

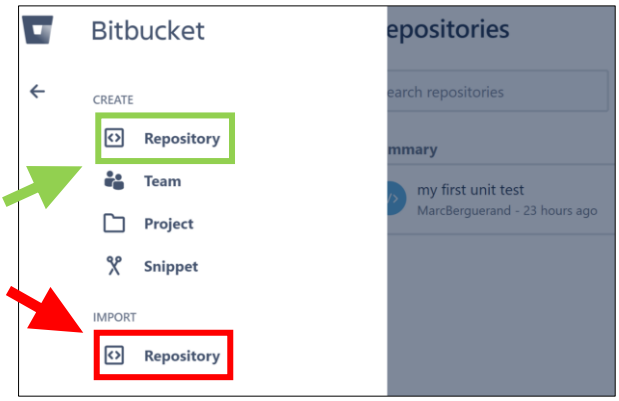


Figure 8: Menu pour créer un nouveau répertoire Bitbucket

4.2 Lier le projet au répertoire Bitbucket

4.2.1 Importer un répertoire déjà existant

L'option en rouge dans la Figure 8 a alors été choisie. Ainsi, la Figure 9 est affichée. Ici est importé un répertoire depuis Gitlab. Si ledit répertoire est privé, il faut alors cocher « Requires authorization » et remplire les champs de la Figure 10. Il est également possible de sélectionner le type de gestionnaire de version utilisé sous le champ « Source » (voir Figure 11).

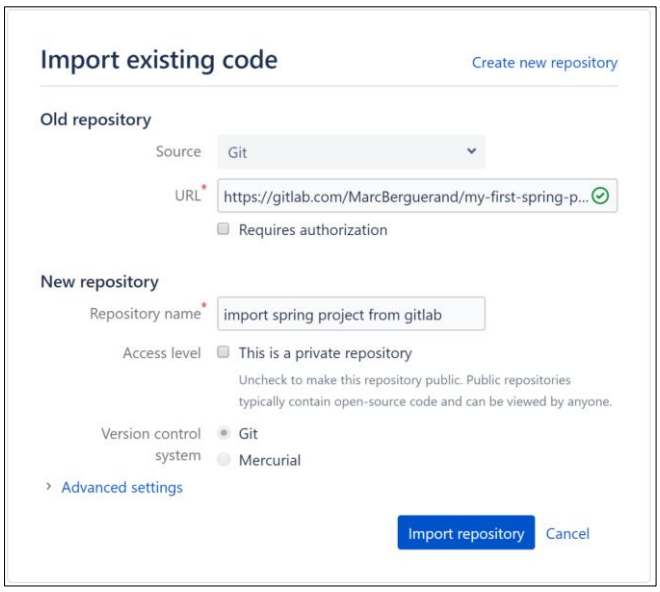


Figure 9: Fenêtre d'importation d'un répertoire dans Bitbucket

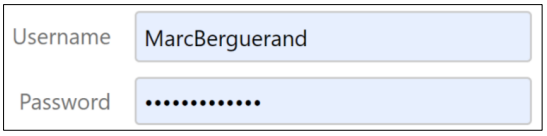


Figure 10: Dans le cas d'une autorisation requise

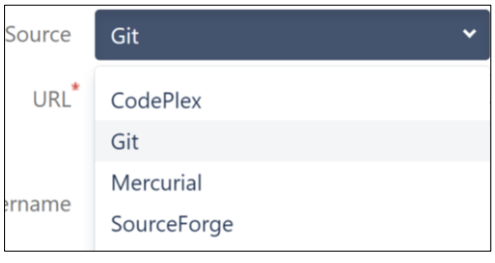


Figure 11: Divers gestionnaires de version importables

4.2.2 Créer un nouveau répertoire

Cette fois-ci, l'option en verte a été choisie dans la Figure 8 (Créer un nouveau répertoire). Voici ce qui est affiché :

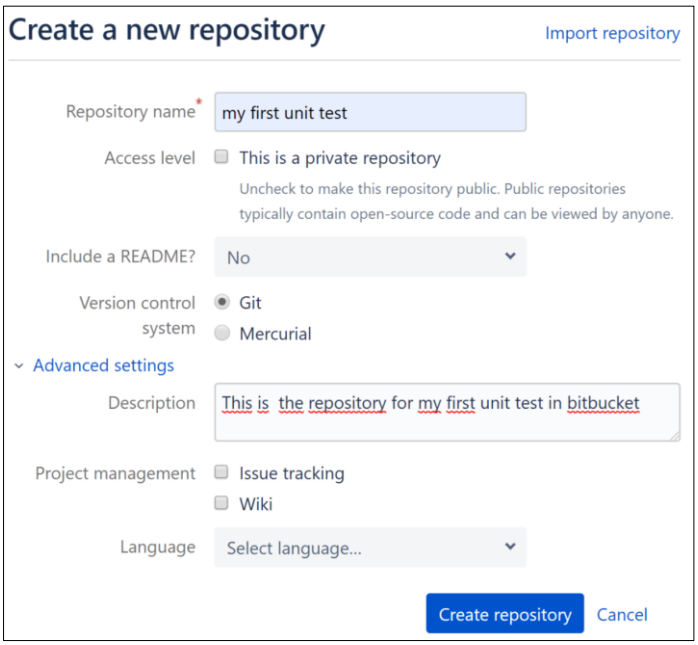


Figure 12: Choix possibles lors de la création du répertoire Bitbucket

Voici les diverses options pour les champs « Include a README ? » (Figure 13) et « Language »(Figure 14) :

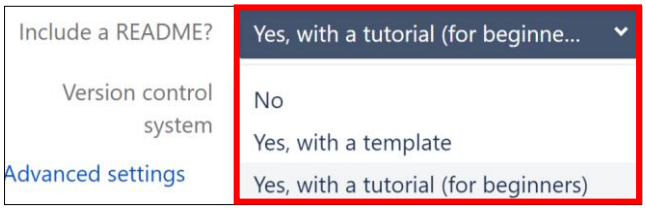


Figure 13: Options pour le champ « Include a README ? »

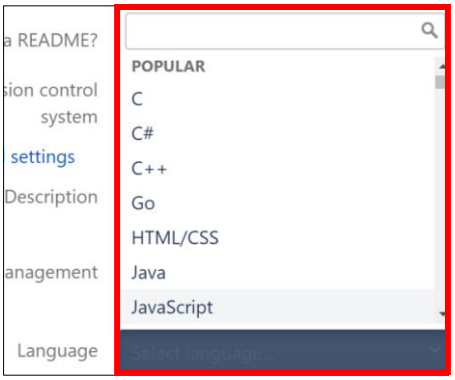


Figure 14: Options pour le champ « Language »

Ainsi, le répertoire est créé et la fenêtre ci-dessous est affichée. Pour y insérer des fichiers, il y a le choix entre les commandes git ou l'importation de fichiers depuis l'onglet « Downloads » (en rouge sur la Figure 15).

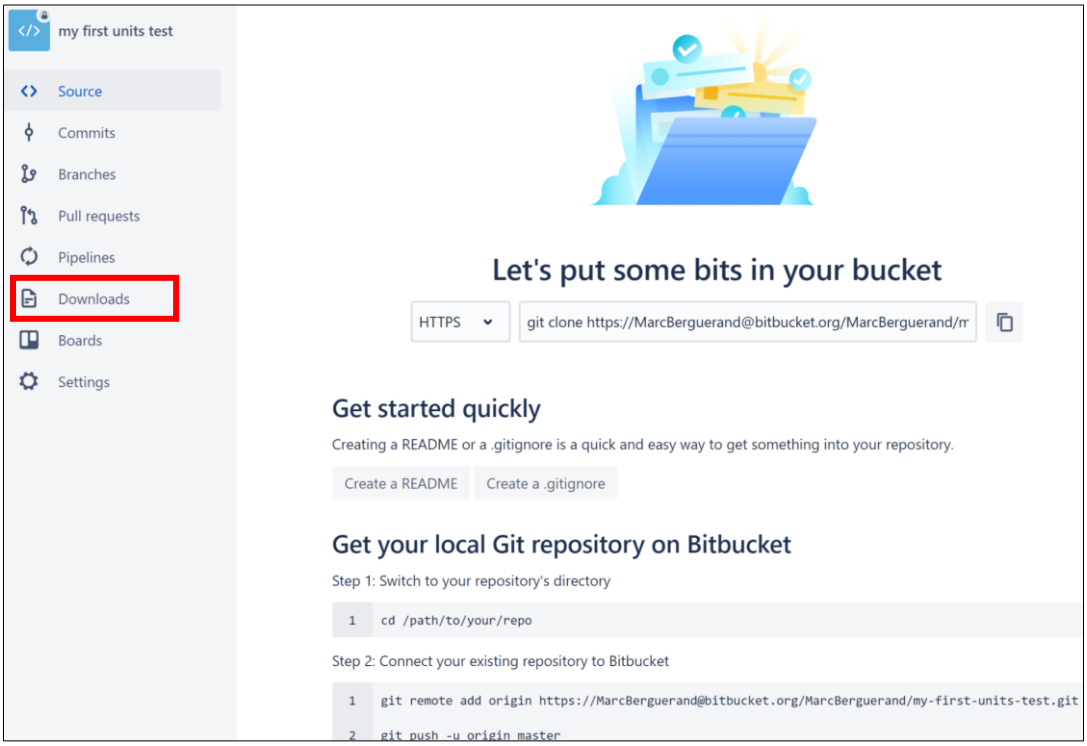


Figure 15: Aperçu de l'interface utilisateur une fois le répertoire Bitbucket créé

Pour cet exemple, les commandes git ont été utilisées. Voici le résultat final :

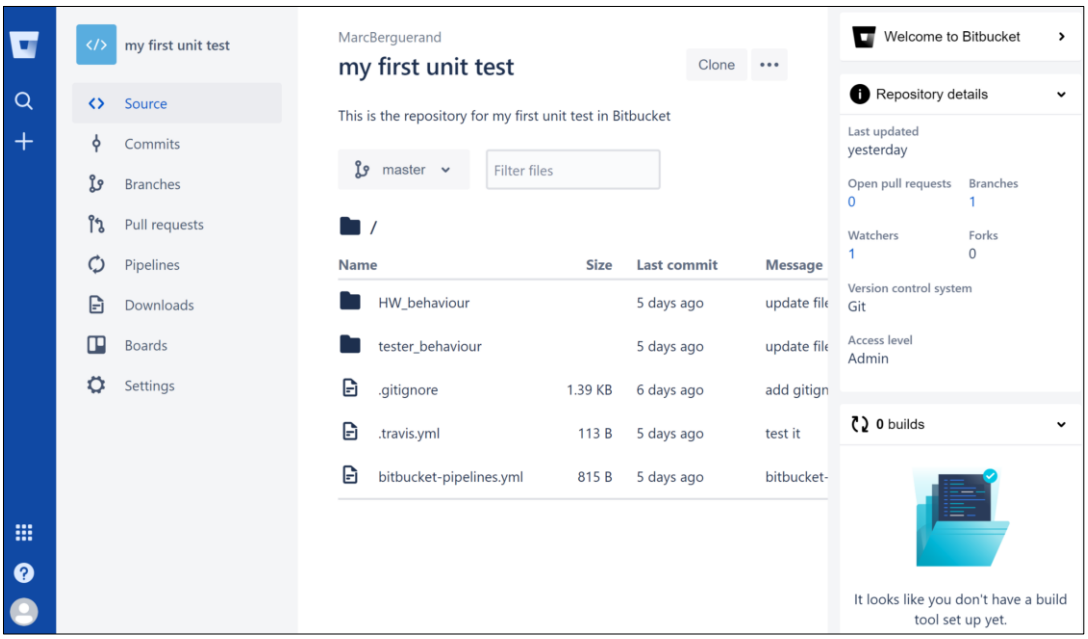


Figure 16: Aperçu de l'interface utilisateur avec le projet associé au répertoire Bitbucket

Annexe n°4

Utilisation de Gitlab CI/CD

Table des matières

Liste des figures

Préambule

Toutes les images apparaissant dans cette annexe proviennent de la plateforme Gitlab. Cette annexe a été réalisée au mois de mai 2019, veuillez prendre en compte que des modifications aient pu avoir lieu au moment où vous lisez ceci.

1 Caractéristique de Gitlab-CI/CD

Voici diverses informations propres à Gitlab-CI :

- Il fait directement partie de Gitlab, ce qui simplifie le déploiement de celui-ci.
- Il est open core
- Il y a un guide d'utilisation très complet, avec un support très compétent en la matière
- Il est scalable : les tests peuvent être exécutés sur diverses machines, autant que l'on veut
- Les tâches peuvent être exécutées en parallèle
- Il est optimisé pour le déploiement

Et d'autres informations concernant ses propriétés :

- Multi-plateforme : Il est possible de lancer les tests sous n'importe quel environnement, tant qu'il supporte le Go
- Multi-langage : Les commandes fonctionnent sous n'importe quel langage
- Stable : Les étapes d'intégrations/déploiements tournent sur d'autres machines que la plateforme elle-même
- Informations en temps réel : mise à jour dynamique des états des divers étapes d'intégration/déploiement
- Pipeline configurable : Il est possible de définir les liens entre les diverses étapes et des conditions pour l'exécution de celles-ci
- Sauvegarde des « artifacts » : enregistre les fichiers souhaités durant les diverses étapes (voir point 3.2 Utiliser des artifacts)
- Tests locaux : Il y a la possibilité de reproduire les tests localement
- Supporte Docker : L'utilisation d'images Docker est simple, également l'utilisation de Kubernetes
- Choix des machines de CI/CD : les « Runners » (voir point 5 Utilisation de son propre « Runner »), peuvent provenir de Gitlab, tout comme d'une propre machine.

2 Intégrer Gitlab-CI/CD au répertoire

Gitlab propose de base un outil d'intégration/déploiement continu. Pour l'utiliser, il suffit simplement d'avoir un fichier « .gitlab-ci.yml » dans le répertoire. Le contenu de ce fichier sera expliqué au chapitre 2.3.

2.1 Fichier « .gitlab-ci.yml » déjà intégré

Dans la figure ci-dessous, le rectangle rouge montre que l'outil CI/CD est utilisé, car le fichier (encadré en vert) est déjà intégré.

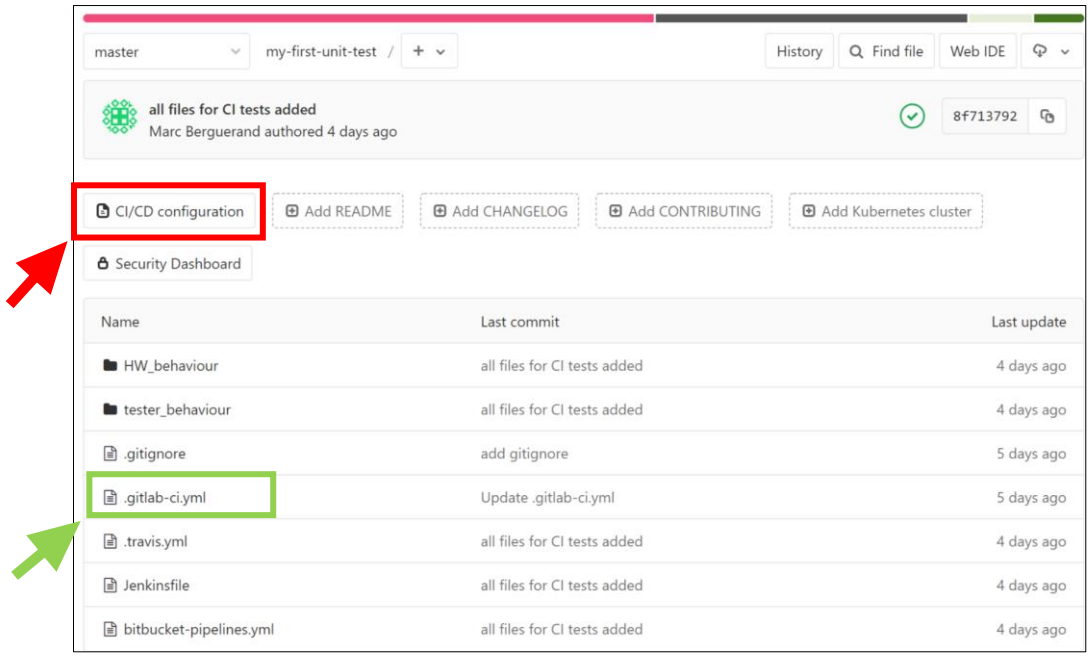


Figure 1: Aperçu du répertoire Gitlab avec le fichier .gitlab-ci.yml déjà existant

2.2 Fichier « .gitlab-ci.yml » à créer

Si le répertoire ne contient pas encore ledit fichier, il est possible de le générer en cliquant sur « Set up CI/CD » :

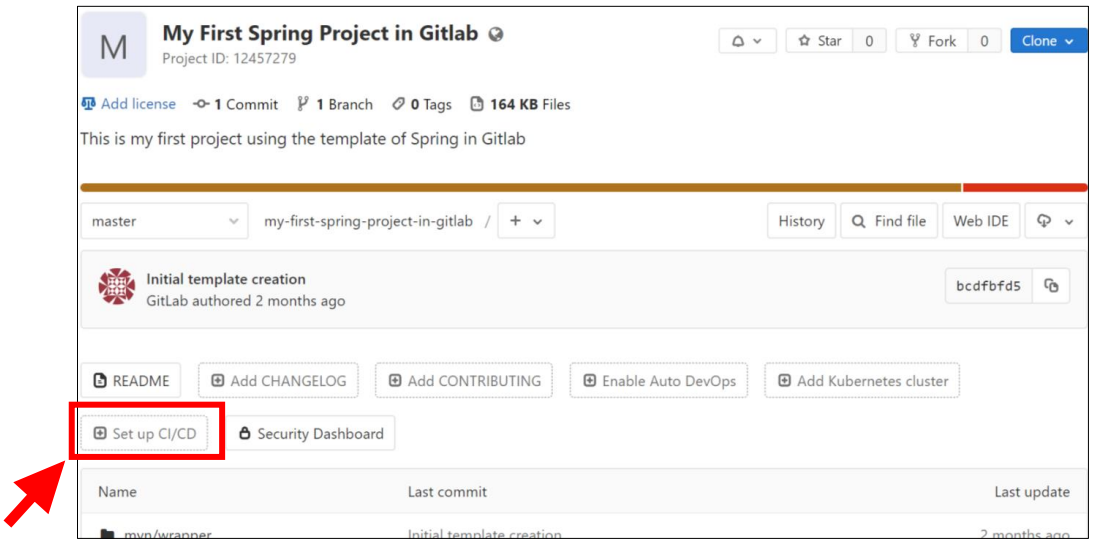


Figure 2: Ajouter l'outil CI/CD à un répertoire Gitlab

À ce moment, la fenêtre ci-dessous apparait. Celle-ci est très utiles dans Gitlab, car elle permet de générer divers fichiers à partir de modèles existants. Dans ce cas-ci, il s'agit d'un modèle de fichier « .gitlab-ci.yml », mais d'autres sont possibles, comme montré à la Figure 4. Au sein même de ce modèle se trouvent d'autres modèles (voir Figure 5) en fonction du type de répertoire. La force de ce générateur est de définir les machines/outils nécessaires au bon déroulement du CI/CD de manière très simple. Une fois la configuration du fichier faite, il suffit de cliquer sur « Commit changes » pour que le CI s'effectue (pour autant que le fichier soit bien formé, plus d'informations à ce sujet au point 3 Contenu du fichier « .gitlab-ci.yml »).

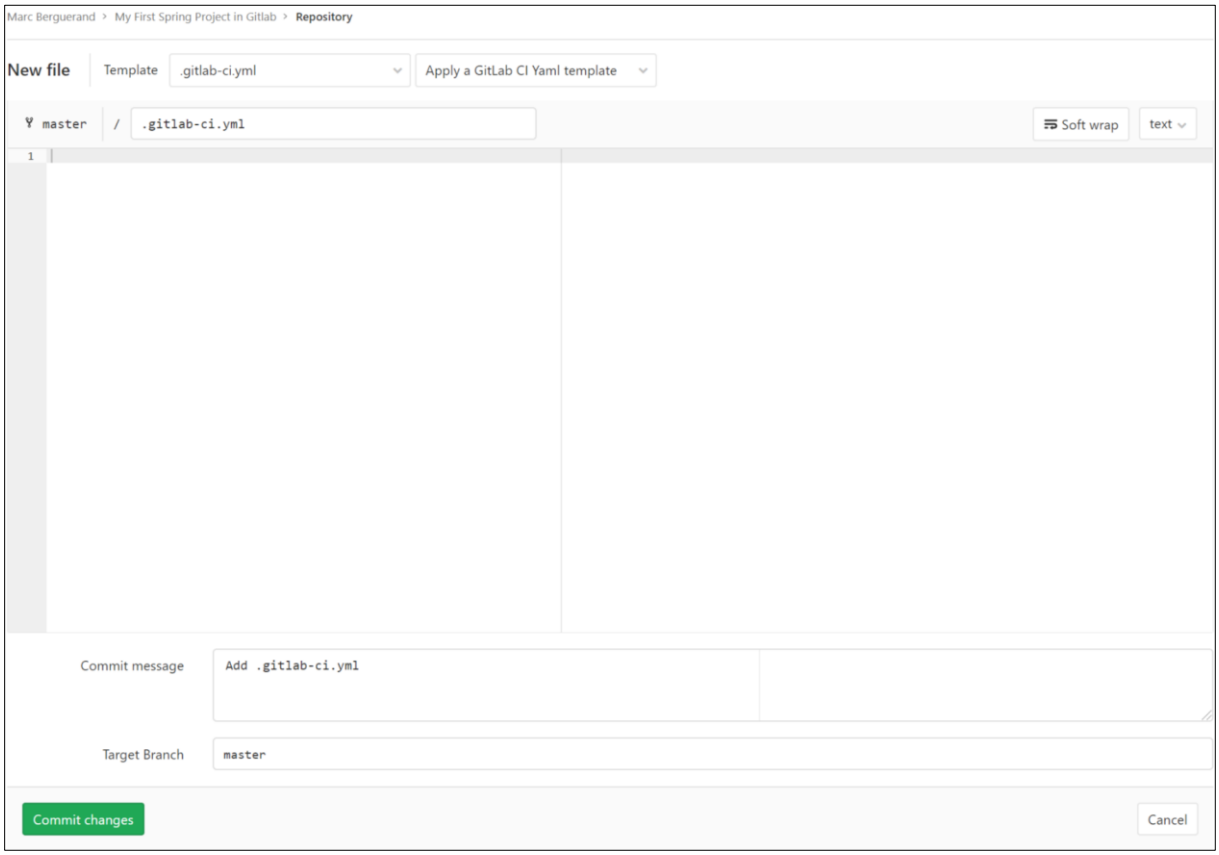


Figure 3: Intégration du CI/CD au répertoire via Gitlab

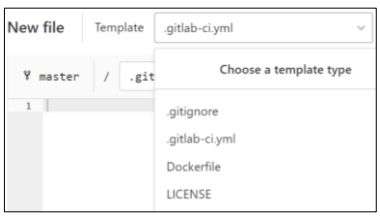


Figure 4: Différents modèles de fichiers

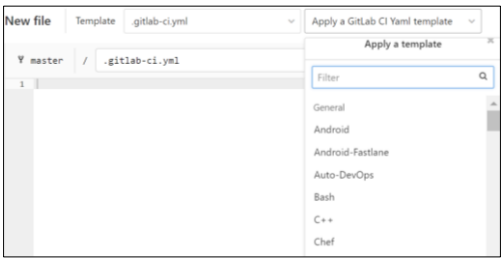


Figure 5: Différents modèles liés au fichier « .gitlab-ci.yml »

2.3 Interface Gitlab-CI/CD dans Gitlab

Pour voir les diverses étapes du pipeline CI du répertoire, il faut aller dans la partie « CI/CD » de celui-ci :

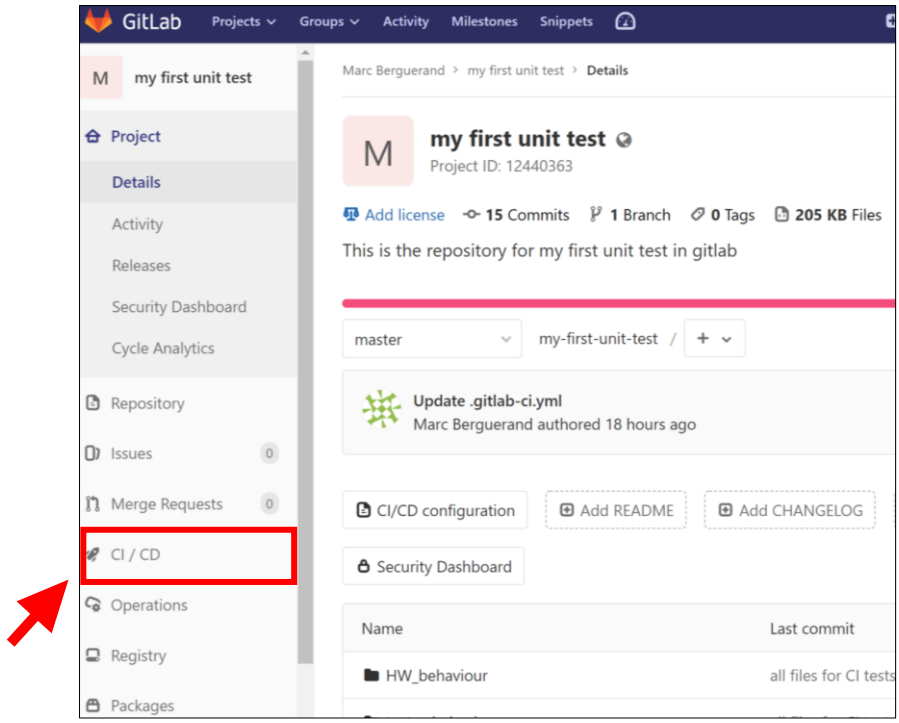


Figure 6: Situation de l'onglet « CI/CD »

Ainsi, la fenêtre suivante apparait :

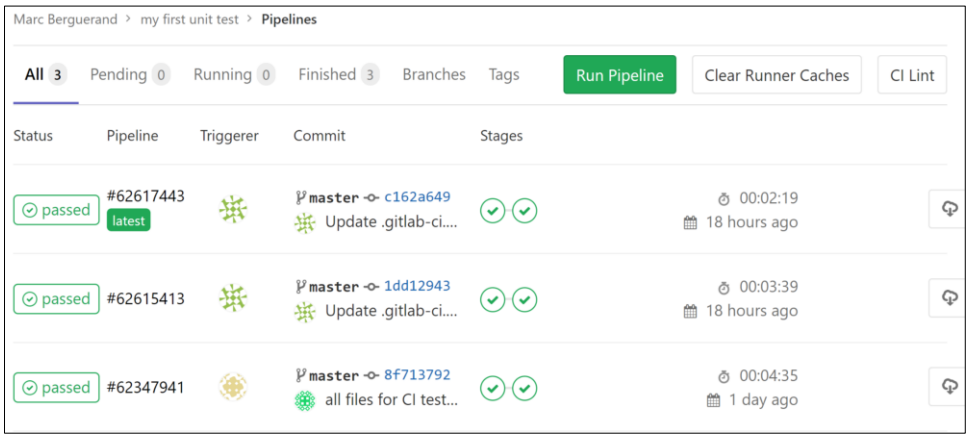


Figure 7: Aperçu du CI du répertoire

Dans ce cas-ci, tous les tests ont passés à chaque fois, car tous les « stages » sont verts. Dans le point 4.2, il y a des exemples de tests échoués.

Pour avoir de plus amples informations sur les tests effectués, il suffit de cliquer sur le statut :

Figure 8: Situation du status dans l'interface CI/CD

Ainsi, la fenêtre ci-dessous apparait. Il est possible d'obtenir de plus amples informations en cliquant sur les diverses étapes du CI (par exemple sur « Building the code », encadré en rouge).

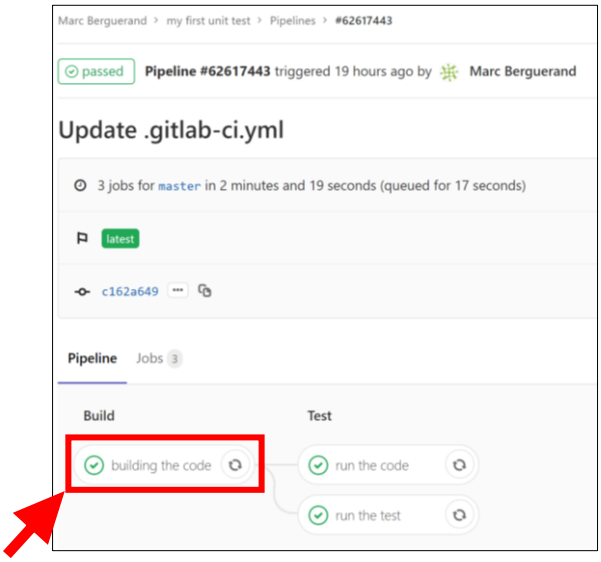


Figure 9: Aperçu de la fenêtre résumant le CI/CD pour un répertoire

En cliquant dessus, les informations ci-dessous sont affichées.

La partie noire correspond à ce qui est exécuté dans l'environnement exécutant le CI.

Comme affiché dans la dernière section de l'environnement CI, des artifacts sont téléversés depuis celui-ci vers Gitlab CI. De plus amples informations concernant ceux-ci sont expliquées au point 3.2. Pour gagner du temps dans l'aperçu des résultats des diverses étapes, Gitlab-CI offre la possibilité de changer directement de stage (en sélectionnant la liste « Pipeline », encadré en rouge ci-dessous, résultat à la Figure 11). Les différentes options correspondent aux diverses étapes configurées dans le fichier « .gitlab-ci.yml » (voir point 3).

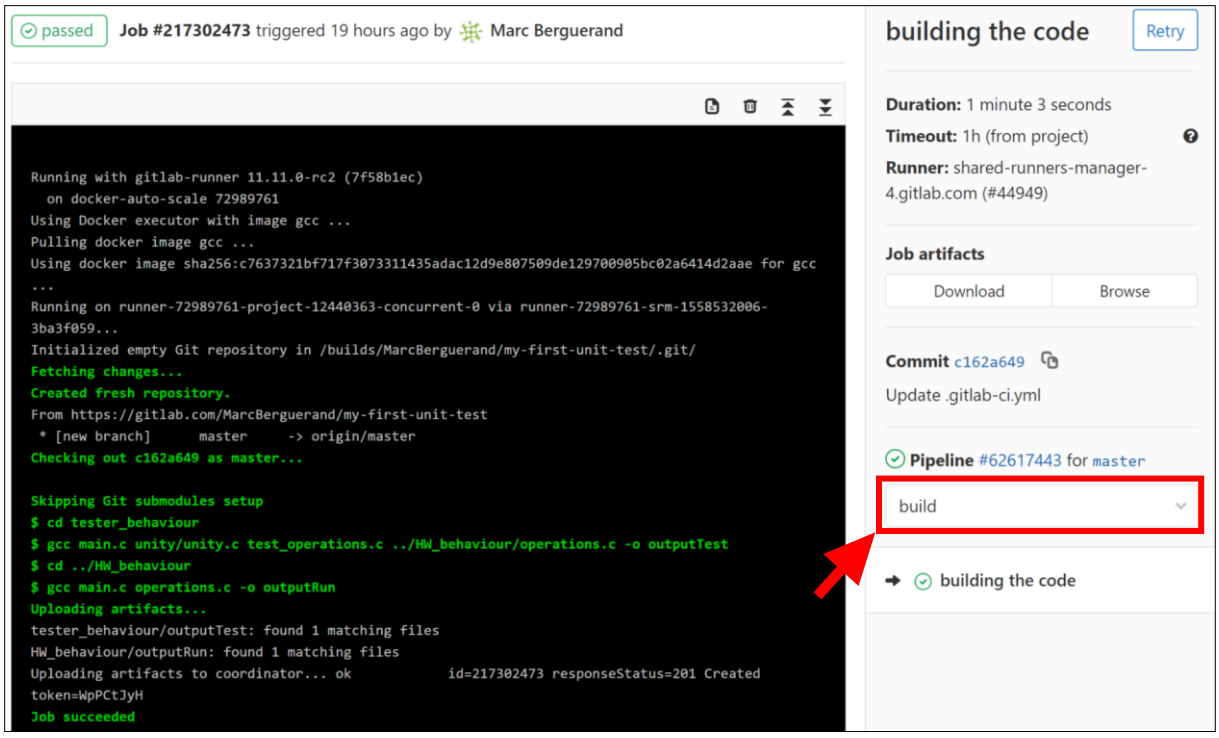


Figure 10: Détails de l'exécution d'une étape

Figure 11: Menu déroulant pour choisir les diverses étapes

Ainsi, en sélectionnant l'étape « test », il est possible de choisir quel « label » est affiché dans la console virtuelle :

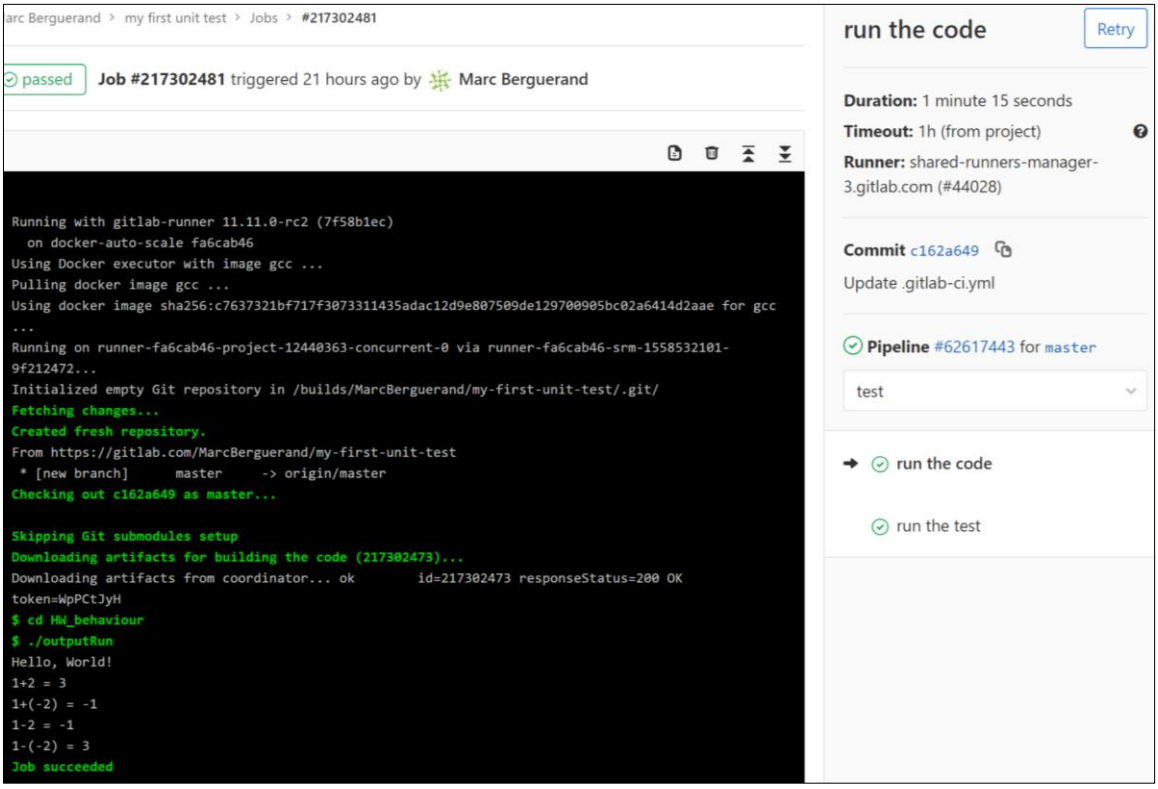


Figure 12: Aperçu de la fenêtre CI sur l'étape « test »

La flèche noire indique quel « label » est affiché. Il est possible de le changer en cliquant sur celui souhaité, ce qui mettra donc à jour la console virtuelle :

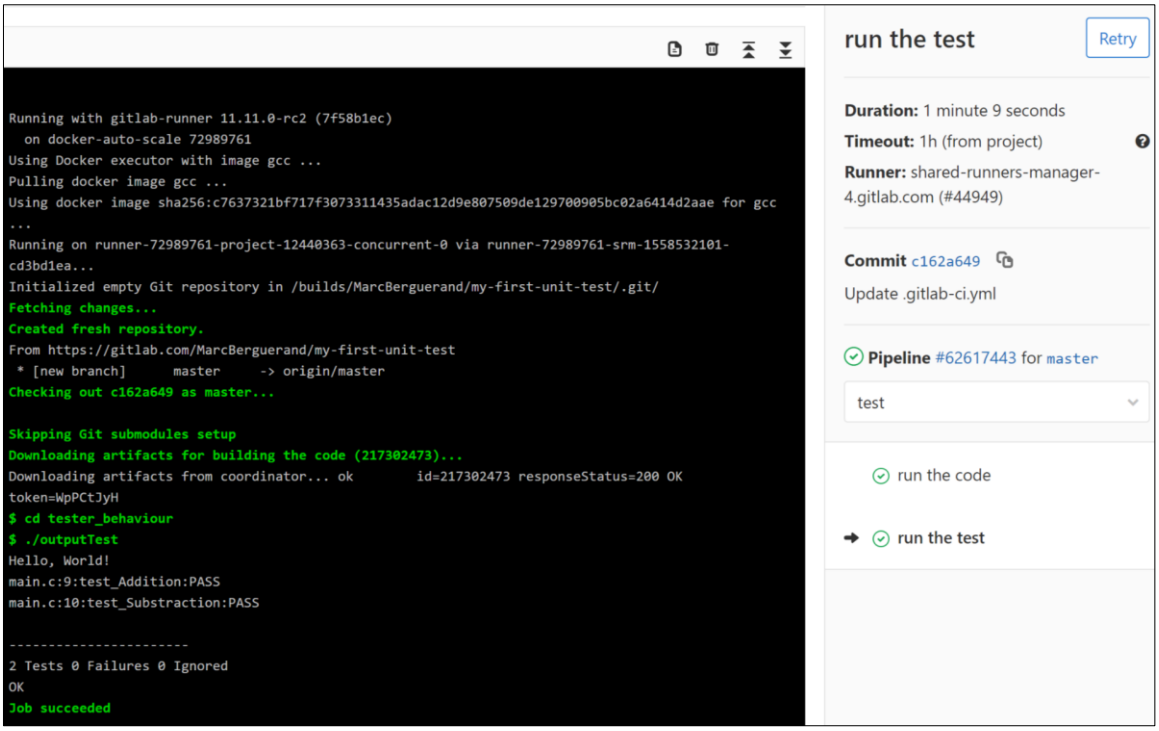


Figure 13: Aperçu de la fenêtre CI sur l'étape « test » avec un label différent

3 Contenu du fichier « .gitlab-ci.yml »

Voici ci-dessous un exemple d'un fichier généré pour un template « C++ », à partir du chapitre 2.2. À noter qu'il faut adapter ce fichier au répertoire désiré.

```
# This file is a template, and might need
#   editing before it works on your project.
# use the official gcc image, based on debian
# can use verions as well, like gcc:5.2
# see https://hub.docker.com/_/gcc/
image: gcc

build:
  stage: build
  # instead of calling g++ directly you can also
  #   use some build toolkit like make
  # install the necessary build tools when needed
  # before_script:
  #   - apt update && apt -y install make autoconf
  script:
    - g++ helloworld.cpp -o mybinary
  artifacts:
    paths:
      - mybinary
  # depending on your build setup it's most likely
  #     a good idea to cache outputs to reduce
  #     the build time
  # cache:
  #   paths:
  #     - "*.o"

# run tests using the binary built before
test:
  stage: test
  script:
    - ./runmytests.sh
```

Figure 14: Exemple de la génération du fichier « .gitlab-ci.yml »

Voici de brèves explications sur le fichier généré :

- Image : désigne quelles sont les outils/environnements nécessaires pour déterminer les configurations des machines qui vont tester le répertoire.
- Labels « build, test » : Il s'agit d'un label qui sera affiché dans l'UI de Gitlab.
    - Stage : définit quelle est l'étape à laquelle le label associé est exécuté
    - Script : désigne les commandes que la machine va effectuer durant l'étape ayant le label « build/test »
    - Artifacts : ceci indique quels fichiers sont gardés en mémoire par Gitlab CI pour être utilisé plus tard (soit pour le CI en soit, soit pour observer le fichier à la fin de l'exécution du CI)
        - Paths : détermine quels fichiers sont à sauvegarder

Pour avoir une meilleure compréhension du fonctionnement du système, l'exécution du CI pour le répertoire « my first unit test » est effectué et expliqué en détails ci-après.

3.1 Exemple du CI pour le répertoire « my first unit test »

Voici un exemple de configuration possible pour le répertoire « my first unit test », fait dans l'annexe 2 :

```
image: gcc

stages:
  - build
  - test

building the code:
  stage: build
  script:
    - cd tester_behaviour
    - gcc main.c unity/unity.c test_operations.c
          ../HW_behaviour/operations.c -o outputTest
    - cd ../HW_behaviour
    - gcc main.c operations.c -o outputRun

  artifacts:
    paths:
      - tester_behaviour/outputTest
      - HW_behaviour/outputRun

run the test:
  stage: test
  script:
    - cd tester_behaviour
    - ./outputTest

run the code:
  stage: test
  script:
    - cd HW_behaviour
    - ./outputRun
```

Figure 15: Exemple d'un fichier « .gitlab-ci.yml » pour le répertoire « my first unit test »

Pour mieux comprendre ce qu'interprète le CI, voici une représentation graphique du résultat :

Figure 16: Représentation graphique du CI par Gitlab

Ainsi, les « labels » simplifient la lecture des tâches effectuées par le CI. Les « stages » déterminent à quel moment est effectué chaque « label ». Il est possible de remplir par n'importe quoi les « labels » et « stages ».

Les « artifacts » sont, dans ce cas, les fichiers compilés des différents projets (un des projets est l'exécution d'additions et de soustractions, l'autre est le testeur des méthodes d'additions et de soustractions). Pour que l'étape « Test » puisse exécuter les fichiers sans devoir se soucier de la compilation, une étape a été effectuée auparavant (l'ordre d'exécution est donné sous le label « stages »). Ceci permet également de mieux déterminer d'où proviendrait une éventuelle erreur, à la compilation ou à l'exécution.

Le script correspond simplement à ce que la machine va effectuer pour chaque étape/label. Les labels peuvent être lancés en parallèle (ce qui est le cas lorsqu'ils sont définis dans le même « stage », comme le sont les labels « run the code » et « run the test »). Il est important de mentionner le fait que chaque « labels » effectués dans un même « stage » est effectué en même temps, mais le changement de « stage » ne se fait que lorsque tous les « labels » auront été effectués en son sein. Ainsi, si une tâche prend beaucoup plus de temps que d'autres à être exécutée, il serait intéressant de faire d'abord s'exécuter toutes les autres puis finalement celle qui prend le plus de temps, afin de se rendre compte au plus vite s'il y a des erreurs dans les autres tâches. Voici une illustration pour mieux comprendre :

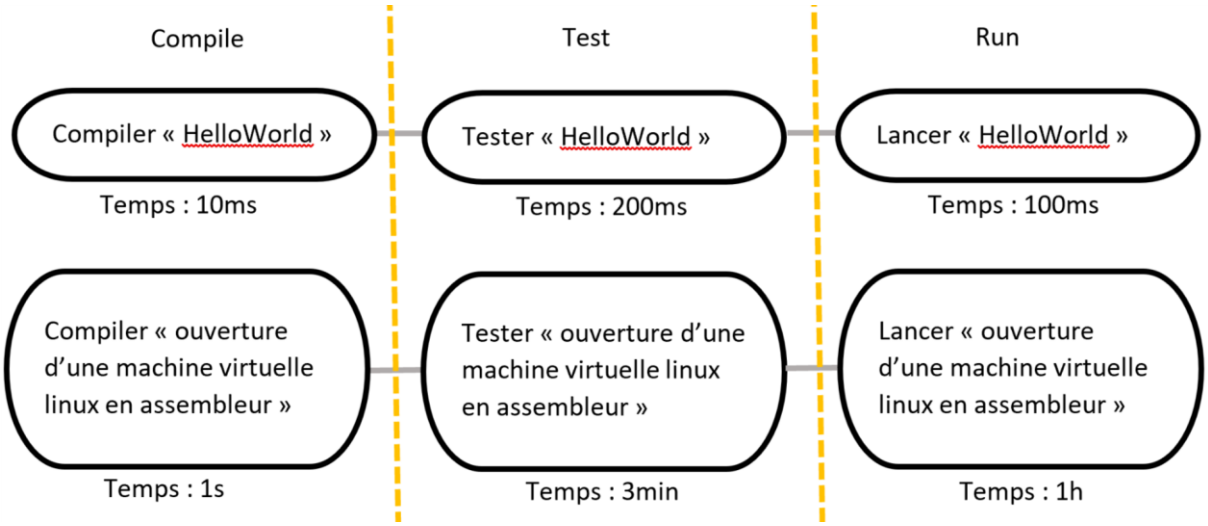


Figure 17: Exemple de temps nécessaire pour le CI

Dans cet exemple, le programme « Hello World » pourrait avoir fini son exécution en 310ms, mais comme il y a un autre programme en parallèle, il doit attendre que celui-ci s'exécute. Ainsi, « Hello World » finira son exécution en une heure, trois minutes et une seconde (Lorsqu'une étape est terminée, l'on peut directement voir le résultat, pas besoin d'attendre que les autres étapes soient terminées, c'est uniquement pour les changements d'états qu'il faut attendre).

Ainsi, dans l'exemple de la Figure 17, il serait plus judicieux de rajouter deux étapes, pour la compilation et le test du logiciel « Hello World », puis de commencer le parallélisme avec le lancement de « Hello World » et la compilation de l'autre programme.

3.2 Utiliser des artifacts

Les artifacts ont été mentionnés au chapitre 2.3. Pour mieux illustrer ceux-ci, la Figure 17 va être utile. En effet, il y a différentes étapes dans ce CI (Compile, Test et Run). Sans les artifacts, il faudrait recompiler le code à chaque étape. En revanche, en utilisant ceux-ci, la compilation ne se ferait uniquement à l'étape « Compile », puis le fichier résultant est défini comme artifact. Ainsi, il peut être utilisé tel quel dans les autre étapes (Test et Run).

Il y a également un autre avantage avec Gitlab-CI : Il est possible de télécharger ces artifacts une fois qu'ils ont été générés (voir Figure ci-dessous, en cliquant sur l'encadré rouge). Une autre possibilité est d'avoir un aperçu des artifacts, en cliquant où se trouve l'encadré vert.

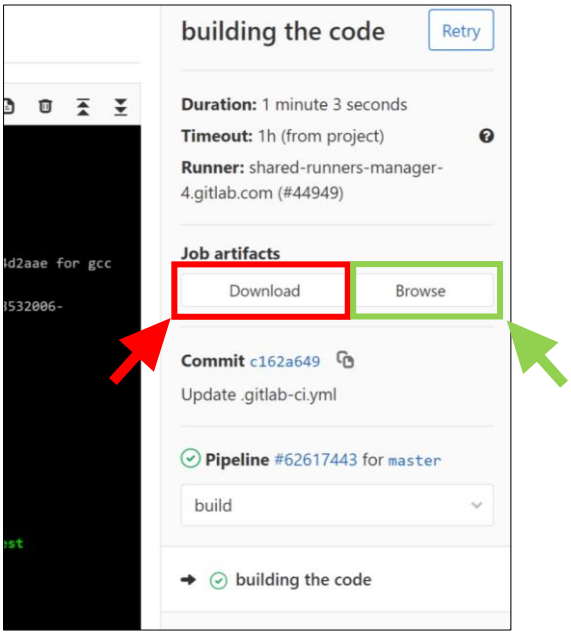


Figure 18: Situation de l'onglet « téléchargement des artifacts »

Ainsi, si un fichier est généré dans une étape (par exemple un fichier JSON contenant des informations sur le déroulement du code, ou un fichier CSV où des données sont introduites durant l'exécution dudit code), il est possible de les télécharger et de regarder leurs contenus. Ceci permet d'avoir une vision supplémentaire sur les résultats liés à l'exécution du code. Ces artifacts ne disparaissent pas, ils restent dans l'historique des pipelines de Gitlab-CI.

4 Déploiement de Gitlab-CI

Il est important de comprendre ce qui se passe lors du CI. En effet, lorsque le CI est effectué, il est préférable de savoir où le code est effectué. Voici le résultat d'une étape du CI :

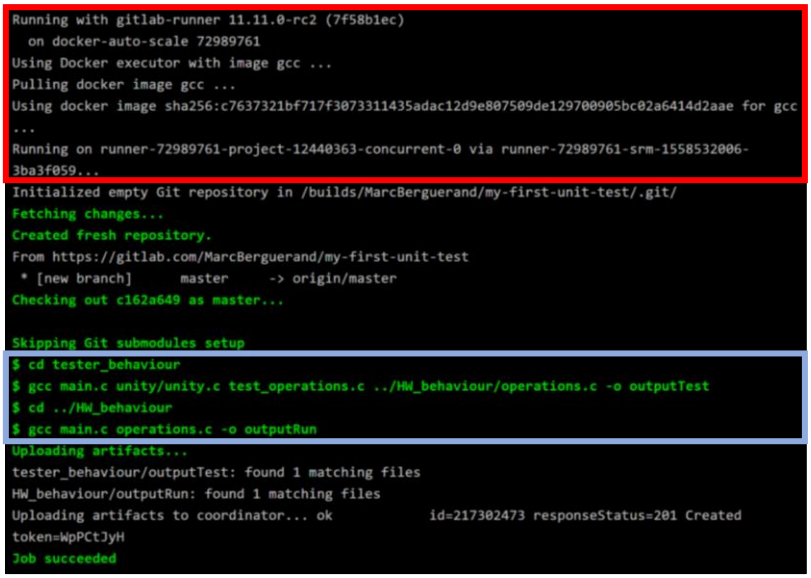


Figure 19: Aperçu de l'invite de commande virtuelle reprenant les résultats des tests

4.1 Machine utilisée pour le CI

Ainsi, au tout début de la Figure 19 (encadré en rouge), sont écrites les informations de la machine qui exécute le code. Cette machine est définie en fonction du fichier « .gitlab-ci.yml ». Gitlab a donné un nom à celle-ci : un « Runner ». Ainsi, un Runner est sélectionné en fonction de son contenu. Si le programme est codé en langage C, la machine devrait être équipée, par exemple, de gcc. La plateforme va alors rechercher parmi tous ces Runners la première machine libre qui peut exécuter le CI demandé.

Le point fort de Gitlab est qu'il permet de configurer notre propre Runner sur une de nos machines. Ainsi, nous devons installer les outils nécessaires au bon déroulement du CI. L'avantage d'avoir son/ses prope/s Runner/s est qu'il n'y a plus besoin d'attendre qu'un Runner de Gitlab soit libre (ce qui peut prendre du temps s'il y a beaucoup d'utilisateurs qui doivent utiliser la même configuration), et que l'on peut mettre exactement la configuration souhaitée, qui pourrait différer d'une configuration standard d'un Runner mis à disposition. Pour de plus amples informations concernant les Runners, voir le point 5.

4.2 Exécution du CI

À la suite de l'encadré rouge, après les importations du répertoire à tester sur la Figure 19, se trouve le script effectué (encadré bleu). Celui-ci est celui entré dans le fichier « .gitlab-ci.yml », avec les résultats affichés, tels que dans une invite de commande. Dans cet exemple, il y a encore à la suite de ceci les artifacts à sauvegarder pour les étapes suivantes. Il s'en suit le résultat de l'opération, si tout est en ordre alors un « Succeed » apparait, sinon une erreur avec l'indication d'où elle provient, en fonction de la commande entrée, ou de l'exécution de ladite commande (voir ci-dessous).

Veuillez prendre note que les figures ci-dessous sont représentatives du framework de test « Unity ».

Afin d'échouer à un test, le fichier de tests a été modifié afin d'inclure une erreur à la soustraction, en disant que -5-5 = 0, ce qui est bien évidemment faux :

```
$ ./outputTest
Hello, World!
main.c:9:test_Addition:PASS
main.c:16:test_Substraction:FAIL: Expected 0 Was -10

-----------------------
2 Tests 1 Failures 0 Ignored
FAIL
ERROR: Job failed: exit code 1
```

Figure 20: Tests échoués

Et voici le résultat lorsque les tests sont réussis :

```
$ ./outputTest
Hello, World!
main.c:9:test_Addition:PASS
main.c:10:test_Substraction:PASS

-----------------------
2 Tests 0 Failures 0 Ignored
OK
Job succeeded
```

Figure 21: Tests réussis

5 Utilisation de son propre « Runner »

Gitlab-CI offre la possibilité d'avoir sa propre machine qui va effectuer les tests CI. Avec ceci, la dépendance envers les machines offertes par Gitlab n'a plus lieu. En effet, étant donné qu'il y a énormément d'utilisateurs de Gitlab, beaucoup de tests sont effectués sur les machines mise à dispositions. Pour rappel, l'utilisation des machines offertes par Gitlab est offerte pendant 2000 minutes par mois. Ensuite, un tarif est appliqué s'il est souhaité de continuer à utiliser lesdites machines. En revanche, un Runner « interne » n'est pas payant. Ainsi, il est souhaitable de lier une machine physique à Gitlab pour effectuer ses propres tests.

Pour lier une machine physique en tant que Runner pour Gitlab, il faut installer gitlab-runner :

```
sudo apt-get update
sudo apt-get install gitlab-runner
```

Figure 22: Commandes pour installer gitlab-runner

5.1 Lier une machine en tant que « Runner »

Afin de pouvoir profiter de ses propres Runners, il faut les enregistrer. Pour se faire, il faut taper la commande suivante :

```
sudo gitlab-runner register
```

Figure 23: Commande pour enregistrer un runner

Il s'en suit tout une suite de champs à remplir. En voici l'intégralité, pas à pas :

Tout d'abord, il faut alors insérer le domaine sur lequel le répertoire voulant faire du CI se trouve (dans l'exemple ci-dessous, sur gitlab.com).

```
Runtime platform                    arch=amd64 os=linux
pid=27784 revision=6946bae7 version=12.0.0
Running in system-mode.

Please enter the gitlab-ci coordinator URL (e.g.
https://gitlab.com/):
https://gitlab.com
```

Figure 24: Configurer un runner - domaine d'hébergement

Ensuite doit être transmis le token du répertoire voulant effectuer le CI. Pour le connaitre, il faut aller sur gitlab, sélectionner un projet puis aller dans les paramètres, sous CI/CD :

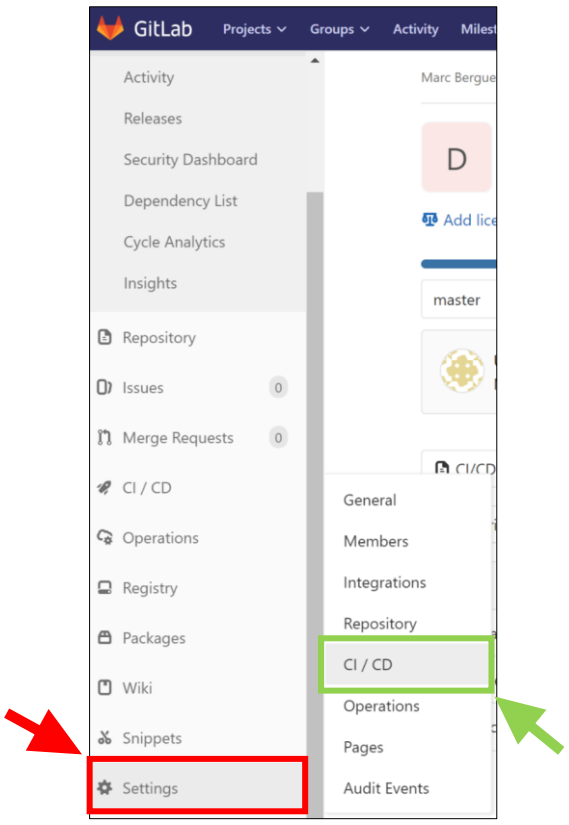


Figure 25: Configurer un runner - trouver le token partie 1

Puis, il faut aller sous la section « Runners » :

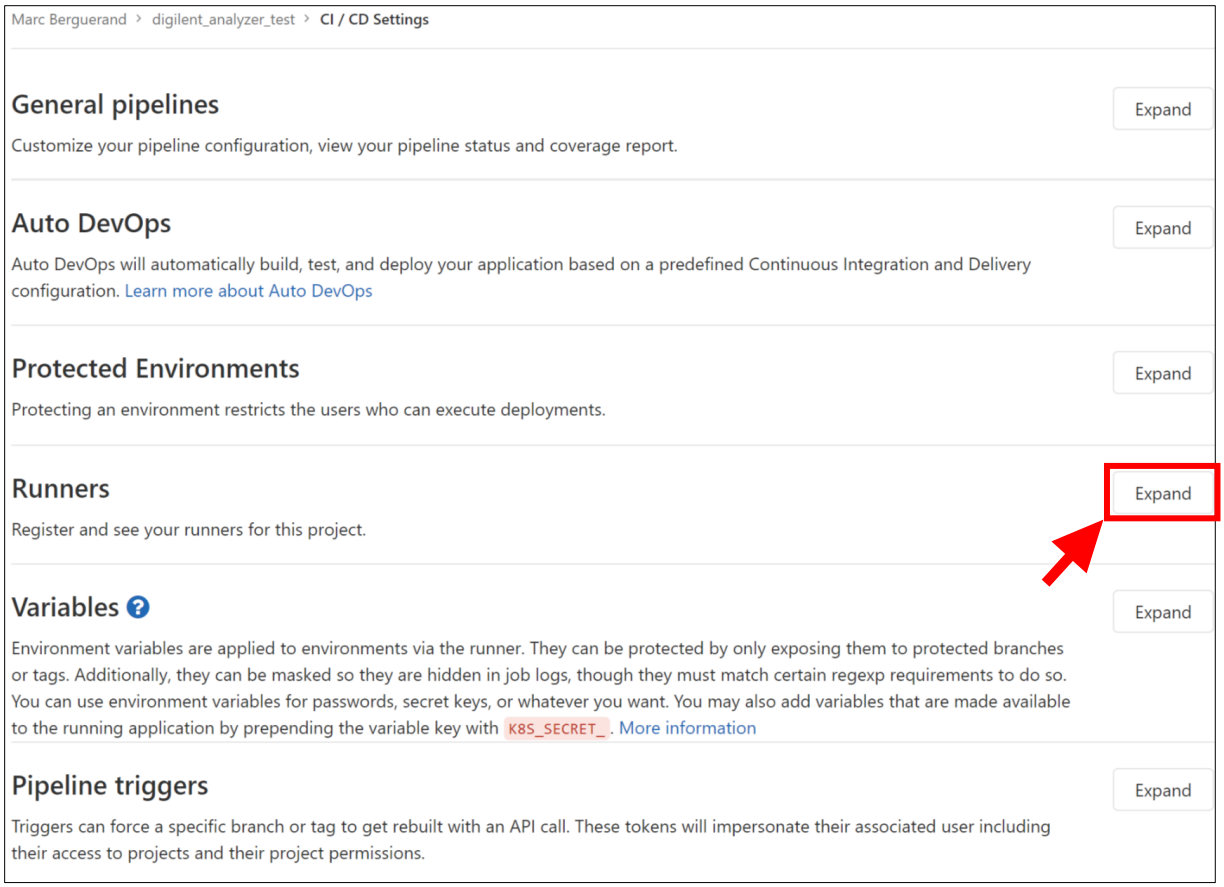


Figure 26: Configurer un runner - trouver le token partie 2

Et le token se trouve dans la partie « Set up a specific Runner manually » :

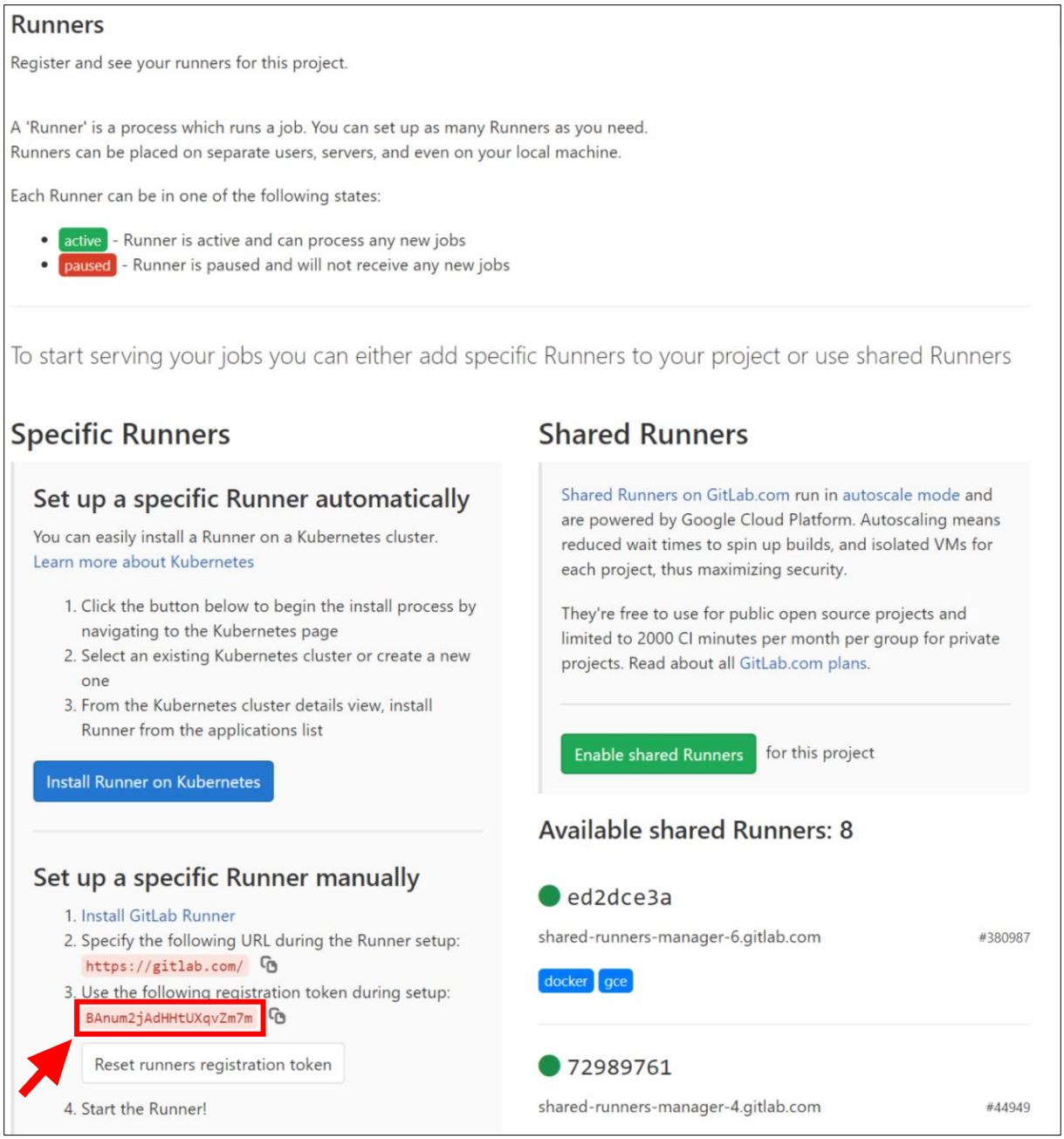


Figure 27: Configurer un runner - trouver le token partie 3

Il faut alors entrer le token dans la configuration :

```
Please enter the gitlab-ci token for this runner:
BAnum2jAdHHtUXqvZm7m
```

Figure 28: Configurer un runner - token du répertoire

Ensuite est demandé une description du runner, afin de voir de manière très simple en quoi il consiste dans l'interface de Gitlab pour les Runners :

```
Please enter the gitlab-ci description for this runner:
[marcberguera-OptiPlex-7050]: exemple of runner with docker
image
```

Figure 29: Configurer un runner – description

Puis les tags associés au runner, qui serviront à déterminer quel runner peut être utilisé pour quelle étape du CI :

```
Please enter the gitlab-ci tags for this runer (comma
separated):
docker, example
```

Figure 30: Configurer un runner – tags

Une fois ceci fait, le runner est alors enregistré :

```
Registering runner... succeeded     runner=BAnum3jA
```

Figure 31: Configurer un runner – enregistré

Il reste toutefois à spécifier de quel type de runner il s'agit :

```
Please enter the executor: parallels, virtualbox,
docker+machine, docker-ssh+machine, docker, docker-ssh,
shell, ssh, kubernetes:
docker
```

Figure 32: Configurer un runner - choisir le type de runner

Dans le cas d'un runner exécutant des images Docker, il faut alors entrer « docker », puis indiquer quelle est l'image qui est utilisée, en sachant que l'image provient de la plateforme DockerHub :

```
Please enter the default Docker image (e.g. ruby 2.0):
MarcBerguerand/example_image:v1.0
```

Figure 33: Configurer un runner - définir l'image docker à exécuter

Apparait alors la confirmation indiquant que le runner est bien configuré :

```
Runner registered successfully. Feel free to start it, but
if it's running already the config sould be automatically
reloaded!
```

Figure 34: Configurer un runner - message final de réussite

5.2 Activer les runners

Afin de lancer les runners pour qu'ils puissent exécuter le CI sur la machine physique, il faut utiliser cette commande :

```
sudo gitlab-runner run
```

Figure 35: Commande pour activer les runners sur la machine physique

6 Gérer les Runners

D'une fois qu'un runner a été lié à un projet, il n'est pas possible de l'utiliser directement pour d'autres projets. Heureusement, Gitlab offre une possibilité simple pour réaliser ceci.

Il faut aller sous le Runner que l'on souhaite partager entre nos projets, dans la partie concernant ceux-ci, soit où l'on trouve le token pour le projet (cf. Figure 25-Figure 26-Figure 27). Puis, diverses options sont possibles pour chaque runner :

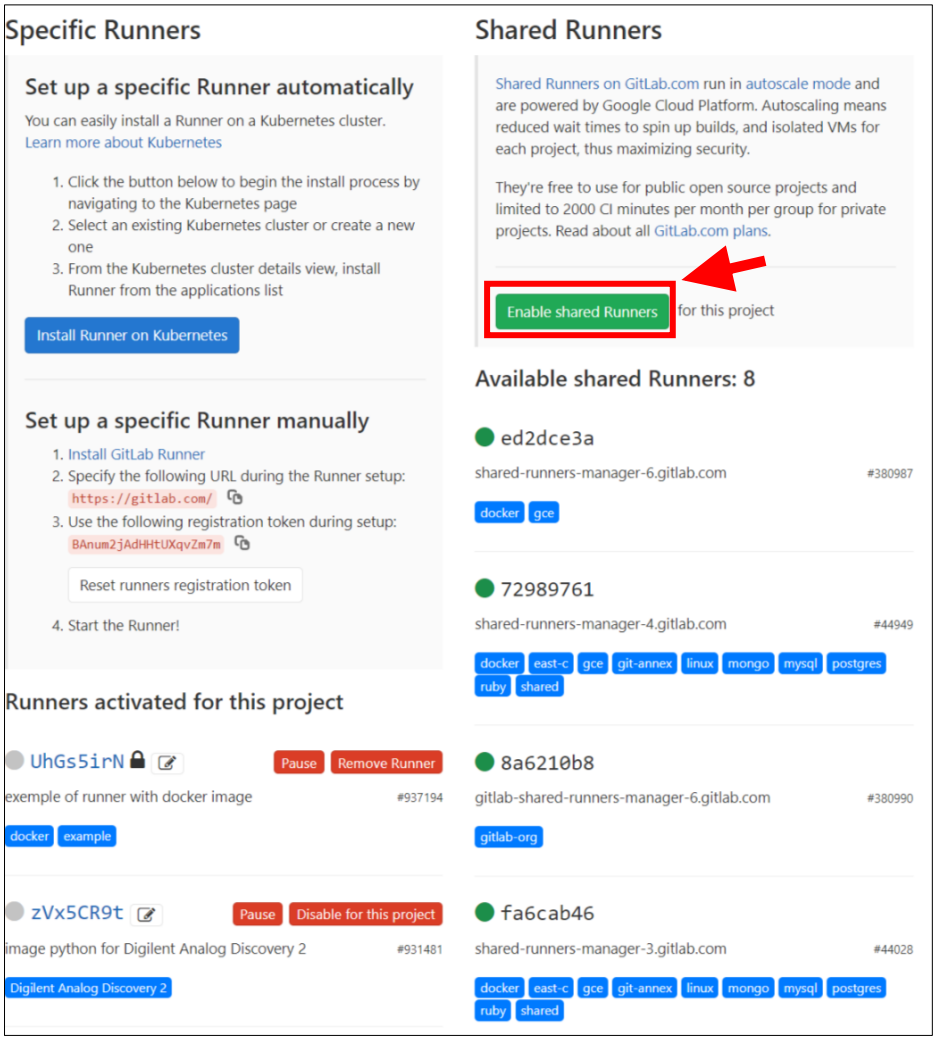


Figure 36: Gestion des runners - liste de ceux-ci

Dans la colonne de gauche se trouvent tous les runners liés au projet sur lequel l'on se trouve. Dans celle de droite, il s'agit des runners que Gitlab met à disposition. Il est possible d'activer/désactiver l'utilisation des runners de Gitlab en cliquant sur la partie en rouge dans la figure ci-dessus.

6.1 Paramètres des runners pour le projet

Attardons-nous sur la colonne de gauche, celle qui concerne les runners propres au projet :

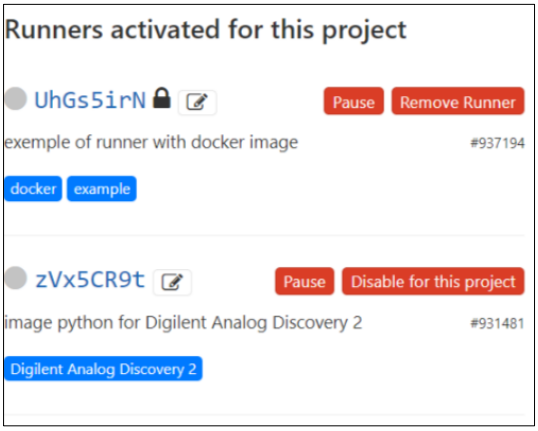


Figure 37: Runners propres au projet

La première option est qu'il est possible de mettre en pause des runners, en cliquant simplement sur « Pause ». Ainsi, ce runner ne sera pas dans la liste des runners disponibles pour le CI, et l'état de celui-ci est le suivant :

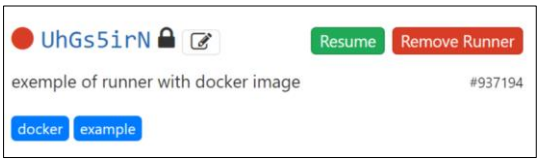


Figure 38: Runner mis en pause

Pour l'activer, il faut alors cliquer sur « Resume ».

Comme le montre la Figure 37, il y a déjà une distinction entre les deux runners. L'un d'entre eux a un cadenas et l'autre non. De base, le cadenas apparait. Ceci signifie que le runner est verrouillé pour ce projet spécifique, et qu'il est possible de l'enlever en cliquant sur l'icône encadrée en rouge ci-dessous:

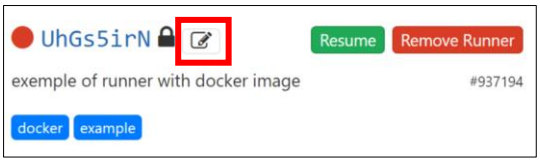


Figure 39: Modifier les propriétés d'un runner - partie 1

Ainsi, la fenêtre ci-dessous apparait. Depuis celle-ci, il est possible de cocher/décocher des checkboxs :

- Active : définis s'il est en pause ou non
- Protected : utilise le runner seulement sur des branches protégées
- Run untagged jobs : définis si le runner peut exécuter des étapes qui n'ont aucun tag (le tag est expliqué au point 6.2)
- Lock to current projects : définis si le runner est verrouillé pour le projet actuel

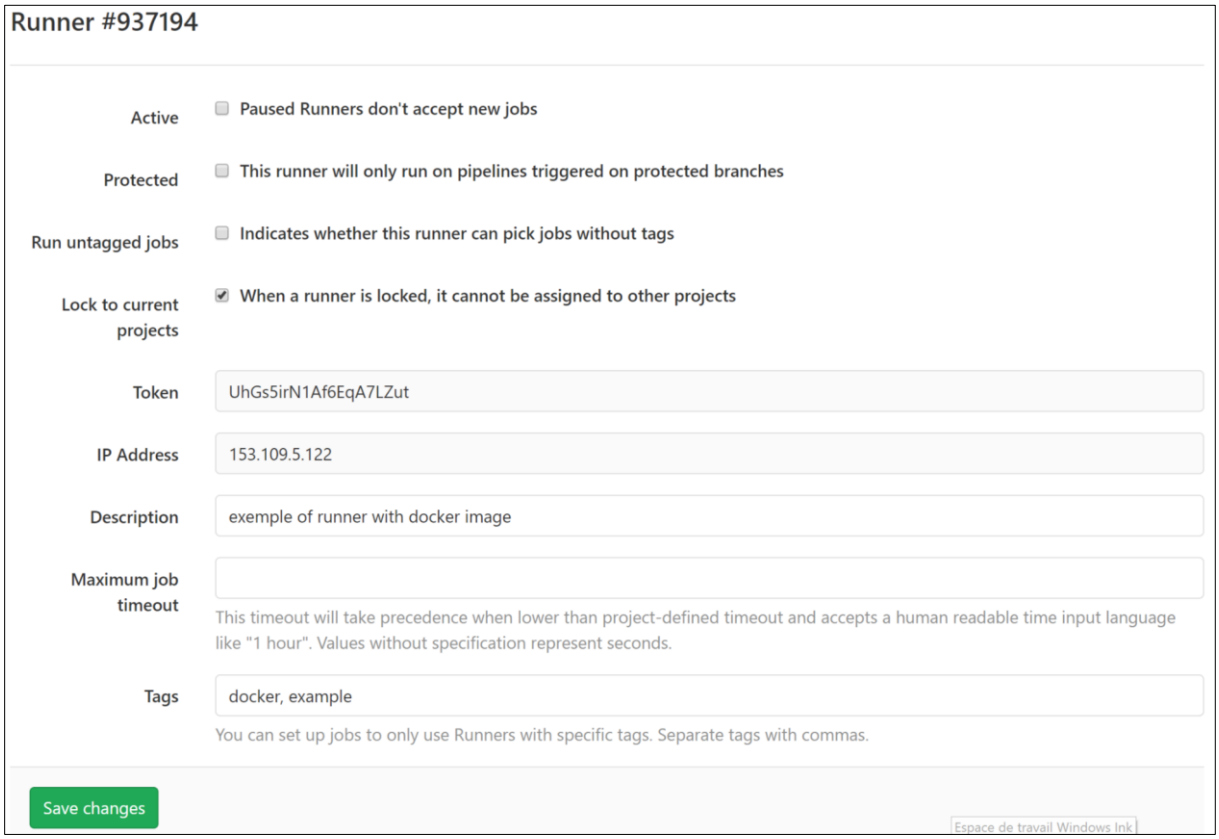


Figure 40: Modifier les propriétés d'un runner - partie 2

Ainsi, en décochant la dernière checkbox, le runner est disponible pour tous les autres projets personnels. Il faut cependant l'activer dans ceux où l'on souhaite utiliser le dit runner, en allant dans les paramètres CI/CD du projet souhaité.

6.2 Utilisation des tags

Les tags permettent de déterminer quel runner est utilisé pour chaque étape. Par exemple, si l'on souhaite utiliser une image docker ayant ubuntu, le tag « Ubuntu » devrait être spécifié dans les tags de l'étape au sein du fichier « gitlab-ci.yaml » (voir figure ci-dessous) et dans le tag du runner (par exemple ou se trouvent les mots « docker, example » dans la figure ci-dessus).

```yaml
stages:
  - program


execute the makefile:
  stage: program

  script:
    - python3 digilent_analyzer.py
  tags:
    - Ubuntu
```

Figure 41: Utilisation de tags dans le fichier "gitlab-ci.yaml"

7 Paramètres supplémentaires concernant les runners

En enregistrant les runners, uniquement quelques paramètres sont définis (les tags, une description, un exécuteur, un token). Mais il y en a d'autres que l'on peut modifier. En effet, ceux-ci sont enregistrés dans un fichier « config.toml ». Celui-ci se trouve, dans le cas d'une utilisation sous Ubuntu, dans le répertoire « /etc/gitlab-runner/ ».

Voici un aperçu dudit fichier pour un runner docker :

```
concurrent = 4
check_interval = 0

[session_server]
  session_timeout = 1800

[[runners]]
  name = "exemple of runner with docker image"
  url = "https://gitlab.com"
  token = "BAnum2jAdHHtUXqvZm7m"
  executor = "docker"
  [runners.custom_build_dir]
  [runners.docker]
    tls_verify = false
    image = "MarcBerguerand/example_image:v1.0"
    privileged = false
    disable_entrypoint_overwrite = false
    oom_kill_disable = false
    disable_cache = false
    volumes = ["/cache"]
    shm_size = 0
  [runners.cache]
    [runners.cache.s3]
    [runners.cache.gcs]
```

Figure 42: Fichier config.toml

Ainsi, le premier paramètre détermine combien de container peuvent être exécutés simultanément. Un autre paramètre important est le « privileged ». S'il est à « true », alors le container a libre accès à l'intégralité du système de la machine hôte. Attention donc lorsqu'il est mis à « true ».

Pour autant, tous les paramètres ne sont pas remplis à ce moment dans ce fichier. La liste complète se trouve sur la documentation officielle de Gitlab :

https://docs.gitlab.com/runner/configuration/advanced-configuration.html

Par exemple, le paramètre « limit » permet de déterminer le nombre de container pouvant être exécutés simultanément pour chaque runner. Ainsi, si l'un d'entre eux ne peut avoir qu'une seule « instance », il faut insérer « limit = 1 » dans la partie « runners » :

```
[[runners]]
  name = "exemple of runner with docker image"
  limit = 1
  url = "https://gitlab.com"
  token = "BAnum2jAdHHtUXqvZm7m"
  executor = "docker"
```

Figure 43: Ajout du paramètre "limit"

Un autre paramètre intéressant est le « devices ». Celui-ci permet d'accéder à du contenu de la machine hôte, de manière restreinte. Ainsi, uniquement les composants nécessaires peuvent être utilisés. Si l'on souhaite accéder au bus du données USB par exemple, la ligne suivante est à entrer dans la partie « runners.docker »:

```
  [runners.docker]
    tls_verify = false
    image = "MarcBerguerand/example_image:v1.0"
    privileged = false
    devices = ["/dev/bus"]
    disable_entrypoint_overwrite = false
    oom_kill_disable = false
    disable_cache = false
    volumes = ["/cache"]
    shm_size = 0
```

Figure 44: Ajout du paramètre "devices"

7.1 Établissement d'une connexion sécurisée

Afin d'avoir une connexion entre la machine physique et le serveur Gitlab sécurisée, il faut utiliser des certificats. Voici deux cas possibles, l'un étant avec l'utilisation de certificats auto-signés et l'autre pour des certificats officiels.

7.1.1 Utilisation de certificats auto-signés

Dans ce cas-ci, le serveur dispose du certificat racine auto-signé (dans cet exemple, il se nomme « ca_cert.crt »), de son propre certificat signé par le racine et de sa clé privée. La machine physique doit donc connaître le certificat racine, ainsi que la clé publique du serveur, qui a été fournie par le certificat racine. Elle doit ensuite générer une requête de certification pour en obtenir un qui est validé par le racine. Pour ce faire, il faut d'abord créer une clé privée, dans ce cas-ci d'une longueur de 2048 bits :

```
openssl genrsa -out privateKey.pem 2048
```

Figure 45: Génération d'une clé privée

Puis générer la requête de certification avec la clé générée précédemment :

```
openssl req -new -key privateKey.pem -out request.csr
Can't load /home/marcberguera/.rnd into RNG
140036100735424:error:2406F079:random number generator:RAND_load_file: Cannot open
file:../crypto/rand/randfile.c:88:Filename=/home/marcberguera/.rnd
You are about to be asked to enter information that qill be incorporated into your
certificate request.
What you are about to enter is what is called a Distinguished Name or DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:CH
State or Province Name (full name) [Some-State]:Valais Wallis
Locality Name (eg, city) []:Sion
Organization Name (eg, company) [Internet Widgits Pty Ltd]:HES-So Valais/Wallis
Organizational Unit Name (eg, section) []:Diplom thesis
Common Name (e.g. server FWDN or YOUR name) []:Berguerand Marc
Email Address []:marc.berguerand@students.hevs.ch

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

Figure 46: Génération d'une requête de certification

Ensuite, il faut que l'autorité signe la requête et retourne la clé publique, afin de pouvoir utiliser une connexion sécurisée (dans cet exemple, elle est nomée « public_cert.crt »). Cependant, il faut indiquer à Gitlab-runner où se trouvent les fichiers nécessaires au cryptage des données. Ceci se fait dans le fichier « config.toml ». Il faut ajouter dans la partie « runners » les trois points suivants :

```
[[runners]]
  name = "exemple of runner with docker image"
  limit = 1
  url = "https://gitlab.com"
  token = "BAnum2jAdHHtUXqvZm7m"
  tls-ca-file = "etc/gitlab-runner/certs/ca_cert.crt"
  tls-key-file = "etc/gitlab-runner/certs/privateKey.pem"
  tls-cert-file = "etc/gitlab-runner/certs/public_cert.crt"
  executor = "docker"
  [runners.custom_build_dir]
```

Figure 47: Modification de la configuration de Gitlab-runner

7.1.2 Utilisation de certificats officiels

Ceci s'avère beaucoup plus simple qu'au point précédent. En effet, les certificats étant officiels, il n'y a besoin que de télécharger la chaine de certification relative au serveur. Voici le procédé :

Tout d'abord, aller sur le serveur. Puis, télécharger la chaine de certification (Dans le cas de Google Chrome comme explorateur, cliquer sur le petit cadenas, puis sur « Certificat ». Une fenêtre va alors s'ouvrir. Aller sous l'onglet « Chemin d'accès de certification » et enregistrer chacun d'entre eux) :

Figure 48: Location du cadenas

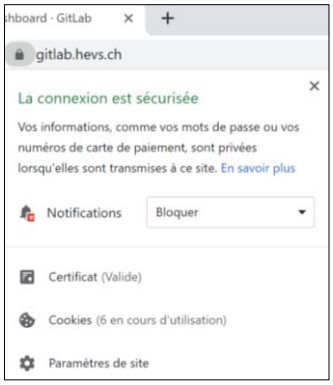


Figure 49: Ouverture des certificats

Figure 50: Informations sur les certificats

Figure 51: Chaine de certification

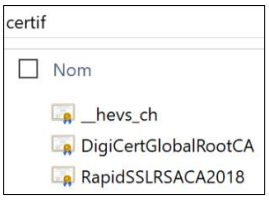


Figure 52: Téléchargement des certificats

Une fois ceci fait, il suffit simplement de créer un nouveau fichier, contenant à la suite tous les certificats précédemment téléchargés (trois dans ce cas-ci) :

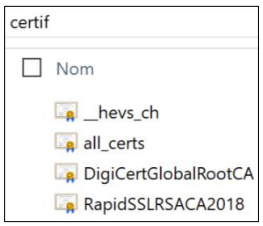


Figure 53: Mise en commun des certificats dans un fichier "all_certs.crt"

Ainsi, il faut définir le chemin pour accéder au fichier « all_certs.crt » dans le fichier « config.toml » utilisé par Gitlab-runner sous « runners »:

```
[[runners]]
  name = "exemple of runner with docker image"
  limit = 1
  url = "https://gitlab.com"
  token = "BAnum2jAdHHtUXqvZm7m"
  tls-ca-file = "etc/gitlab-runner/certs/all_certs.crt"
  executor = "docker"
  [runners.custom_build_dir]
```

Figure 54: Mise à jour de la configuration de Gitlab-runner

Annexe n°5

Utilisation de Bitbucket- Pipeline

Table des matières

Liste des figures

Préambule

Toutes les images apparaissant dans cette annexe proviennent de la plateforme Bitbucket. Cette annexe a été réalisée au mois de mai 2019, veuillez prendre en compte que des modifications aient pu avoir lieu au moment où vous lisez ceci.

1 Caractéristique de Bitbucket-Pipeline

Voici diverses informations propres à Bitbucket-Pipeline :

- Le CI/CD est intégré nativement à Bitbucket
- Configuration et Utilisation simple
- Intégration avec la suite Jira rapide
- Prise en charge de toutes les plateformes (Java, Javascript, PHP, Ruby, Python, …)
- Capable de s'aligner sur la structure des branches du répertoire
- Visualisation en temps réel de l'intégration

2 Intégrer Bitbucket-Pipeline au répertoire

Bitbucket propose de base un outil d'intégration/déploiement continu. Pour le configurer, il suffit simplement d'avoir un fichier « bitbucket-pipelines.yml » dans le répertoire. Le contenu de ce fichier sera expliqué au point 3. Pour accéder à cet outil, il suffit de cliquer sur l'onglet « Pipelines » du répertoire souhaité:

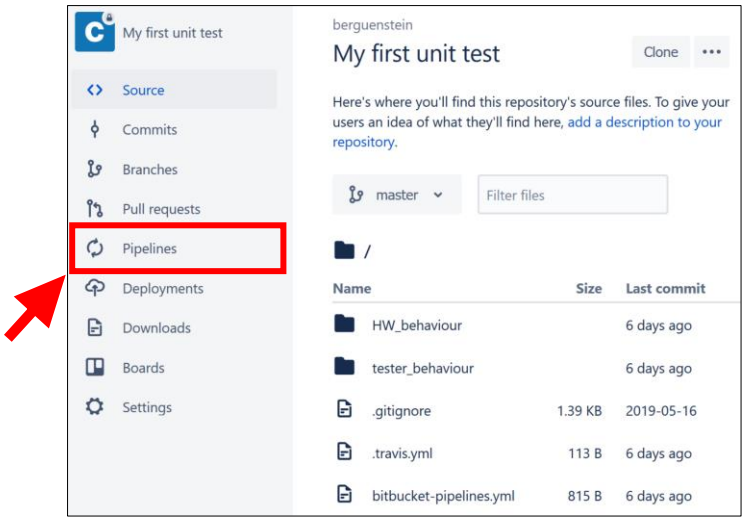


Figure 1: Situation de l'onglet « Pipelines »

2.1 Fichier « bitbucket-pipelines.yml » déjà intégré

Dans la figure ci-dessous, l'encadré en vert montre que le fichier est déjà intégré.

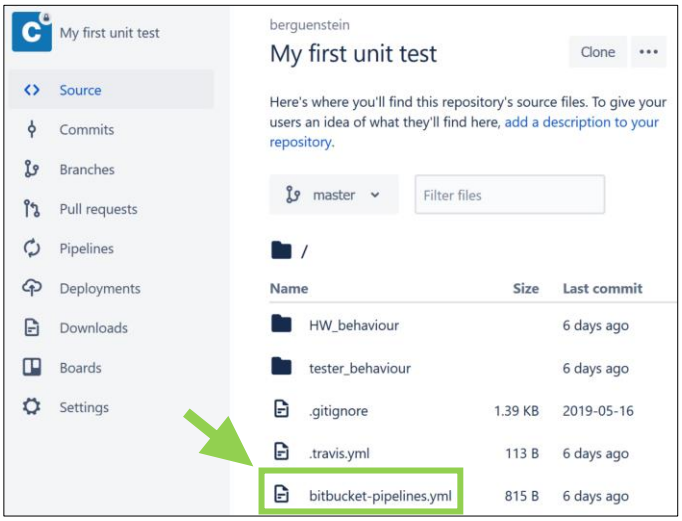


Figure 2: Aperçu du répertoire Bitbucket avec le fichier bitbucket-pipelines.yml déjà existant

2.2 Fichier « bitbucket-pipelines.yml » à créer

Si le répertoire ne contient pas encore ledit fichier, il est possible de le générer en allant sous l'onglet « Pipelines ». ci-dessous est affiché ce qui apparait alors sur la fenêtre. La partie qui est intéressante est encadrée en rouge :

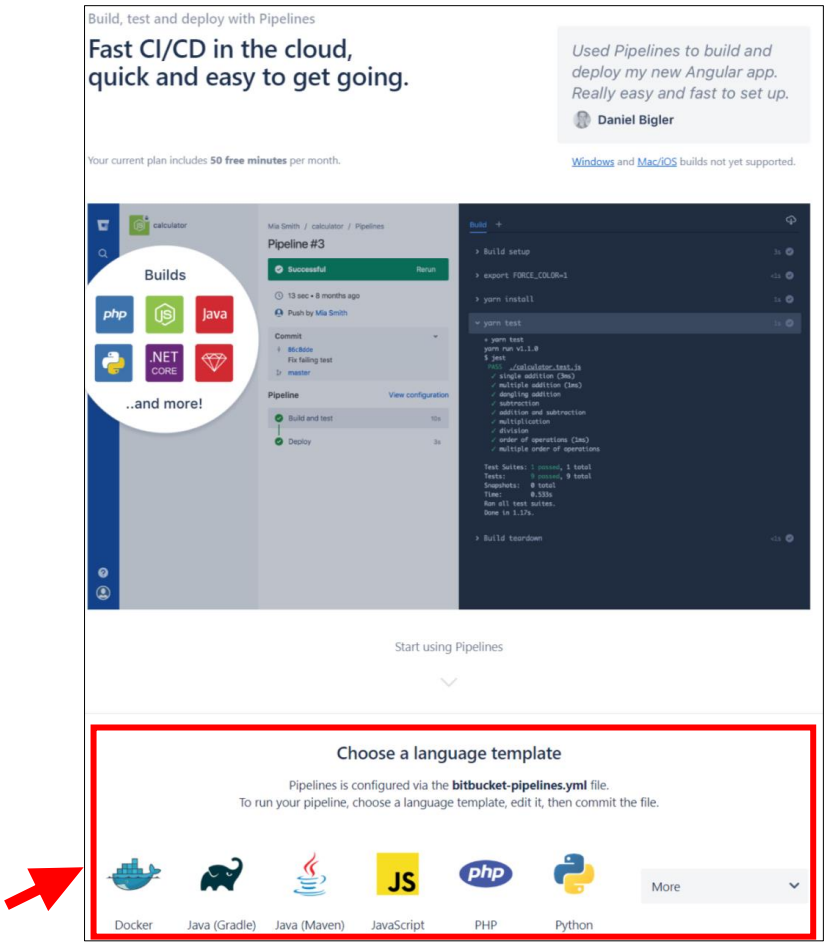


Figure 3: Ajouter l'outil CI/CD à un répertoire Bitbucket

Il faut alors choisir quel est le langage de programmation utilisé dans le répertoire. Pour cet exemple, le répertoire « my first unit test » est utilisé, et donc le langage C est choisi :

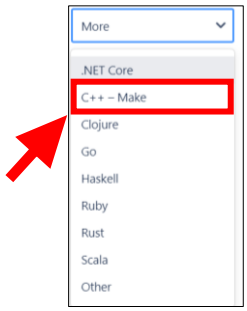


Figure 4: Menu déroulant pour plus de langages de programmation

Il s'en suit la fenêtre ci-dessous, qui est un fichier généré par défaut. Celui-ci sert uniquement à l'intégration continue. Si de plus amples options sont souhaitées, il est possible de les sélectionner dans l'encadré rouge, qui possède de multiples possibilités (quelques exemples sont affichés dans la Figure 6). Le contenu du fichier généré est expliqué plus en détail dans le point 3.

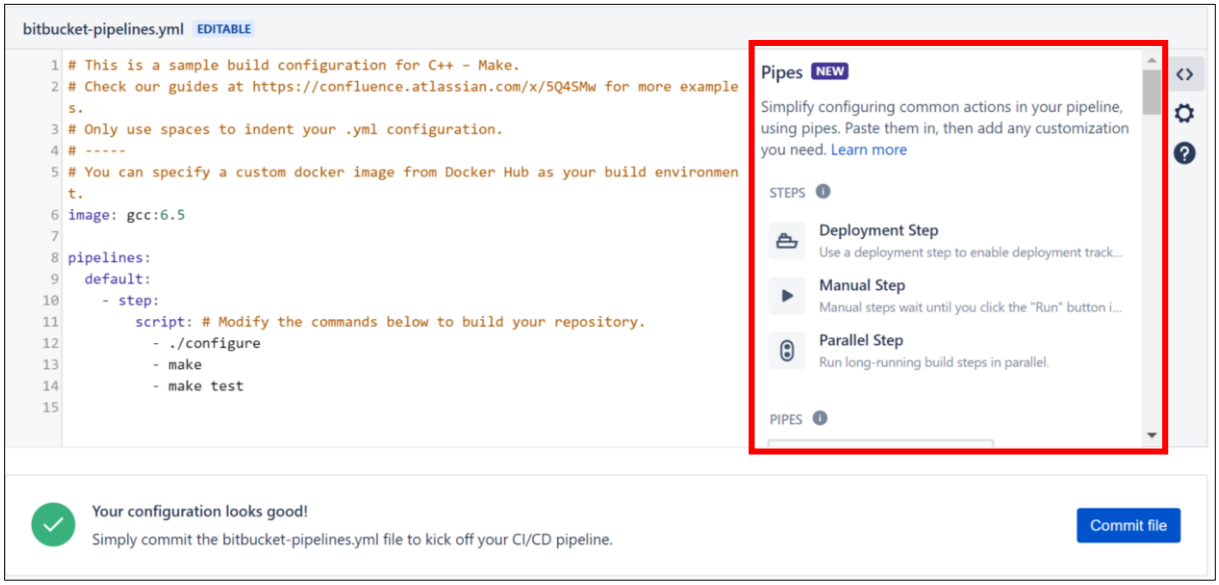


Figure 5: Aperçu du fichier généré par défaut

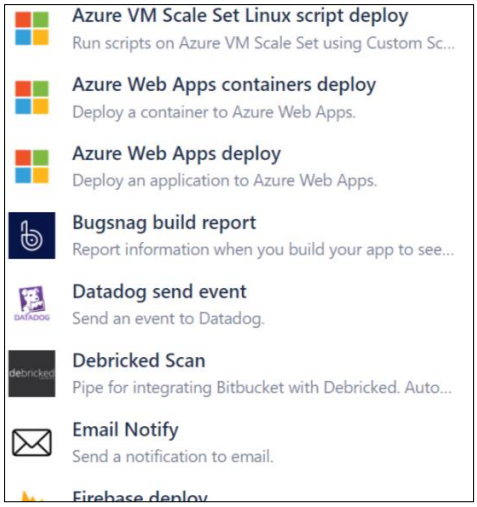


Figure 6: Quelques options pour Bitbucket CI/CD

2.3 Interface Bitbucket-Pipelines dans Bitbucket

Pour voir les diverses étapes du pipeline CI du répertoire, il faut aller dans la partie « Pipelines » de celui-ci :

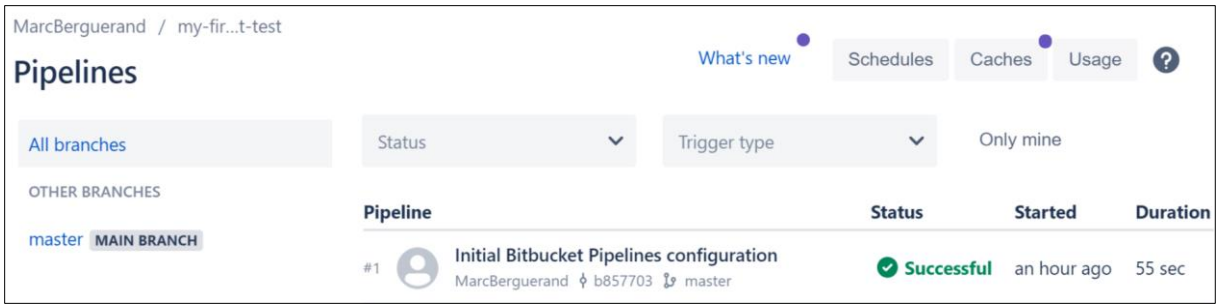


Figure 7: Aperçu de l'interface utilisateur du CI

Pour obtenir de plus amples informations sur l'exécution du CI, il suffit de cliquer sur le pipeline désiré. La fenêtre ci-dessous apparait alors. Si de plus amples données sont souhaitées, un clic sur l'étape (par exemple dans l'encadré en rouge) voulue change la console et il est alors possible de voir plus en détail l'exécution des scripts exécutés (encadré en vert). Ainsi apparait le résultat dans la Figure 9.

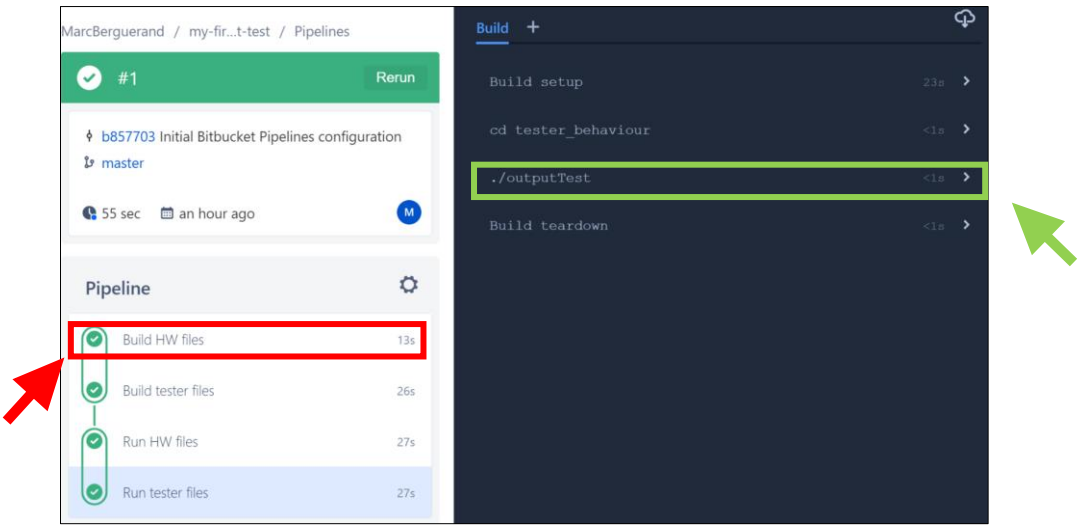


Figure 8: Aperçu du résultat du CI

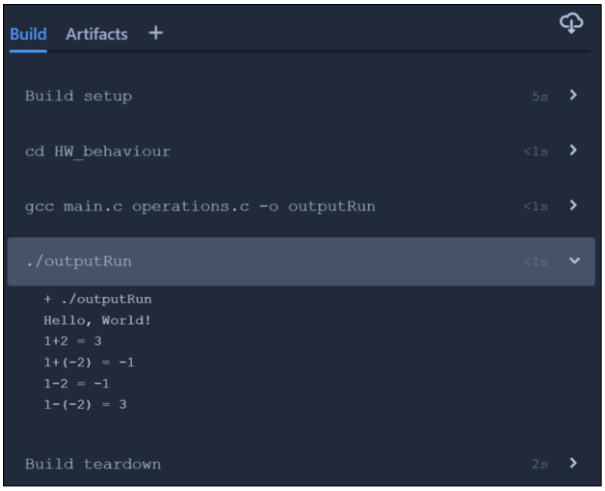


Figure 9: Aperçu détaillé d'un CI

3 Contenu du fichier « bitbucket-pipelines.yml »

Voici ci-dessous un exemple d'un fichier généré pour un template « C++ - Make », à partir du chapitre 2.2. À noter qu'il faut adapter ce fichier au répertoire désiré.

```
# This is a sample build configuration for C++ – Make.
# Check our guides at
#      https://confluence.atlassian.com/x/5Q4SMw for more
#      examples.
# Only use spaces to indent your .yml configuration.
# -----
# You can specify a custom docker image from Docker Hub as
your build environment.
image: gcc:6.5

pipelines:
  default:
    - step:
        script: # Modify the commands below
                #     to build your repository.
          - ./configure
          - make
          - make test
```

Figure 10: Exemple de la génération du fichier « bitbucket-pipelines.yml »

Voici de brèves explications sur le fichier généré :

- Image : désigne quelles sont les outils/environnements nécessaires pour déterminer les configurations des machines qui vont tester le répertoire.
- Labels « default » : Il s'agit d'un label qui définit comment le CI va effectuer les étapes.
- « step » : Il s'agit d'une étape réalisée par le CI.
- Script : désigne les commandes que la machine va effectuer durant l'étape ayant le label « step»

Pour avoir une meilleure compréhension du fonctionnement du système, l'exécution du CI pour le répertoire « my first unit test » est effectué et expliqué en détails ci-après.

3.1 Exemple du CI pour le répertoire « my first unit test »

Voici un exemple de configuration possible pour le répertoire « my first unit test », fait dans l'annexe 1 :

```yaml
image: gcc:6.5

pipelines:
  default:
    - parallel:
      - step:
          name: Build HW files
          script:
            - cd HW_behaviour
            - gcc main.c operations.c -o outputRun
            - ./outputRun
          artifacts:
            - HW_behaviour/outputRun
      - step:
          name: Build tester files
          script:
            - cd tester_behaviour
            - gcc main.c unity/unity.c test_operations.c
                  ../HW_behaviour/operations.c
                  -o outputTest
```

Figure 11: Exemple d'un fichier « bitbucket-pipelines. yml » pour le répertoire « my first unit test »

Pour mieux comprendre ce qu'interprète le CI, voici une représentation graphique du résultat :

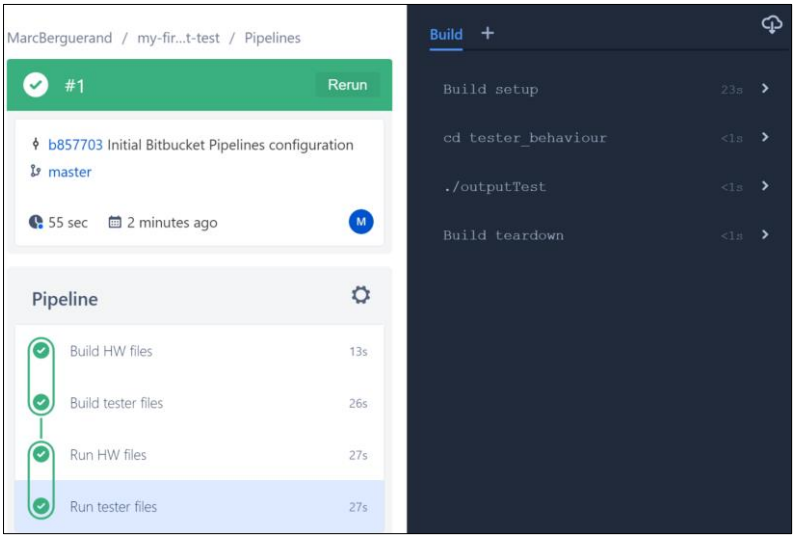


Figure 12: Représentation graphique du CI par Bitbucket

Ainsi, le « name » simplifie la lecture des tâches effectuées par le CI.

Les « artifacts » sont, dans ce cas, les fichiers compilés des différents projets (un des projets est l'exécution d'additions et de soustractions, l'autre est le testeur des méthodes d'additions et de soustractions). Pour que les étapes « Run » puissent exécuter les fichiers sans devoir se soucier de la compilation, une étape a été effectuée auparavant (l'ordre d'exécution est en fonction de la structure du fichier). Ceci permet également de mieux déterminer d'où proviendrait une éventuelle erreur, à la compilation ou à l'exécution.

Le script correspond simplement à ce que la machine va effectuer pour chaque étape/label. Les labels peuvent être lancés en parallèles (ce qui est le cas lorsqu'ils sont dans le même « parallel », comme le sont les labels « run tester files » et « run HW files »). Il est important de mentionner le fait que chaques « step » effectués dans un même « parallel » est effectué en même temps, mais le changement de « parallel » ne se fait que lorsque tous les « steps » auront été effectués en son sein. Ainsi, si une tâche prend beaucoup plus de temps que d'autres à être exécutée, il serait intéressant de faire d'abord s'exécuter toutes les autres puis finalement celle qui prend le plus de temps, afin de se rendre compte au plus vite s'il y a des erreurs dans les autres tâches. Voici une illustration pour mieux comprendre :

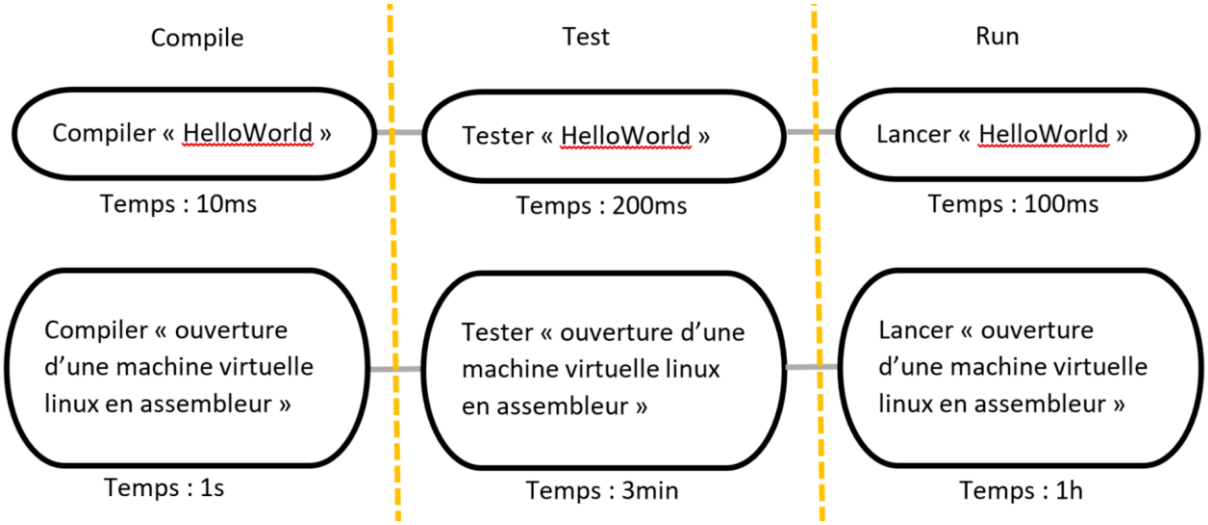


Figure 13: Exemple de temps nécessaire pour le CI

Dans cet exemple, le programme « Hello World » pourrait avoir fini son exécution en 310ms, mais comme il y a un autre programme en parallèle, il doit attendre que celui-ci s'exécute. Ainsi, « Hello World » finira son exécution en une heure, trois minutes et une seconde (Lorsqu'une étape est terminée, l'on peut directement voir le résultat, pas besoin d'attendre que les autres étapes soient terminées, c'est uniquement pour les changements d'états qu'il faut attendre).

Ainsi, dans l'exemple de la Figure 13, il serait plus judicieux de rajouter deux étapes, pour la compilation et le test du logiciel « Hello World », puis de commencer le parallélisme avec le lancement de « Hello World » et la compilation de l'autre programme.

3.2 Utiliser des artifacts

Les artifacts ont été mentionnés au chapitre 3.1. Pour mieux illustrer ceux-ci, la Figure 14 va être utile. En effet, il y a différentes étapes dans ce CI (Compile, Test et Run). Sans les artifacts, il faudrait recompiler le code à chaque étape. En revanche, en utilisant ceux-ci, la compilation ne se ferait uniquement à l'étape « Compile », puis le fichier résultant est défini comme artifact. Ainsi, il peut être utilisé tel quel dans les autre étapes (Test et Run).

Il y a également un autre avantage avec Bitbucket-Pipelines : Il est possible de télécharger ces artifacts une fois qu'ils ont été sauvegardés. Pour se faire, il faut aller sur l'étape qui a généré celui-ci, puis aller dans l'onglet « Artifacts » (encadré en rouge), puis le télécharger en cliquant où se trouve l'encadré vert.

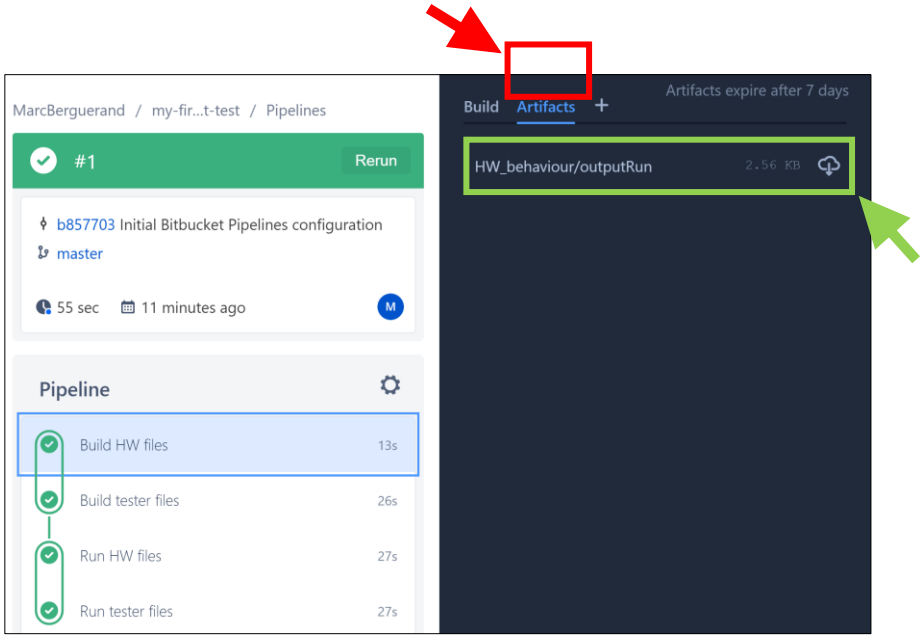


Figure 14: Situation de l'onglet « téléchargement des artifacts »

Ainsi, si un fichier est généré dans une étape (par exemple un fichier JSON contenant des informations sur le déroulement du code, ou un fichier excel où des données sont introduites durant l'exécution dudit code), il est possible de les télécharger et de regarder leurs contenus. Ceci permet d'avoir une vision supplémentaire sur les résultats liés à l'exécution du code. Ces artifacts sont supprimés après 7 jours.

4 Déploiement de Bitbucket-Pipelines

Il est important de comprendre ce qui se passe lors du CI. En effet, lorsque le CI est effectué, il est préférable de savoir où le code est effectué. Voici le résultat d'une étape du CI :

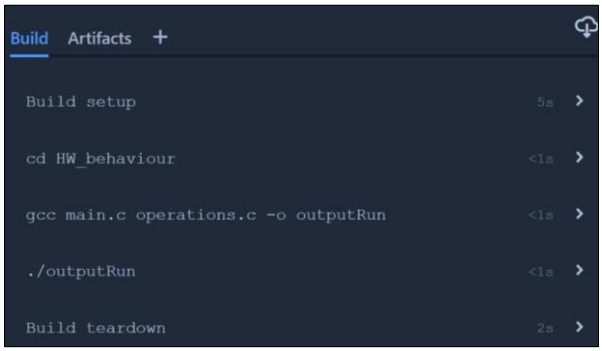


Figure 15: Aperçu de l'invite de commande virtuelle reprenant les résultats des tests

4.1 Machine utilisée pour le CI

Bitbucket ne permet pas de choisir une machine spécifique pour faire tourner le CI. La plateforme la définit elle-même.

4.2 Exécution du CI

Bitbucket-Pipelines classe les différentes étapes dans son interface graphique. Chaque étape est subdivisée par les commandes entrées dans le script et l'initialisation/destruction des machines CI. Les résultats sont affichés tels que dans une invite de commande. Dans cet exemple, il y a encore, à la suite de ceci, les artifacts à sauvegarder pour les étapes suivantes. Il s'en suit le résultat de l'opération, si tout est en ordre alors un « Succeed » apparait, sinon une erreur avec l'indication d'où elle provient, en fonction de la commande entrée, ou de l'exécution de ladite commande (voir ci-dessous).

Veuillez prendre note que les figures ci-dessous sont représentatives du framework de test « Unity ».

Afin d'échouer à un test, le fichier de tests a été modifié afin d'inclure une erreur à la soustraction, en disant que -5-5 = 0, ce qui est bien évidemment faux :

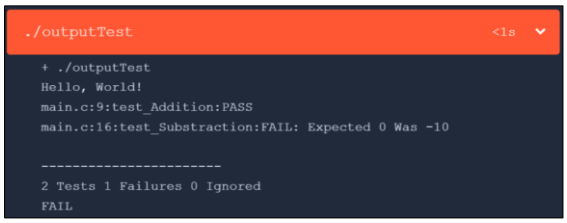


Figure 16: Tests échoués

Et voici le résultat lorsque les tests sont réussis :

Figure 17: Tests réussi

Annexe n°6

Utilisation de Travis CI

Table des matières

Liste des figures

Préambule

Toutes les images apparaissant dans cette annexe proviennent de la plateforme Travis CI. Cette annexe a été réalisée au mois de mai 2019, veuillez prendre en compte que des modifications aient pu avoir lieu au moment où vous lisez ceci.

1 Caractéristique de Travis

Voici diverses informations propres à Travis CI :

- Se lie à Github
- Test les pull requests
- Déploiement facile
- Bases de données et services préinstallés
- Configuration rapide
- Visualisation en temps réel de l'intégration
- Auto-déploiement possible
- Support pour Mac, Linux et iOS
- Supporte énormément de langages (PHP, Ruby, Python, Java, Scala, …)

2 Intégrer Travis au répertoire

Travis est un outil en soit, qui n'est pas intégré à une plateforme d'hébergement de base. Toutefois, une connexion simplifiée avec Github est mise en place. Ainsi, il est possible de s'identifier à l'aide de celle-ci. C'est d'ailleurs avec ceci que cette annexe est conçue.

2.1 Connexion via Github

Afin de pouvoir utiliser Travis avec Github, il suffit de choisir l'option « Sign in with GitHub » (voir ci-dessous) :

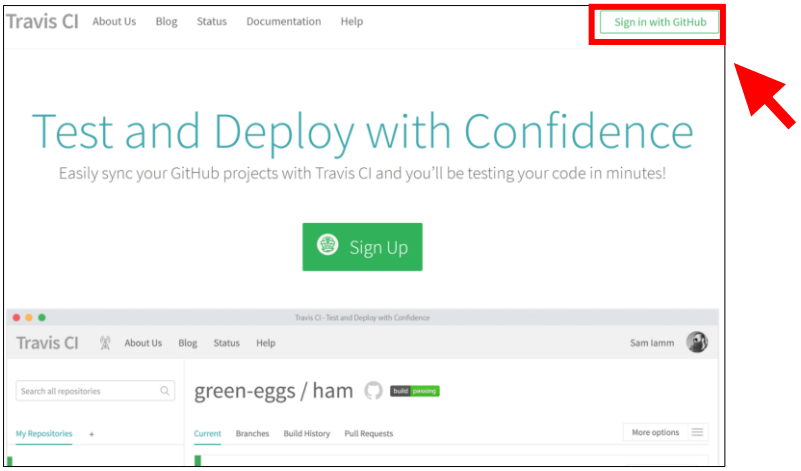


Figure 1: Aperçu de la fenêtre principale de Travis

Ainsi, il faut entrer les identifiants de Github :

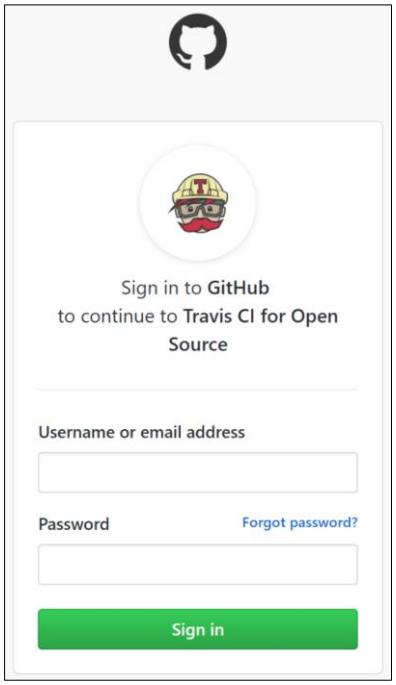


Figure 2: Identification de Travis vis Github

Puis il faut autoriser l'utilisation de Travis pour Github :

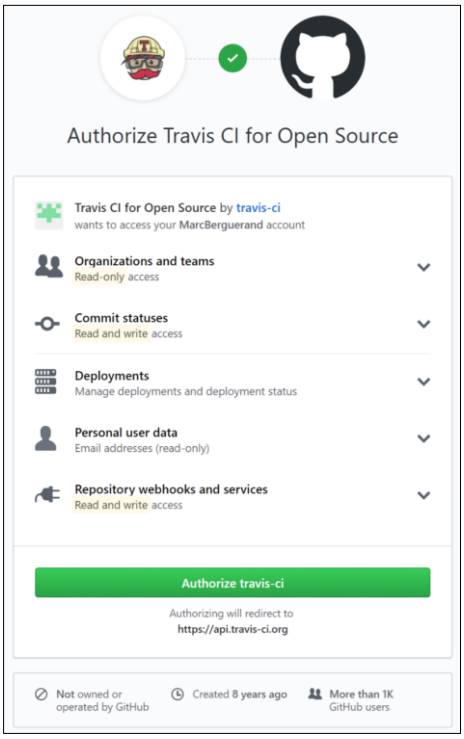


Figure 3: Autorisation de Travis sur Github

Ainsi, les répertoire Github apparaissent sur la plateforme de Travis :

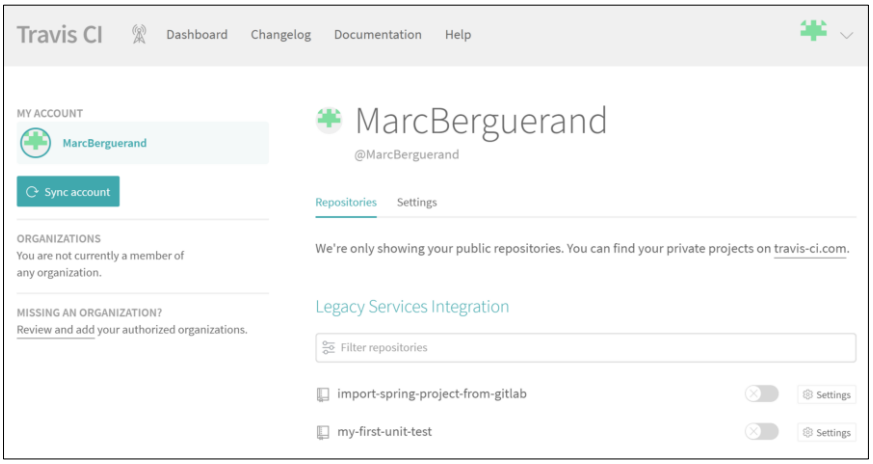


Figure 4: Aperçu des répertoires Github dans Travis

En sélectionnant un répertoire à tester, la fenêtre suivante apparait. Il faut alors activer Travis pour ledit répertoire.

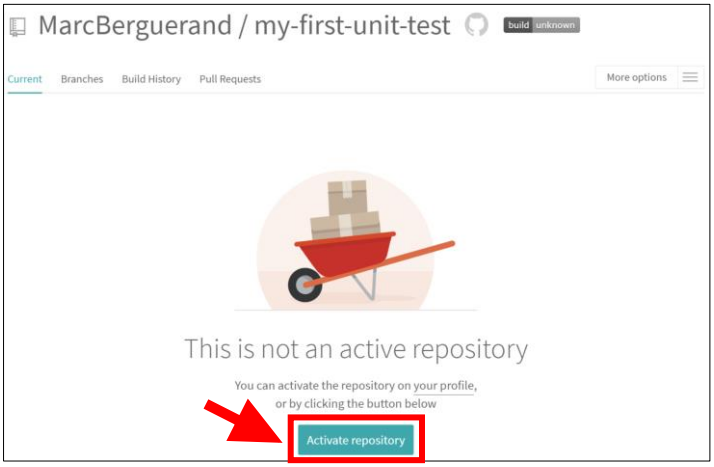


Figure 5: Activer Travis pour un répertoire

Ainsi, le répertoire est activé sur Travis. Il faut maintenant ajouter le fichier permettant à Travis d'effectuer le CI.

2.2 Fichier «.travis.yml » à créer

Travis ne fournit pas de modèle de base pour générer un fichier. Il faut ainsi le concevoir soi-même. Pour de plus amples informations sur le contenu de celui-ci, voir le point 3.

2.3 Interface Travis

Par défaut, à chaque fois que le répertoire est mis à jour sur Github, Travis vérifie s'il y a un fichier «.travis.yml ». Si c'est le cas, alors il va se lancer et effectuer le CI. Voici un exemple lorsque Travis est enclenché :

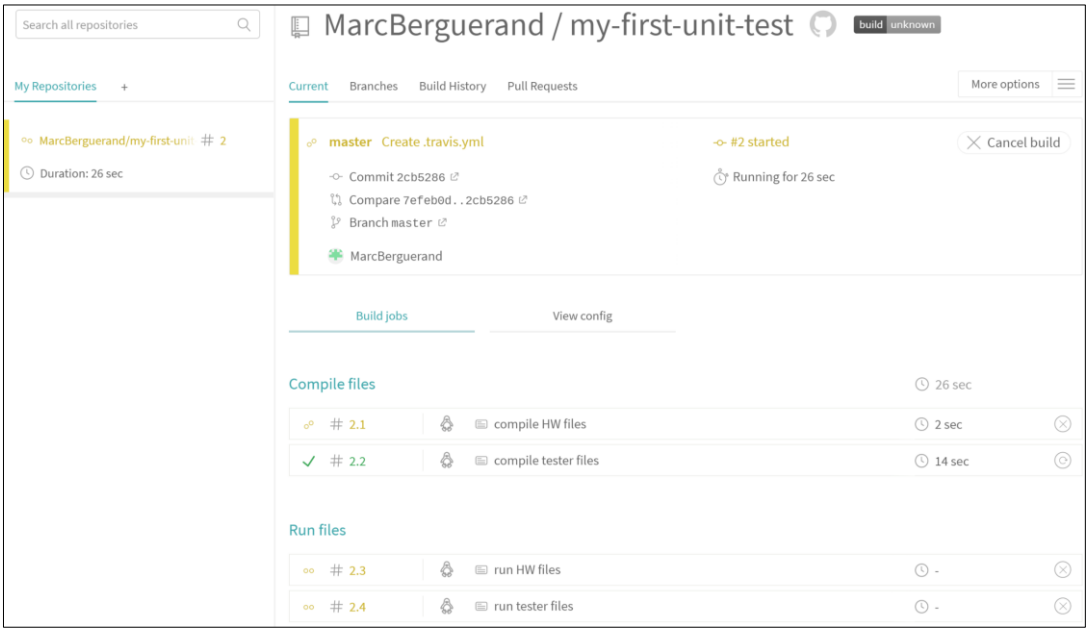


Figure 6: Enclenchement du CI de Travis

Lorsque plusieurs sessions de tests ont eu lieu pour le même répertoire, l'historique de ceux-ci peut être visible dans l'onglet « Build History » :

Figure 7: Aperçu de l'historique CI

En cliquant sur le CI souhaité, les informations détaillées apparaissent :

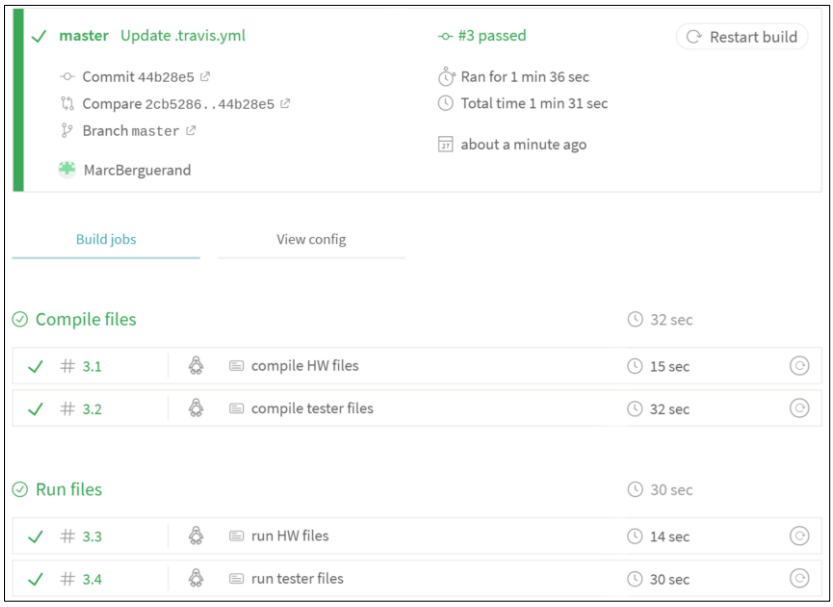


Figure 8: Aperçu plus en détail du CI

Pour apercevoir les résultats propres à chaque étape, il suffit de cliquer sur celle souhaitée :

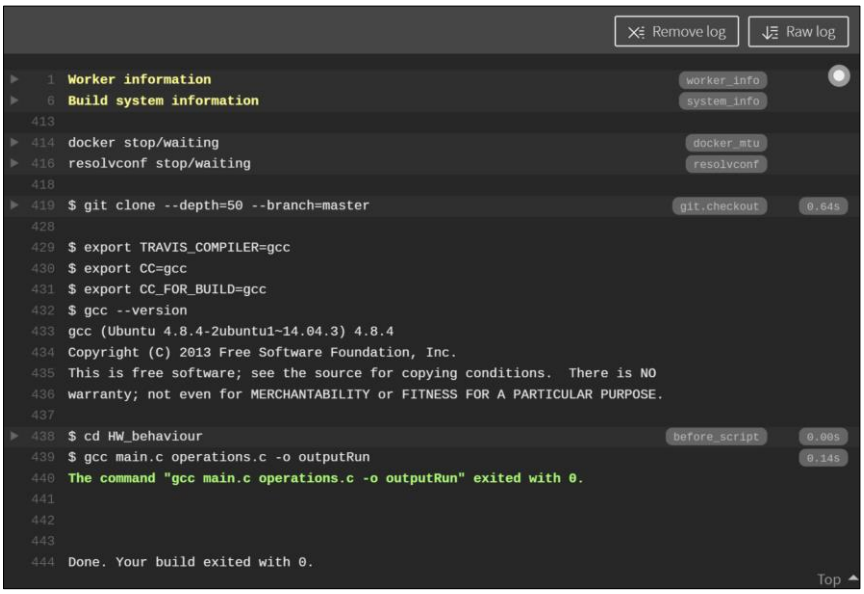


Figure 9: Informations détaillées d'une étape

3 Contenu du fichier « .travis.yml »

Voici ci-dessousFigure 10 un exemple du dit fichier, réalisé pour le répertoire « my first unit test » :

```yaml
language: c

stages:
  -name: compile files
  -name: run files

jobs:
  include:
    - stage: compile files
      name: "compile HW files"
      before_script:
        - cd HW_behaviour
      script:
        - gcc main.c operations.c -o outputRun
    - stage: compile files
      name: "compile tester files"
      before_script:
        - cd tester_behaviour
      script:
        - gcc main.c unity/unity.c test_operations.c
                  ../HW_behaviour/operations.c -o outputTest

    - stage: run files
      name: "run HW files"
      before_script:
        - cd HW_behaviour
      script:
        - gcc main.c operations.c -o outputRun
        - ./outputRun
    - stage: run files
      name: "run tester files"
      before_script:
        - cd tester_behaviour
      script:
        - gcc main.c unity/unity.c test_operations.c
                  ../HW_behaviour/operations.c -o outputTest
        - ./outputTest
```

Figure 10: Exemple d'un fichier « .travis.yml »

Voici de brèves explications sur le fichier généré :

Au tout début, le langage est spécifié, ceci déterminera la machine qui va devoir effectuer le CI.

« Stages » définit l'ordre d'exécution des étapes. Il s'en suit « jobs » et « include » qui permettent de déterminer plusieurs étapes se déroulant en parallèles, en fonction du « stage » définit pour chaque étape.

« Stage » détermine à quel moment est réalisé la tâche. « name » permet de mieux visualiser quelle est l'étape réalisée dans l'interface utilisateur.

« before_script » réalise les commandes à faire pour initialiser la machine afin d'un bon déroulement du CI. Après cette partie a lieu le « script », qui est le test en soi.

Il est important de mentionner qu'il n'est pas possible de sauvegarder des fichiers entre deux « stage » de base. C'est pourquoi lors des étapes « Run », la compilation des fichiers est refaite. Pour profiter d'une sauvegarde il faut lier Travis à un compte Amazone S3 par exemple, et donc créer un compte supplémentaire.

Ainsi, les « names » simplifient la lecture des tâches effectuées par le CI. Les « stages » déterminent à quel moment est effectué chaque « travaux ». Il est possible de remplir par n'importe quoi les « names » et « stages ».

Le script correspond simplement à ce que la machine va effectuer pour chaque étape/label. Les labels peuvent être lancés en parallèles (ce qui est le cas lorsqu'ils sont définis dans le même « stage », comme le sont les labels « run HW files » et « run tester files »). Il est important de mentionner le fait que chaques « travaux » effectués dans un même « stage » est effectué en même temps, mais le changement de « stage » ne se fait que lorsque tous les « labels » auront été effectués en son sein. Ainsi, si une tâche prend beaucoup plus de temps que d'autres à être exécutée, il serait intéressant de faire d'abord s'exécuter toutes les autres puis finalement celle qui prend le plus de temps, afin de se rendre compte au plus vite s'il y a des erreurs dans les autres tâches. Voici une illustration pour mieux comprendre :

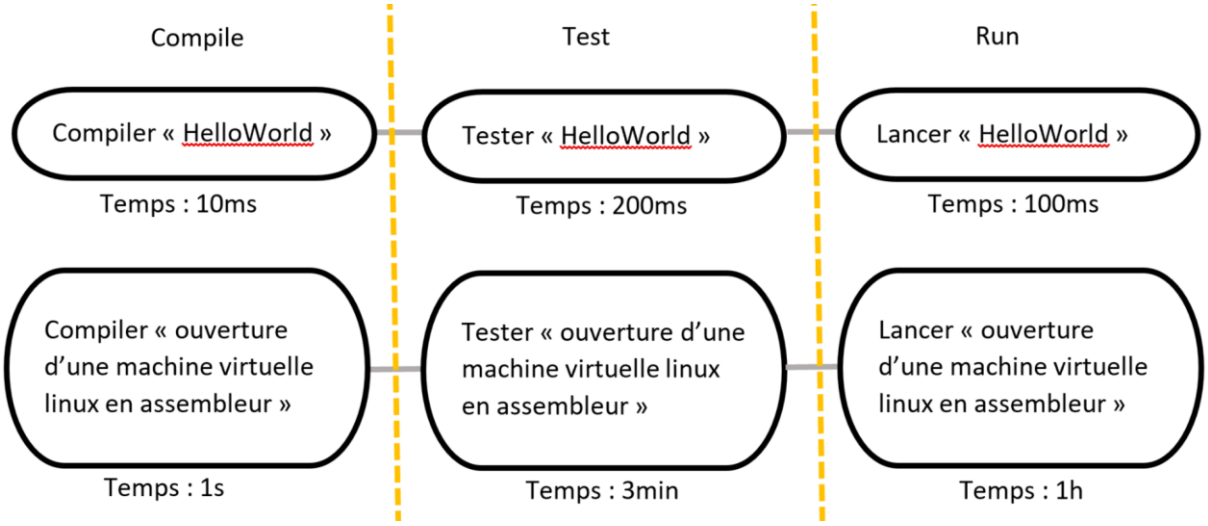


Figure 11: Exemple de temps nécessaire pour le CI

Dans cet exemple, le programme « Hello World » pourrait avoir fini son exécution en 310ms, mais comme il y a un autre programme en parallèle, il doit attendre que celui-ci s'exécute. Ainsi, « Hello World » finira son exécution en une heure, trois minutes et une seconde (Lorsqu'une étape est terminée, l'on peut directement voir le résultat, pas besoin d'attendre que les autres étapes soient terminées, c'est uniquement pour les changements d'états qu'il faut attendre).

Ainsi, dans l'exemple de la Figure 11, il serait plus judicieux de rajouter deux étapes, pour la compilation et le test du logiciel « Hello World », puis de commencer le parallélisme avec le lancement de « Hello World » et la compilation de l'autre programme.

4 Déploiement de Travis

Il est important de comprendre ce qui se passe lors du CI. Travis ne permet pas de choisir une machine spécifique pour faire tourner le CI. La plateforme la définit elle-même.

4.1 Exécution du CI

Travis classe les différentes étapes dans son interface graphique. Chaque étape est subdivisée par les commandes entrées dans le script et l'initialisation/destruction des machines CI. Les résultats sont affichés tels que dans une invite de commande. Il s'en suit le résultat de l'opération, si tout est en ordre alors un « Succeed » apparait, sinon une erreur avec l'indication d'où elle provient, en fonction de la commande entrée, ou de l'exécution de ladite commande (voir ci-dessous).

Veuillez prendre note que les figures ci-dessous sont représentatives du framework de test « Unity ».

Afin d'échouer à un test, le fichier de tests a été modifié afin d'inclure une erreur à la soustraction, en disant que -5-5 = 0, ce qui est bien évidemment faux :

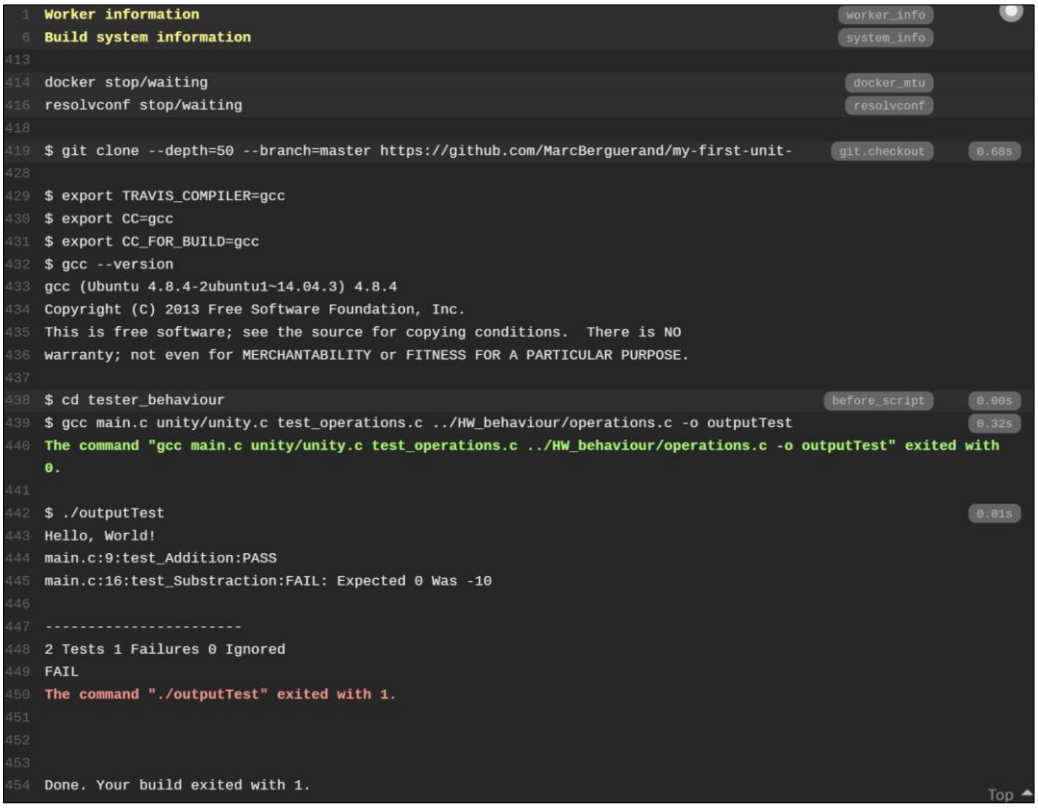


Figure 12: Tests échoués

Et voici le résultat lorsque les tests sont réussis :

```
Worker information                                                          worker_info
Build system information                                                    system_info

docker stop/waiting                                                         docker_mtu
resolvconf stop/waiting                                                     resolvconf

$ git clone --depth=50 --branch=master https://github.com/MarcBerguerand/my-first-unit-   git.checkout   0.57s

$ export TRAVIS_COMPILER=gcc
$ export CC=gcc
$ export CC_FOR_BUILD=gcc
$ gcc --version
gcc (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4
Copyright (C) 2013 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

$ cd tester_behaviour                                                       before_script  0.00s
$ gcc main.c unity/unity.c test_operations.c ../HW_behaviour/operations.c -o outputTest   0.24s
The command "gcc main.c unity/unity.c test_operations.c ../HW_behaviour/operations.c -o outputTest" exited with
0.

$ ./outputTest                                                                            0.00s
Hello, World!
main.c:9:test_Addition:PASS
main.c:10:test_Substraction:PASS

-----------------------
2 Tests 0 Failures 0 Ignored
OK
The command "./outputTest" exited with 0.



Done. Your build exited with 0.
```

Figure 13: Tests réussis

Annexe n°7

Utilisation de Docker

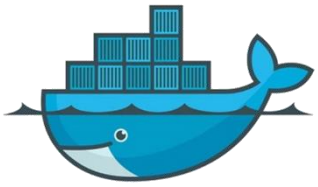


Table des matières

Liste des figures

Préambule

Toutes les images apparaissant dans cette annexe proviennent de la plateforme Docker Hub. Cette annexe a été réalisée au mois de juin 2019, veuillez prendre en compte que des modifications aient pu avoir lieu au moment où vous lisez ceci.

1 Identification

Pour s'identifier, il suffit de cliquer sur « Sign In » (en vert dans la figure ci-dessous). Au cas où il faudrait créer un compte, cliquer sur « Sign up for Docker Hub » (en rouge dans la figure ci-dessous).

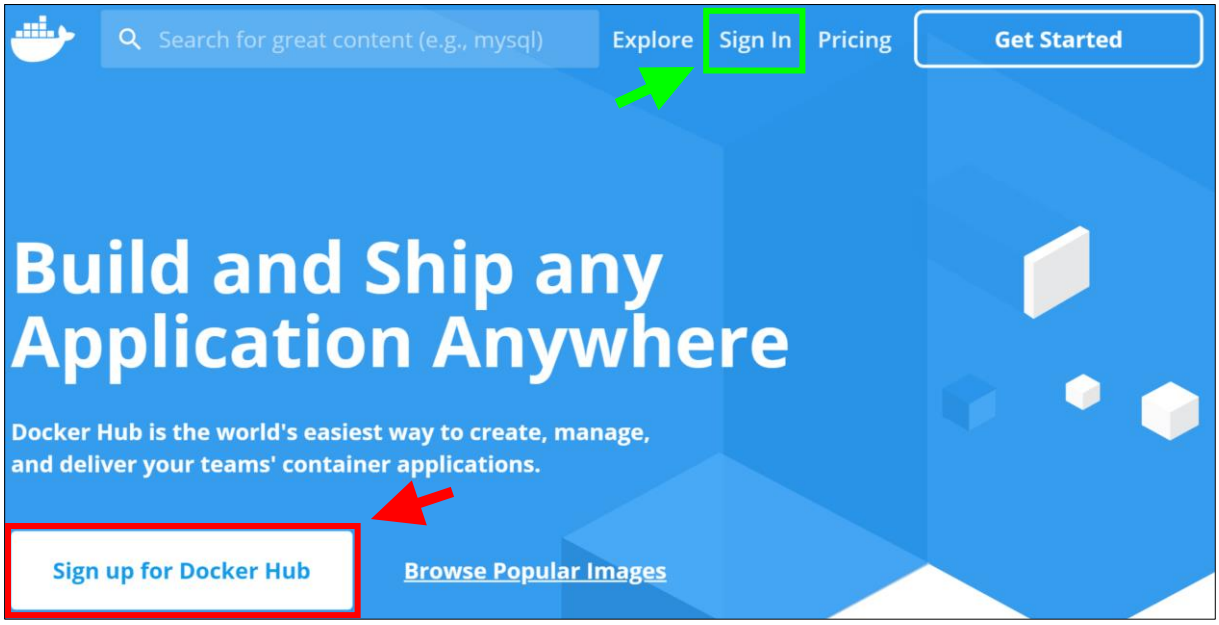


Figure 1: Fenêtre d'accueil de Docker Hub

1.1 Création d'un compte

Dans le cas où il faut créer un compte, alors la page suivante apparait :

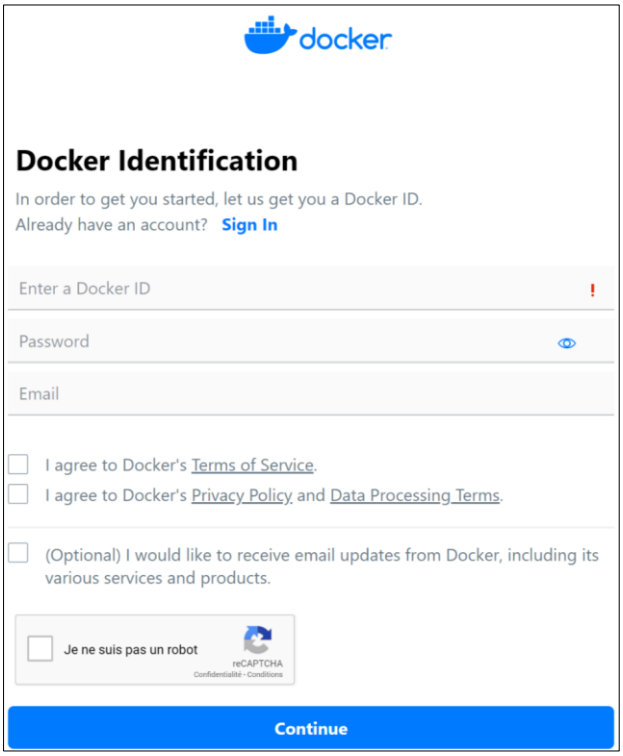


Figure 2: Fenêtre d'enregistrement d'un compte

Ainsi, il faut entrer un identifiant, un mot de passe et une adresse mail.

Une fois cette étape effectuée, il est demandé de plus amples informations personnelles :

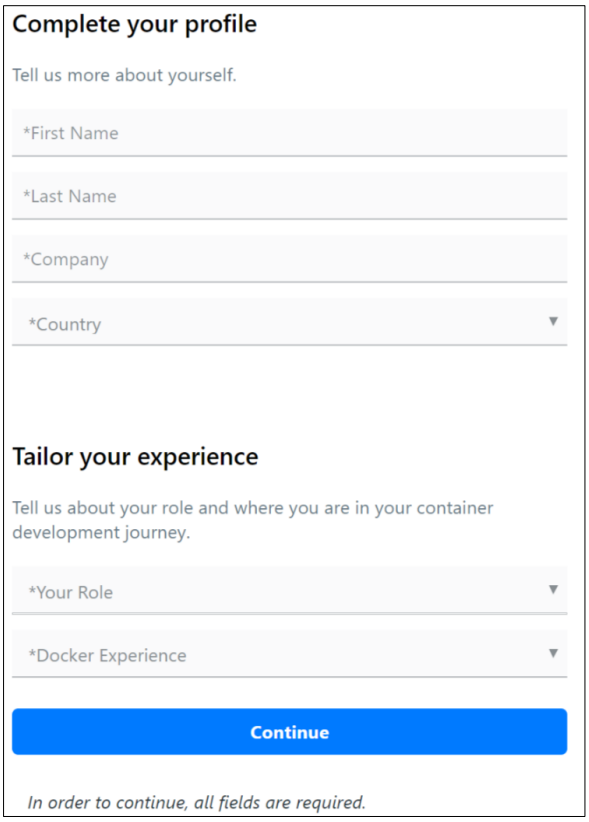


Figure 3: Enregistrement d'un compte - de plus amples informations

Ensuite, une invitation à visualiser ses mails est présentée, afin de valider le compte :

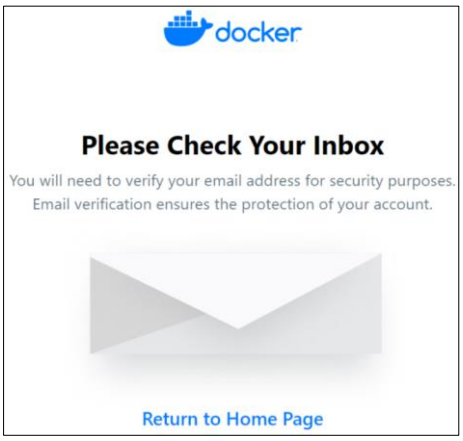


Figure 4: Enregistrement d'un compte - vérification de mails

1.2 Se connecter

Pour se connecter à son compte, il a fallu cliquer sur « Sign in »dans la Figure 1. Il s'en suit cette fenêtre :

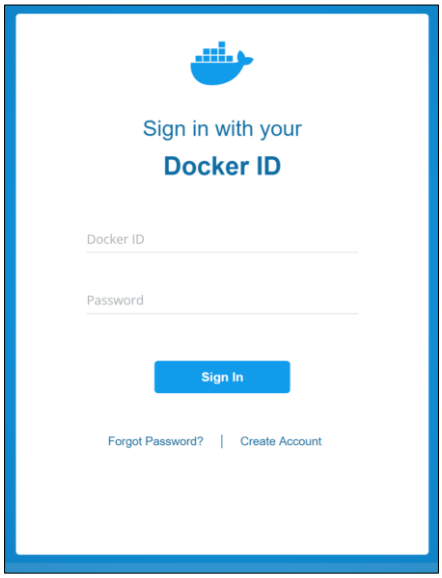


Figure 5: S'identifier avec son compte Docker Hub

Ensuite, la fenêtre d'exploration principale de Docker Hub apparait :

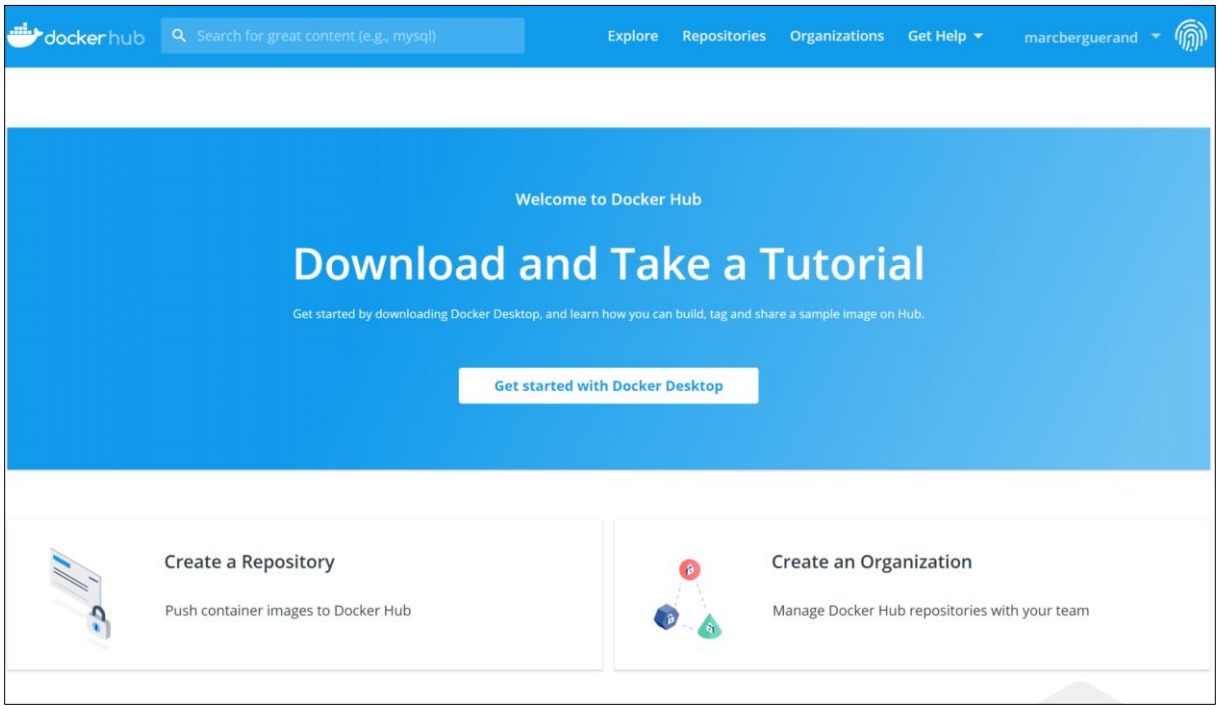


Figure 6: Fenêtre principale utilisateur de Docker Hub

2 Déconnexion

Pour se déconnecter, il suffit de cliquer sur l'onglet utilisateur (en vert dans la figure ci-dessous) puis sur « Log out » (en rouge dans la figure ci-dessous).

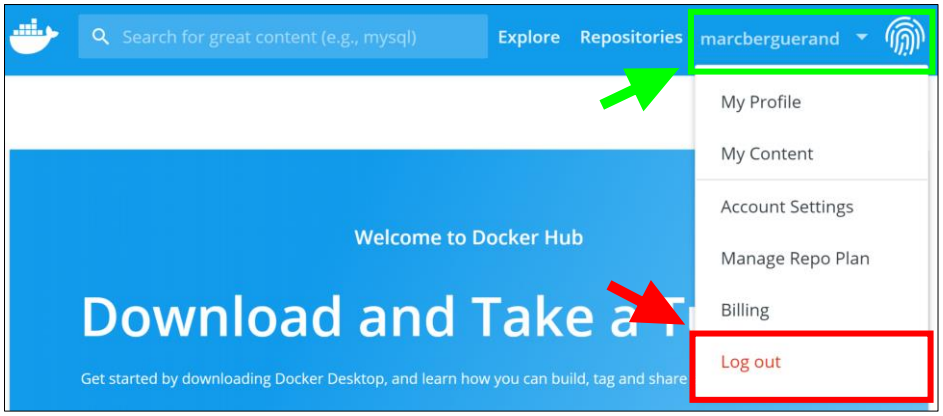


Figure 7: Se déconnecter de Docker Hub

3 Rechercher des images

Pour effectuer une recherche d'images déjà existantes dans la base de données de Docker Hub, il suffit d'entrer le nom des spécificités que l'on souhaite dans la barre de recherche (en rouge dans la figure ci-dessous).

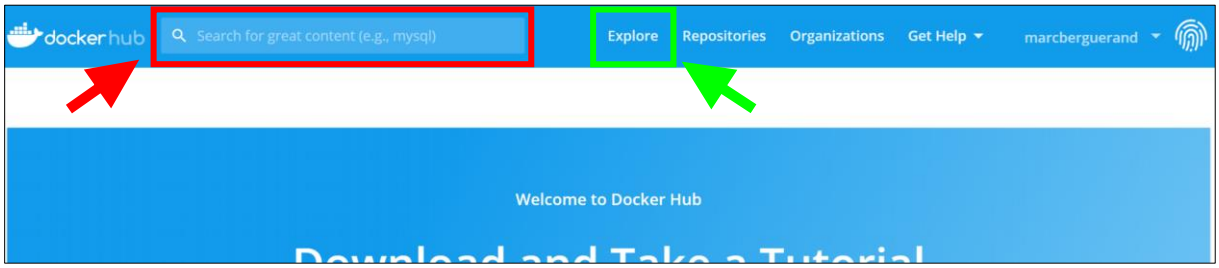


Figure 8: Rechercher des images sur Docker Hub

Ainsi, si la recherche porte sur une image ayant un lien avec gcc, voici le résultat :

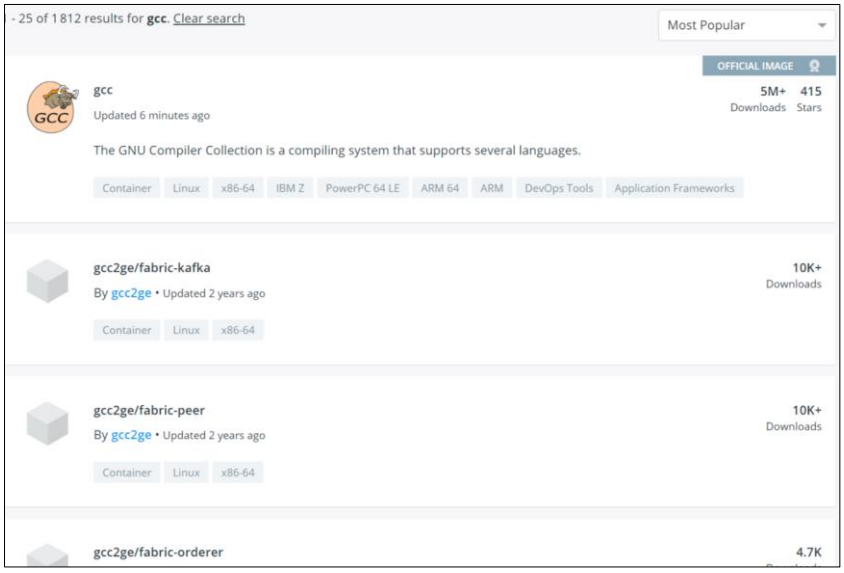


Figure 9: Aperçu des images ayant un lien avec gcc

À noter que le lien avec gcc n'est pas que l'image contienne gcc, mais que le nom contienne gcc.

S'il n'y a qu'une simple envie de voir le contenu disponible, il est alors possible de cliquer sur « Explore » (en vert dans la figure ci-dessous), ce qui affiche toutes les images de Docker Hub :

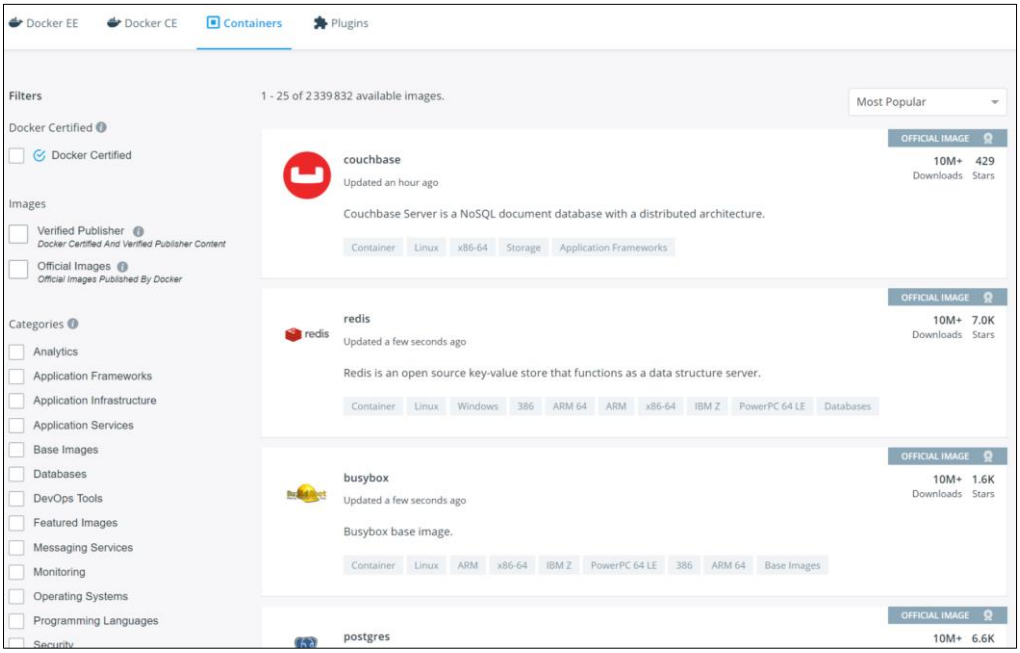


Figure 10: Aperçu de toutes les images de Docker Hub

4 Gestion de ses propres images

Pour apercevoir les images de l'utilisateur, il faut cliquer sur l'onglet « Repositories » :

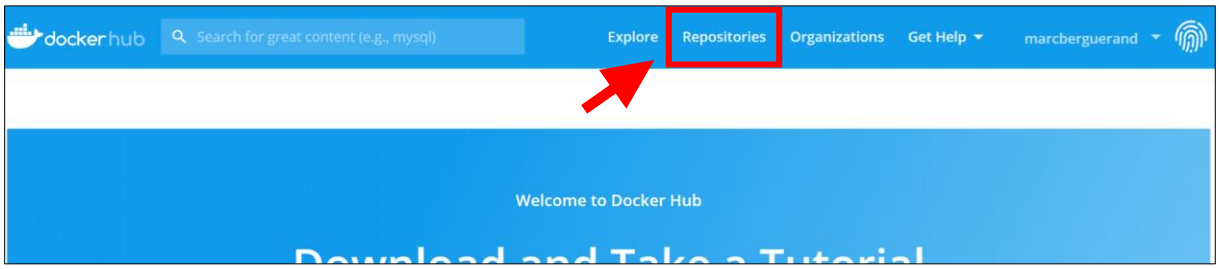


Figure 11: Situation de l'onglet "Repositories"

Ainsi, cette fenêtre apparait (dans ce cas-ci, aucune image n'est encore associée au compte). :

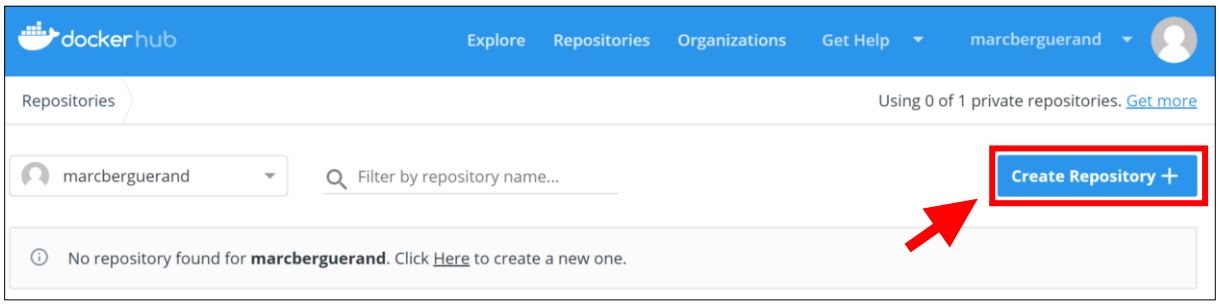


Figure 12: Aperçu des images de l'utilisateur

Depuis cette interface, il est possible de créer un répertoire qui pourra alors contenir une image. Pour se faire, cliquer sur « Create Repository+ » (en rouge dans la figure ci-dessus).

Ainsi, la fenêtre suivante apparait :

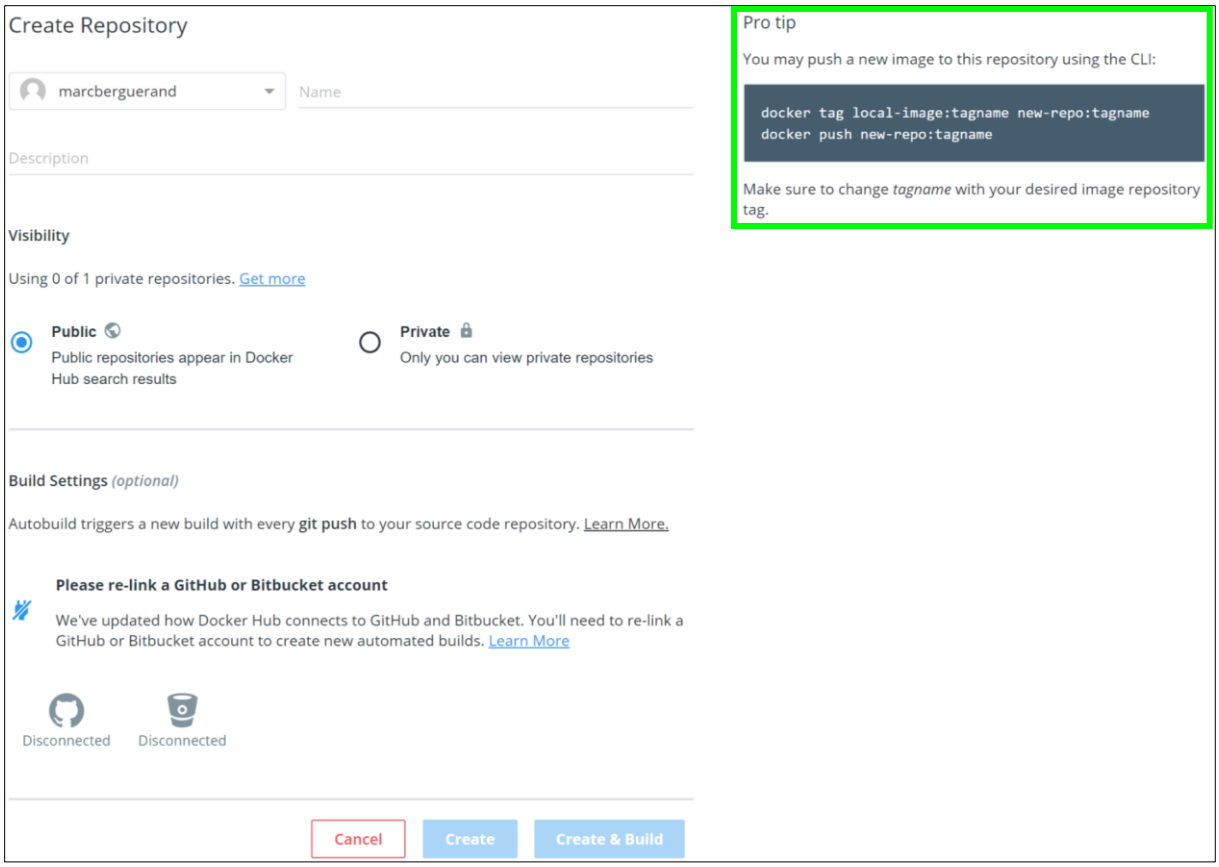


Figure 13: Création d'un nouveau répertoire Docker Hub

Cette interface permet de concevoir un répertoire de manière soit publique (n'importe quel utilisateur de Docker pourra alors utiliser l'image générée), soit privée (attention cependant, car le compte gratuit n'offre qu'un seul répertoire privé possible). Il est également possible de lier le répertoire à Github ou à Bitbucket, afin d'appeler l'image Docker à chaque fois que le répertoire est mis à jour.

Sur cette fenêtre, la partie de droite (encadré en vert sur la figure ci-dessus) indique comment téléverser une image depuis un ordinateur via l'invite de commande. Ceci fait donc qu'il y a deux manières de créer un répertoire avec Docker Hub.

Ainsi, une fois le répertoire généré, l'aperçu des images de l'utilisateur a été modifié. Donc, en cliquant sur l'onglet « Repositories », voici ce qui est affiché :

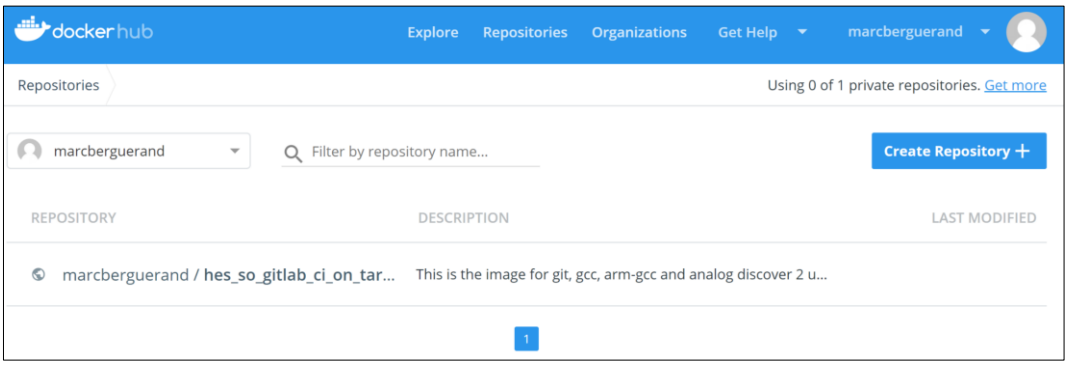


Figure 14: Aperçu des images de l'utilisateur

5 Configuration d'un répertoire depuis l'interface Docker Hub

Lorsque le répertoire est créé, l'on peut aller sur celui-ci, en cliquant simplement sur son nom. Ainsi, la fenêtre suivante apparait :

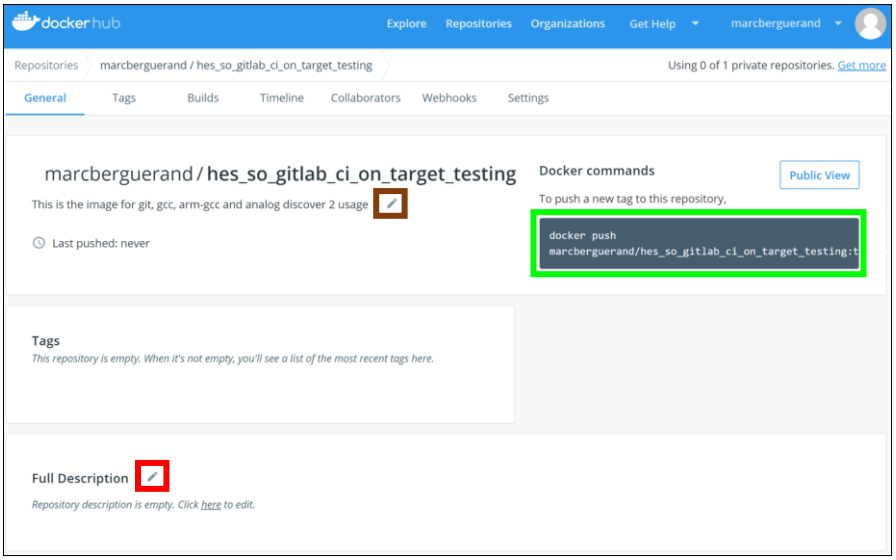


Figure 15: Aperçu d'un répertoire nouvellement créé

Ainsi, l'encadré en vert dans la figure ci-dessus est la suite de commandes à entrer dans un terminal afin de lier une image à ce répertoire.

Il est possible de modifier le résumé du répertoire en cliquant dans l'encadré brun, et l'on peut entrer une description pour le répertoire en cliquant sur l'encadré rouge. Ainsi, une fois les descriptions et résumés modifiés, le résultat est, par exemple, le suivant :

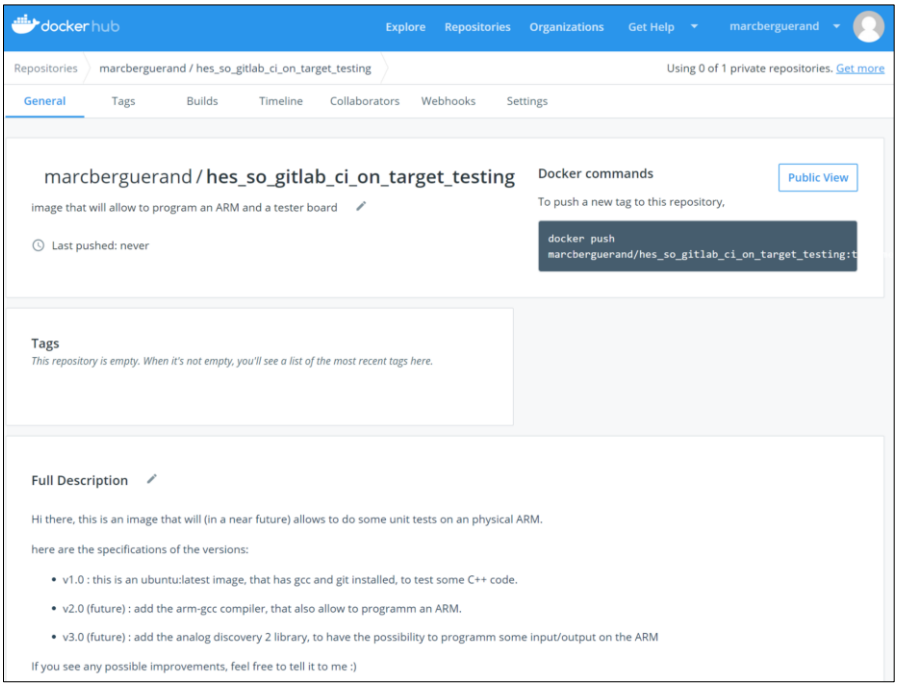


Figure 16: Aperçu d'un répertoire avec résumé et description

Il est également possible de changer de vue, afin de s'apercevoir ce que voient les autres utilisateurs, qui ne sont pas responsables du répertoire. Pour se faire, il faut cliquer sur « Public View » dans la Figure 16. Ainsi, voici le résultat :

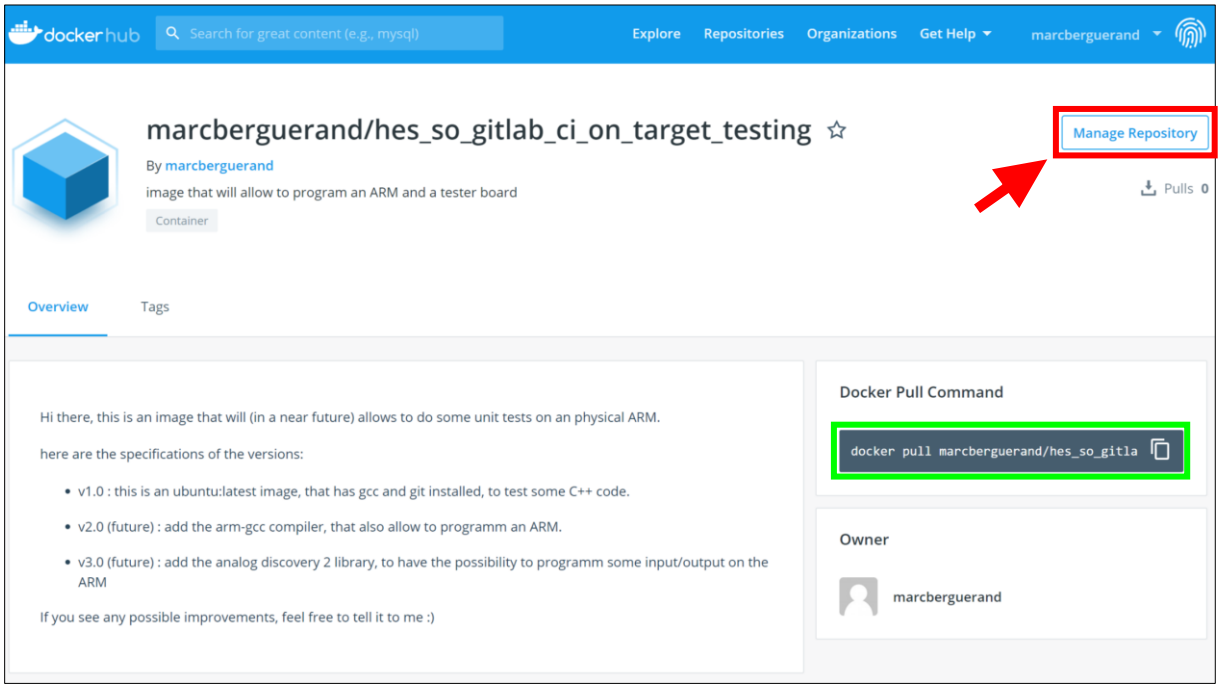


Figure 17: Aperçu du répertoire vu par la communauté

Pour retourner sur l'interface permettant de modifier le répertoire, il suffit de cliquer sur « Manage Repository », en rouge dans la figure ci-dessus.

6 Récupérer une image depuis un répertoire

Afin de pouvoir utiliser une image hébergée sur Docker Hub, il faut utiliser la commande encadrée en verte dans l'image ci-dessus. Ceci va télécharger l'image. Il faut ensuite l'exécuter pour que l'environnement soit utilisable. Pour exécuter une image, voici une commande type :

```
docker run image_exemple:v1.0
```

Figure 18: Commande pour lancer une image docker

Il est important de mentionner que dans cette thèse, ces commandes n'ont pas besoin d'être exécutées, car Gitlab CI les fait nativement en créant le Runner.

7 Créer une image Docker

Plusieurs possibilités existent pour concevoir une image Docker. En voici quelques-unes :

- À partir d'un fichier Dockerfile
- À partir d'un Container déjà existant, en ajoutant des éléments en son sein et en enregistrant l'image actuelle sous un nouveau Tag.

7.1 Dockerfile

Dans le cadre de ce travail, la création à partir d'un Dockerfile est choisie. De telle manière, il est également possible d'utiliser cette image depuis le répertoire Git. En effet, d'autres outils d'intégration continue ne propose pas nativement de lier une machine directement en tant que Container Docker. Ainsi, il sera possible, si l'outil CI choisi est différent de Gitlab, de pouvoir utiliser Docker sur celle-ci, en utilisant des lignes de commandes au sein du fichier du CI.

Voici donc un exemple de Dockerfile très simple :

```
# Download base image ubuntu latest version
FROM ubuntu:latest

# Define the maintainer of this image
MAINTAINER marc

# Update the software repository
RUN apt-get update

# Install git and gcc, to test a C++ code
RUN apt-get install -y git gcc
```

Figure 19: Exemple d'un Dockerfile

Ainsi, les lignes commençant par le caractère « # » sont des commentaires.

Le premier mot d'une ligne correspond à ce que l'image doit faire. Voici une explication pour les trois mots-clés utilisés ci-dessus :

- FROM : détermine depuis quelle image celle-ci se base. Il est possible de partir de « Scratch » et ainsi construire soi-même son propre environnement entièrement personnalisé.
- MAINTAINER : définis le responsable de l'image (ceci est optionnel)
- RUN : ce qui est écrit à la suite de ce mot est exécuté au sein de l'image, en tant que super utilisateur.

Donc, le Dockerfile de la Figure 19 va être une image basée sur la dernière version d'Ubuntu définie sur Docker Hub. Elle aura comme supplément la possibilité d'utiliser Git et GCC. Ainsi, elle peut télécharger des répertoires git provenant d'une plateforme d'hébergement et exécuter des programmes en C++ par exemple.

7.2 Image déjà existante

Une image docker peut être exécutée sur n'importe quelle machine, du moment qu'elle possède Docker. La plateforme d'hébergement Docker Hub offre de multiples images disponibles au téléchargement.

Il est possible de modifier les images provenant de la plateforme ou celles créées localement à l'aide d'un Dockerfile. Pour se faire, il faut lancer l'image. Il faut savoir qu'une image ne s'exécute pas en tant que telle, mais qu'elle est cloisonnée dans un container. Ci-dessous une représentation des différents environnements pour mieux comprendre la nomenclature et le fonctionnement de Docker :

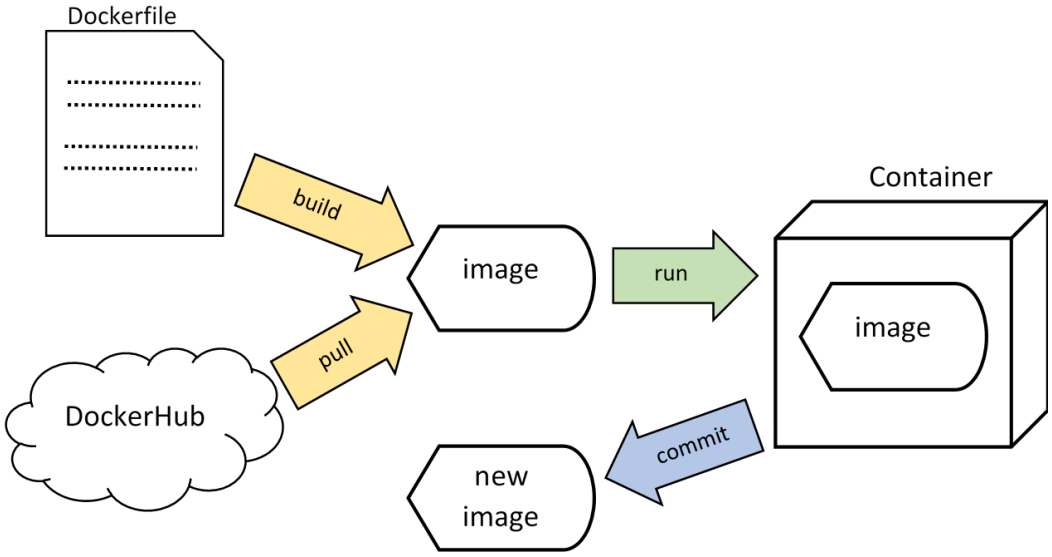


Figure 20: Représentation des étapes concernant Docker

Ainsi, une image n'est rien d'autre que le contenu d'un environnement. Pour que celui-ci soit actif, il faut le lancer à l'aide la commande « run ». De cette manière, l'image est lancée dans un container. À l'aide d'un « docker commit », une nouvelle image est générée, qui est le contenu à l'état actuel de l'image.

Grâce à cela, il est possible d'avoir une image de base avec, par exemple, Ubuntu, qui est lancée dans un container. Depuis celui-ci, des programmes sont installés. Afin de ne pas devoir effectuer ces installations à chaque fois, il est possible de faire un docker commit du container, qui va concrètement générer une nouvelle image. Dès lors, en lançant la nouvelle image, les programmes précédemment installés sont déjà inclus. Voici une représentation graphique de ceci :

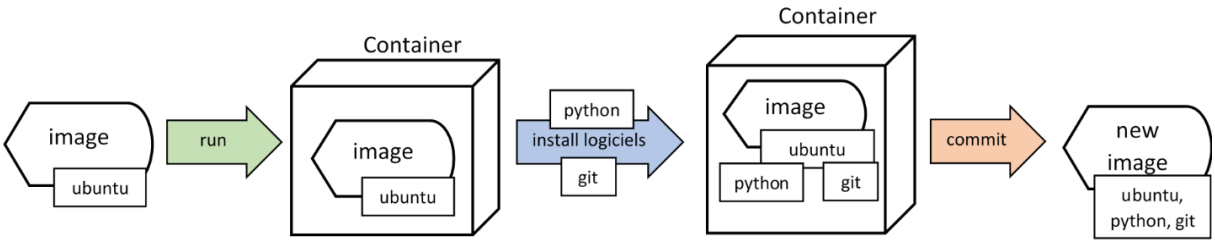


Figure 21: Représentation d'une modification d'image en direct