MGL7760 – PROJET 2

Mise en place un environnement d'intégration continue avec Jenkins

Yao Kevin Amouzou : AMOY02319402

Khadija Dia : DIAK91360008

**DOCUMENT DE SYNTHESE**

Sommaire

[I. Introduction 2](#_Toc130005137)

[II. La construction logicielle (build) avec make 3](#_Toc130005138)

[a. Expliquez comment vous l'invoquez à partir de la ligne de commande 3](#_Toc130005139)

[b. Expliquez comment vous l'invoquez à partir de Visual Studio Code ? 3](#_Toc130005140)

[c. Expliquez les tâches accomplies par votre build 3](#_Toc130005141)

[III. Environnement d'intégration continue avec Jenkins 5](#_Toc130005142)

[a. Expliquez la mise en place de l'environnement CI 5](#_Toc130005143)

[b. Expliquez le fonctionnement de l'environnement CI 5](#_Toc130005144)

[IV. Importance des enjeux reliés à la qualité et à la productivité 6](#_Toc130005145)

[a. Par l'utilisation d'un build 6](#_Toc130005146)

[b. Par l'utilisation d'un environnement CI 6](#_Toc130005147)

[CONCLUSION 7](#_Toc130005148)

# I. Introduction

Le projet que nous présentons est un exemple de mise en place d'un environnement de développement logiciel basé sur des pratiques de qualité telles que l'automatisation des tâches de build et de tests, ainsi que l'utilisation de l'intégration continue (CI). L'objectif de ce projet est de fournir un guide pratique pour la mise en place d'un tel environnement, en utilisant des outils couramment utilisés dans l'industrie, tels que Jenkins, Docker et Make. Nous expliquons également comment écrire des tests unitaires en Python et comment générer des rapports de couverture de code à l'aide de *coverage*. Nous montrons comment automatiser le processus de build et de tests en utilisant un Makefile, et comment intégrer ces tâches dans un pipeline Jenkins. Enfin, nous discutons de l'importance des pratiques de qualité dans le développement logiciel, en mettant en évidence les avantages de l'automatisation des tâches de build et de tests, ainsi que l'utilisation de l'intégration continue pour améliorer la qualité du code et la productivité des développeurs.

# II. La construction logicielle (build) avec make

Le Makefile est un fichier qui permet d'automatiser l'exécution de tâches récurrentes, telles que la compilation, la génération de documentation, l'exécution de tests, etc. Il contient des règles qui spécifient comment les fichiers sources doivent être compilés pour générer des fichiers exécutables ou des documents. Il est ici utilisé pour automatiser le processus de construction (ou build) de notre projet.

## a. Expliquez comment vous l'invoquez à partir de la ligne de commande

Nous pouvons exécuter les cibles de notre makefile en utilisant la commande "*make*" suivie du nom de la cible. Par exemple : **make unittest** exécute la cible unittest.

Pour invoquer le build à partir de la ligne de commande, il suffit d'ouvrir un terminal dans le répertoire où se trouve le Makefile et d'exécuter la commande **make build**. Cela va lancer le build et exécuter toutes les étapes définies dans le Makefile.

## b. Expliquez comment vous l'invoquez à partir de Visual Studio Code ?

Pour invoquer le build à partir de VS Code, nous utilisons l'extension *Tasks* de l’IDE. Voici les étapes à suivre :

On ouvre le menu "Terminal" dans la barre de menu supérieure de VS Code, puis on ouvre un "Nouveau terminal".

Dans le terminal, on tape la commande **make -p** pour lister les tâches disponibles. **Puis make build** pour exécuter la tâche de build.

## c. Expliquez les tâches accomplies par votre build

**Notre makefile contient plusieurs cibles pour effectuer différentes tâches :**

* "***lint***" : cette cible exécute *Pylint* pour effectuer une analyse statique du code et détecter les erreurs de style.
* "***unittest***" : cette cible exécute les tests unitaires avec *unittest*.

* "***docs***" : pour générer la documentation HTML pour le code. Cela générera une documentation HTML dans le dossier "docs" du projet.
* "***coverage***" : pour vérifier la couverture du code.
* **"*all*"** : exécute toutes les tâches *(lint, unittest, docs, coverage*) en une seule fois.

Le Makefile peut générer des rapports XML pour les résultats des tests en utilisant l’outil unittest par exemple.

Pour générer un rapport de tests en format XML avec unittest on a installé le module "xmlrunner" qui permet de générer des rapports XML à partir des résultats de tests unittest.

Dans notre code de tests, on utilise le module *xmlrunner* pour générer le rapport XML à la fin de l'exécution des tests. On a le code suivant a la fin du fichier de test :

**if \_\_name\_\_ == '\_\_main\_\_':**

**with open('resultats\_tests.xml', 'wb') as output:**

**unittest.main(**

**testRunner=xmlrunner.XMLTestRunner(output=output),**

**failfast=False, buffer=False, catchbreak=False)**

Puis dans le makefile, on a rajouté la ligne suivante dans la cible unittest pour exécuter les tests.

**python -m unittest discover -s testunit -p "\*.py" -v**

On indique à Python de lancer le module unittest, d'exécuter tous les tests présents dans le répertoire “testunit” avec le motif de recherche des fichiers de test avec l'extension .py et enfin d’afficher les résultats des tests à la console.

Cela exécutera les tests et générera un rapport de tests en format XML dans un fichier appelé "resultats\_tests.xml".

# III. Environnement d'intégration continue avec Jenkins

## a. Expliquez la mise en place de l'environnement CI

**Environnement d'intégration continue avec Jenkins :**

Pour mettre en place un environnement CI avec Jenkins, nous avons d'abord installé *Minikube* et *Jenkins* sur nos machines de développement. Une fois que les installations sont terminées, nous accédons à l'interface web de *Jenkins* via l'URL<http://localhost:8080> et configurons les paramètres de base, tels que les informations d'identification. Ensuite, il faut installer le plugin *Kubernetes* pour permettre à *Jenkins* de communiquer avec le cluster *Kubernetes*. Il est également important de configurer *Jenkins* pour surveiller le référentiel de code source pour les changements de code. Nous l’avons fait en utilisant un plugin *Git*. Une fois le référentiel de code source configuré, Nous avons créé le pipeline CI. Le pipeline est constitué d'une série d'étapes qui sont exécutées en séquence chaque fois qu'un nouveau code est poussé dans le référentiel de code source. Chaque étape du pipeline doit être configurée pour exécuter les tâches nécessaires, telles que la compilation, les tests unitaires et la génération de rapports. Enfin, nous avons jugé important de configurer les notifications pour être informé par e-mail ou par message texte en cas d'échec d'une étape ou de détection d'erreurs. Une fois le pipeline configuré, nous pouvons exécuter et surveiller le processus de build en temps réel à l'aide des tableaux de bord et des outils de surveillance fournis par *Jenkins*.

## b. Expliquez le fonctionnement de l'environnement CI

Le fonctionnement de l'environnement CI est basé sur un cycle continu d'intégration, de construction, de test et de déploiement. Voici les différentes étapes de ce cycle :

1. Intégration : chaque développeur envoie son code source dans le référentiel de code source partagé Git.
2. Build : lorsque le code source est mis à jour, le système de build, ici **Make,** est automatiquement déclenché. Il compile le code source, crée des exécutables,exécute les tests unitaires, génère de la documentation, etc.
3. Test : une fois que le système de build a créé une version exécutable du logiciel, le système de test est automatiquement déclenché. Les tests vérifient si l’application fonctionne correctement et si les exigences de qualité sont satisfaites.
4. Déploiement : si les tests sont réussis, le logiciel est déployé sur un environnement de test, de pré-production ou de production. Le déploiement peut être manuel ou automatique, en fonction de la configuration de l'environnement CI.
5. Notification : si le processus de build ou de test échoue, nous sommes notifiés par courrier électronique, message texte ou notification de l'outil CI.

L'avantage de l'environnement CI est que ce processus est répété chaque fois qu'un nouveau code source est ajouté au référentiel de code source. Cela permet d'identifier rapidement les erreurs de code, de résoudre les problèmes de qualité et de garantir que chaque nouvelle version du logiciel est stable et prête à être déployée.

# IV. Importance des enjeux reliés à la qualité et à la productivité

## a. Par l'utilisation d'un build

En utilisant un build, on peut automatiser le processus de construction de notre logiciel. Cela permet de garantir la cohérence et la qualité de notre produit final. Par exemple, le build peut effectuer des tâches telles que la compilation, les tests unitaires, la génération de rapports de couverture de code, l'analyse statique du code, la création de packages et la génération de documentation. En automatisant ces tâches, nous pouvons détecter rapidement les erreurs et les problèmes de qualité dès qu'ils surviennent, ce qui permet d'économiser du temps et de l'argent.

## b. Par l'utilisation d'un environnement CI

L'environnement CI est comme un assistant qui travaille en arrière-plan pour s'assurer que tout se passe bien lorsque des modifications sont apportées au code source d'un projet. Il vérifie automatiquement que les modifications sont correctement intégrées au code existant en effectuant des tests et en construisant le logiciel Si le build ou les tests échouent, l'environnement CI envoie une notification immédiatement au développeur afin qu'il puisse corriger le problème avant que cela ne devienne un problème plus important. Cela permet de maintenir la qualité du code et d'améliorer la productivité de l'équipe de développement.

L’intégration continue apporte de nombreux avantages pour la gestion de projet logiciel. En effet, elle permet de réduire les risques en détectant les erreurs et les incohérences dans le code source dès que possible, ce qui permet de les corriger rapidement et d'éviter les problèmes plus tard dans le cycle de développement. De plus, l'intégration continue permet de réduire les processus manuels répétitifs liés à la construction, aux tests et au déploiement du logiciel, ce qui permet de gagner du temps et de réduire les coûts. Elle permet également de déployer du code à la demande, en facilitant la livraison continue. L'intégration continue améliore également la visibilité du projet en fournissant des informations précises sur l'état de la construction, des tests et du déploiement du logiciel. Enfin, elle améliore la confiance de l'équipe de développement dans le produit logiciel en fournissant une assurance qualité continue et en garantissant la cohérence et la qualité du code source tout au long du cycle de développement.

# CONCLUSION

Suite du projet 1, le projet 2 consistait à mettre en place un environnement d'intégration continue ou CI avec Jenkins. L’objectif était de se familiariser avec des outils essentiels dans le développement de logiciels, à savoir un outil de construction logicielle (build) tel que Make et un outil d'intégration continue (CI) tel que Jenkins. Il s’agissait de comprendre l'importance de ces outils dans l'amélioration de la qualité et de la productivité du processus de développement de logiciels. Au cours de ce projet, Nous avons pu découvrir les avantages de l'intégration continue, notamment la réduction des risques, la réduction des processus manuels répétitifs, l'amélioration de la visibilité du projet et la confiance de l'équipe de développement dans le produit logiciel.