MASTER 2 E-SERVICES

MÉMOIRE DE STAGE

JUIN 2016 - AOÛT 2016

Mise en place d'un système d'intégration continue et de contrôle de qualité en environnement de recherche

KNAPIK Christopher

Tuteur de stage : M. Fabrice DUCOS
Tuteur universitaire : M. Ioan Marius BILASCO

Remerciements

Je tiens à adresser mes remerciements aux personnes qui m'ont aidé dans la réalisation de ma mission de stage ainsi que de mon mémoire.

En premier lieu, je remercie M. Ducos, mon tuteur au cours de mon stage, qui m'a aidé et guidé autant pour ma mission de stage au sein du LOA que pour la rédaction de ce mémoire.

Je remercie également M. Bilasco, tuteur universitaire et responsable de formation, qui m'a aidé et donné des conseils dans l'orientation du mémoire et de sa rédaction.

Enfin, je remercie l'ensemble de l'équipe du LOA pour leur accueil chaleureux au sein de leurs locaux.

Table des matières

Chapitre 1

Introduction

1.1 Présentation du laboratoire

La mission de stage a été effectuée au Laboratoire d'Optique Atmosphérique (LOA), qui est situé sur le campus de l'Université Lille 1 à Villeneuve d'Ascq. Le LOA est une Unité Mixte de Recherche (UMR) du CNRS : Une partie du personnel est employée par le CNRS, tandis qu'une autre partie est employée par l'Université elle-même. Le LOA est spécialisé dans l'étude des propriétés de l'atmosphère terrestre par voie optique.

L'optique atmosphérique cherche à modéliser la propagation à travers l'atmosphère de la lumière visible reçue du soleil et de la lumière infrarouge émise par l'ensemble des surfaces et de l'atmosphère terrestres. Les travaux menés au LOA dans ce domaine s'insèrent dans l'étude globale du climat.

Un premier objectif est de quantifier le rôle de ce rayonnement visible et infrarouge dans les échanges énergétiques de la planète, en particulier de préciser le rôle des nuages dans le bilan radiatif de la terre dont ils constituent un facteur essentiel.

Un second axe de recherche porte sur la caractérisation à l'échelle du globe de différents paramètres qui sont en relation directe avec l'évolution climatique (nuages, aérosols, surfaces), en utilisant principalement l'observation satellitaire.

Les travaux menés dans ce contexte mettent en oeuvre :

- La conception de logiciels permettant de simuler le transfert du rayonnement, à l'aide de modèles du système terre - atmosphère.
- L'analyse d'observations acquises par les capteurs satellitaires existants, le plus souvent sous forme d'images traitées sur ordinateur, et la conception d'expériences satellitaires nouvelles.
- La réalisation de campagnes d'observation de terrain, utilisant des appareillages développés par le laboratoire, mis en oeuvre au sol ou à partir d'avions ou de ballons stratosphériques, et destinés à valider les modèles ou à mettre en évidence les processus atmosphériques.

1.2 Contexte et objectifs

Les exigences de qualité inhérentes à des travaux faisant l'objet de publications, les contraintes du développement en équipe ou en collaboration avec d'autres laboratoires et sociétés privées, la diversification des projets, le renouvellement régulier des participants (doctorants, post-doctorants, contractuels), conduisent le LOA à l'utilisation d'outils de gestion de configuration (subversion, git) et de développement dirigé par les tests (TDD).

Même si aujourd'hui, l'utilisation d'outils de gestion de configuration ainsi que la rédaction de tests est courante, cela n'a pas toujours été le cas. En effet, de nombreux projets débutés il y a plusieurs dizaines d'années contiennent d'importantes parties de code non testées, dans des langages anciens comme le Fortran 90.
Il serait donc nécessaire de passer un temps extrêmement important pour écrire les tests contrôlant ces lignes de code, ce qui n'est pas possible malheureusement.

L'objectif de ce stage s'inscrit dans la poursuite de cet effort de mise à jour des pratiques de développement informatique, afin d'améliorer la qualité des futurs projets. C'est dans ce contexte que l'utilisation de l'intégration continue entre en jeu afin d'avoir un regard constant sur l'état des développements ainsi que sur la qualité du code. En effet, le but est de détecter au plus vite l'ajout d'éventuels problèmes ou erreurs au code existant, ainsi que de contrôler la qualité du code. Cela passe par la détection d'erreurs, la vérification des résultats des tests, la vérification de leur couverture, ou encore la détection de code dupliqué.

La mission principale consiste en la mise en place d'un serveur d'intégration continue, équipé d'outils de contrôle de qualité (couverture par les tests, contrôle de la redondance de code etc) au sein du laboratoire.

Le système propose des solutions permettant de travailler efficacement au minimum dans les langages suivants, particulièrement utilisés dans notre domaine de recherche : C, C++, Python ou encore Fortran90. Des prototypes d'applications basiques dans ces différents langages, avec des systèmes de construction divers, ainsi que la rédaction de tests unitaires, permettent ainsi de tester et valider la mise en place de la solution.

Un autre objectif du stage consiste en la mise en place d'environnements de tests pour les codes déployés sur le serveur d'intégration continue. Des solutions de type machine virtuelle ou conteneur (type Docker) sont étudiées et la pertinence de leur couplage au système d'intégration continue évaluée.

Dans la première partie, des études seront réalisées à propos de divers outils permettant la mise en place d'un serveur d'intégration continue ainsi que sur les solutions autour des environnements de tests, dans le but d'avoir un aperçu sur les qualités et défauts de chacun et vérifier s'ils correspondent ou non aux besoins.
Des démonstrations à l'aide des outils retenus permettront ensuite de constater les apports possibles aux futurs projets. Un bilan sera ensuite établi à propos des résultats obtenus, ainsi que sur les perspectives éventuelles afin de continuer à améliorer la solution.

Chapitre 2

Etat de l'art

2.1 L'intégration continue

2.1.1 Définition

Avant d'aborder la suite, il est important de bien comprendre ce qu'est l'intégration continue. "L'intégration continue est un ensemble de pratiques utilisées en génie logiciel consistant à vérifier à chaque modification de code source que le résultat des modifications ne produit pas de régression dans l'application développée"[1]. En d'autres termes, l'intégration continue sert à vérifier que l'ajout ou la modification de lignes de code n'a pas introduit ou découvert d'erreurs ou de problèmes dans le code existant.

Différence avec l'intégration classique En règle générale, l'intégration classique consiste à assembler à la fin du projet toutes les fonctionnalités développés. Le projet est ensuite testé et une longue phase de tests et de corrections de bogue débute. Néanmoins, il est plus difficile de corriger le code une fois que tout est réalisé qu'au fur et à mesure du projet. Cela entraîne des coûts supplémentaires de développement et éventuellement, un manque de temps peut provoquer la sortie du projet contenant toujours d'importants bogues.

Fonctionnement Ce sont ces différents problèmes que l'intégration continue cherche à résoudre. Pour simplifier le processus d'intégration continue, on utilise des serveurs d'intégration continue, qui sont des outils permettant de gérer des projets et configurer la compilation ainsi que le processus de déploiement. La plupart de ces outils disposent de fonctionnalités permettant la génération de rapports ou de métadonnées à propos des projets. L'utilisation du schéma ci-dessous permet de comprendre et d'expliquer plus en détail ce qu'est l'intégration continue ainsi que son fonctionnement.

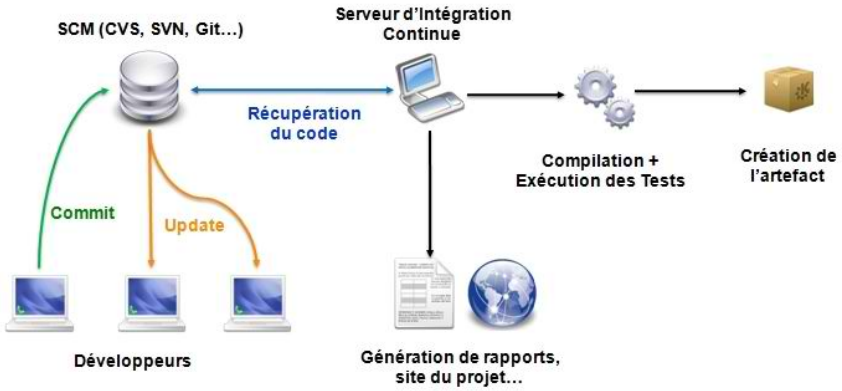

FIGURE 2.1 – Schéma expliquant l'intégration continue

Etape 1 Les différents développeurs du projet travaillent ensemble et développent les fonctionnalités et modules qui leur sont attribués. Chacun veille à tester le code écrit à l'aide de tests unitaires, et vérifie que tout fonctionne dans son environnement. Une fois testé, chaque développeur peut ensuite envoyer son code auprès du gestionnaire de versions. A chaque envoi (appelé *commit*), le code est fusionné au code déjà existant. Les autres développeurs peuvent ensuite récupérer et mettre à jour (*update*) le code sur leur machine.

Etape 2 L'utilisation d'outil d'intégration continue permet de récupérer le code publié sur le gestionnaire de versions. Cela peut être réalisé automatiquement à chaque *commit*, à intervalles réguliers ou manuellement. Ce choix dépend purement de l'outil utilisé ainsi que de la volonté de l'utilisateur.

Etape 3 Une fois le code récupéré par le serveur d'intégration continue, il est ensuite possible de compiler ce code ainsi que d'exécuter les tests qu'il contient.

Etape 4 Une fois la compilation et l'exécution des tests terminés, les informations sur la réussite ou non de l'étape précédente sont transmises à l'utilisateur. De plus, des rapports concernant la qualité, la stabilité ou la découverte de bogues peuvent être générés.

Etape 5 Les rapports peuvent ensuite être analysés afin de corriger les éventuels problèmes rencontrés.

Grâce aux tests très réguliers du code développé, les problèmes sont détectés dès leur apparition. Il est ainsi plus facile et rapide de les corriger : le projet n'étant pas terminé, il y a de fortes chances d'avoir moins de parties du code à modifier, ce qui entraîne un gain de temps au final.

2.1.2 Les outils d'intégration continue

De nombreux outils d'intégration continue sont disponibles actuellement, avec leurs qualités et leurs défauts. Quelques uns d'entre eux seront décrits dans la suite, avec leurs principales qualités et défauts. En plus de leurs pages respectifs, des comparatifs[2] incluant un système de vote permet d'avoir un aperçu des fonctionnalités de nombreux outils d'intégration continue.

Jenkins

FIGURE 2.2 – Logo de Jenkins

Jenkins[3], débuté en 2011 comme branche d'un autre outil populaire Hudson, est l'outil le plus utilisé pour l'intégration continue. C'est un serveur conçu en Java qui, grâce à ses plusieurs centaines de plugins, permet la découverte de défauts au début du cycle d'un projet[4]. Il supporte un nombre important de langages et d'outils de compilation. Parmi ces langages, on peut noter le Java, .NET ou encore le C/C++ ou Python. Jenkins possède également une communauté très active, qui l'enrichit en plugins et apporte de l'aide sur de nombreux forums. De plus, Jenkins est totalement gratuit. On peut toutefois reprocher une documentation peu précise ainsi qu'une grande variété dans la qualité des plugins.

TeamCity

FIGURE 2.3 – Logo de TeamCity

TeamCity[5], dont la première version est sortie en Octobre 2006, est un serveur d'intégration continue et de gestion de compilation. Il supporte les langages Java, .NET et Ruby ainsi que de nombreux gestionnaires de versions comme Subversion, Git ou CVS. Certains outils de qualité de code sont directement intégrés, comme la couverture du code ou la recherche de duplication de code. TeamCity est payant, il est toutefois possible de l'utiliser gratuitement pour un maximum de 20 configurations de compilation.

Travis

FIGURE 2.4 – Logo de Travis CI

Travis CI[6] est une solution d'intégration continue à l'origine créée pour des applications Rails, mais qui supporte désormais des dizaines de langages comme PHP, Javascript, C/C++ ou encore Python. Contrairement aux précédents outils, Travis est hébergé en ligne : Aucune installation n'est donc nécessaire. Il dispose également d'une importante communauté aidant les nouveaux utilisateurs. Cela empêche donc de le personnaliser comme on pourrait le faire avec des plugins pour d'autres outils. Travis est intégré avec GitHub : il enregistre tout les ajouts de code et compile automatiquement ensuite. Il est toutefois limité à celui-ci, il n'est pas possible d'utiliser d'autres outils de gestion de version pour lui fournir le code source. Il est de plus payant pour les dépôts privés de GitHub.

2.1.3 L'outil retenu

Jenkins C'est l'outil Jenkins qui a été choisi et utilisé. L'outil permet de répondre aux besoins du LOA :

— Libre d'utilisation.
— Importante communauté, fournissant un nombre important de fonctionnalités grâce aux plugins.
— Support de nombreux langages, notamment les principaux langages utilisés par le LOA : Fortran, C/C++ et Python.
— Support des deux gestionnaires de version utilisés par le LOA, à savoir Subversion et Git.
— Le LOA souhaitait une solution auto-hébergée.

2.2 Les environnements de tests

2.2.1 Définition

"En informatique, un environnement désigne, pour une application, l'ensemble des matériels et des logiciels système, dont le système d'exploitation, sur lesquels sont exécutés les programmes de l'application."[7] Dans ce cas précis, un environnement de test est l'environnement dans lequel l'application sera testé.

2.2.2 Les outils autour des environnements de tests

Autrefois, les serveurs de test reposaient essentiellement sur des machines virtuelles afin de pouvoir disposer d'un environnement prédéfini. Il existe désormais une autre solution que les machines virtuelles : Docker[8].

Docker

Docker est un logiciel libre qui permet de déployer des applications dans des conteneurs logiciels. Il repose sur le kernel Linux et sur des conteneurs Linux LXC, qui permettent de faire fonctionner des environnements Linux isolés les uns des autres, mais partageant le noyau et les ressources du système hôte. Ces conteneurs, isolés les uns des autres et du reste de système, permettent donc d'embarquer et de travailler sur des applications avec un environnement similaire quelque soit le système d'exploitation. Longtemps limité aux systèmes d'exploitation Linux, Docker est maintenant utilisable également sur Windows et Mac.

Les machines virtuelles

Les machines virtuelles (ou *VM*) permettent, via un logiciel d'émulation, de créer une reproduction d'une machine réelle. Les caractéristiques techniques comme le processeur, la mémoire, le disque dur ainsi que le système d'exploitation sont simulées et il est alors possible d'exécuter différents programmes dans les mêmes conditions que la machine simulée.

Les VM sont utilisées pour différentes tâches. Elles sont par exemple utilisées :
— Pour l'émulation : Il est ainsi possible de reproduire les fonctionnalités d'une machine spécifique à l'aide d'une autre machine.
— Pour la virtualisation : Plusieurs machines sont simulées sur une même machine. Les capacités de la machine sont distribuées entre les différentes simulations.
— En programmation : Elles permettent d'exécuter des programmes dans un environnement contrôlé, ou également tester des programmes encore incomplets. Les ressources sont ainsi dédiées uniquement aux programmes choisis.

Comparaison Docker / Machines virtuelles

Ces deux solutions sont envisageables dans le cas de la création d'environnement de tests. Cependant Docker présente un grand avantage contrairement aux VM : les performances.

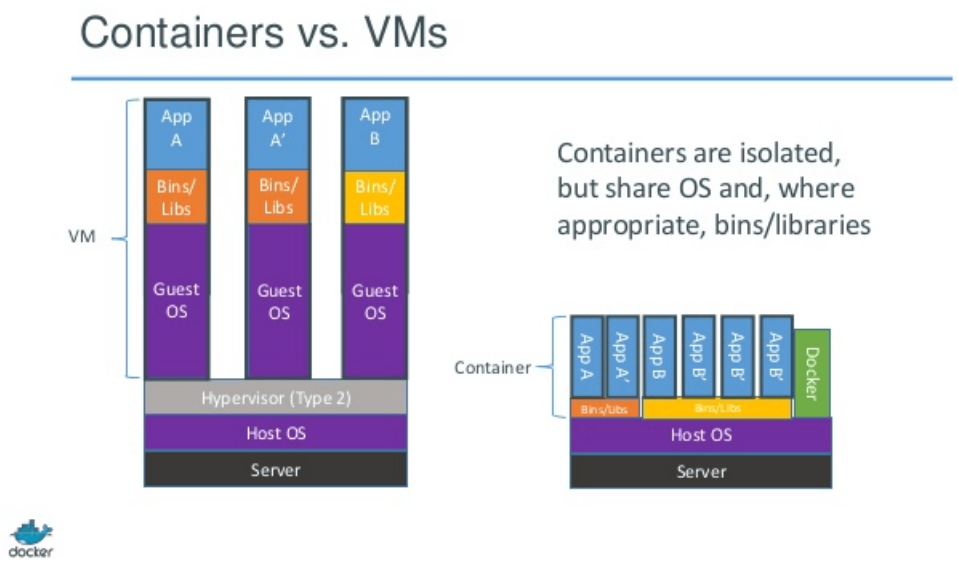

FIGURE 2.5 – Comparaison entre Docker et les VM[8]

En effet, les VM nécessitent pour chacune d'entre elles l'installation d'un système d'exploitation en plus de celui installé sur la machine hôte. Il est également nécessaire pour chaque VM d'installer les différentes librairies et fichiers binaires requis au bon fonctionnement du programme testé.

Les conteneurs créés par Docker sont isolés, mais partage néanmoins le même système d'exploitation de la machine hôte. Il est même possible dans certaines conditions de partager des librairies ou fichiers binaires.

L'utilisation de conteneurs est donc beaucoup moins lourde que celle des machines virtuelles, nécessitant l'installation d'un système d'exploitation propre à chacune d'entre elles.

2.2.3 L'outil retenu

Dans le cadre des besoins du LOA, l'utilisation de Docker est donc la meilleure option. Il est facile de créer des conteneurs possédant l'environnement souhaité afin de réaliser des tests. De plus, le travail régulier du LOA avec d'autres laboratoires ou sociétés induit un échange de code fréquent. Docker répond à ce besoin grâce à ses images, idéales pour reproduire les mêmes environnements de tests qu'au LOA mais également d'y inclure directement des applications prêtes à l'emploi.

Chapitre 3

La mise en place de l'intégration continue via Jenkins

Après avoir exposé le contexte de la mission ainsi qu'étudié quelques outils susceptibles de répondre à ses besoins, il est désormais temps d'aborder plus en détail les réalisations effectuées grâce aux outils retenus, à savoir Jenkins et Docker.

3.1 Présentation de Jenkins

Une fois la procédure d'installation terminée, il est ensuite possible d'accéder à l'interface de Jenkins grâce à un navigateur internet.

3.1.1 La liste des tâches

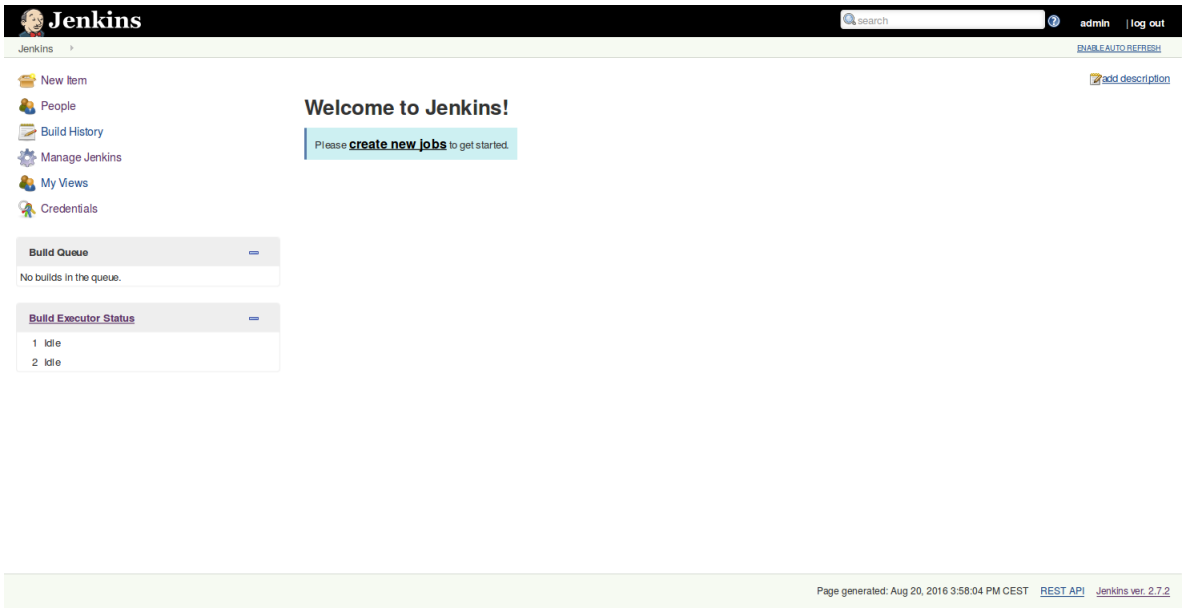

FIGURE 3.1 – Page d'accueil de Jenkins

Cette page, qui est la page d'accueil de Jenkins, contient la liste des tâches (*jobs* en anglais), c'est à dire la liste des projets ainsi que les différentes configurations établies. Elle permet de jeter un coup d'oeil rapide sur quelques éléments importants, à savoir :

— Une icône indiquant le résultat de la dernière compilation à l'aide de trois états : *Success*, *Unstable* ou *Failed*.

— Une icône indiquant l'évolution des compilations récentes comme leur stabilité, les résultats récents des tests, couverture de code etc.
— Les dates de dernière compilation réussie (ainsi que sa durée) et échouée.
— Une dernière icône permet directement de programmer une compilation.

3.1.2 Création d'une tâche

La création d'une tâche permet à l'aide de Jenkins d'ajouter un projet ainsi que configurer les différents paramètres concernant sa compilation. Ces ensembles de paramètres sont séparés en 6 catégories :

General La partie *General* permet la configuration élémentaire du projet, comme choisir son nom, lui donner une description. Elle permet également de choisir si on souhaite ou non supprimer les fichiers des précédentes compilations, entrer un lien GitHub où se trouve le projet ou encore désactiver temporairement le projet.

Source Code Management La partie *Source Code Management* permet de choisir le gestionnaire de versions utilisé pour le projet, comme par exemple Git ou Subversion. Cela permet de récupérer facilement le code source du projet à chaque modification apportée.

Build Triggers La partie *Build Triggers* permet de programmer le déclenchement de la compilation. Il est possible par exemple de déclencher la compilation à des horaires fixes, ou encore à chaque modification détectée sur le gestionnaire de versions utilisé.

Build Environment La partie *Build Environment* permet de configurer l'environnement dans lequel la compilation sera effectuée. Il est par exemple possible de supprimer les fichiers présents dans le dossier du projet Jenkins afin d'éviter des erreurs avec d'anciens fichiers. Il est également possible, grâce à l'ajout de plugins [1] (dont quelques uns seront vus en détail dans la section suivante), de compiler le projet dans un conteneur Docker. L'utilisation de Docker sera vue plus en détail dans une partie suivante.

Build La partie *Build* permet de définir les règles de compilation du projet. Il est par exemple possible de faire appel à Ant ou à un script Gradle, ou tout simplement d'utiliser des lignes de commande shell.

Post-build Actions La partie *Post-build Actions* permet de configurer toutes les opérations à effectuer une fois que la compilation est terminée. C'est une des parties les plus importantes de Jenkins car c'est dans cette partie qu'il est possible de générer des graphes sur les résultats de tests, de couverture du code, de détection d'erreurs dans le code, en faisant appel aux plugins appropriés. Il est également possible d'envoyer une notification par mail sur le résultat de la compilation.

Ce petit aperçu de l'utilisation de Jenkins effectué permet d'avoir une idée sommaire de son utilisation. Il faut maintenant s'attarder plus en détail ce qui la force de Jenkins : ses plugins.

1. Les plugins en informatique sont des modules qui permettent d'étendre les fonctionnalités de base d'un logiciel[9].

3.1.3 Les plugins

La force de Jenkins repose en grande partie sur les membres de sa communauté. Grâce à son utilisation libre, une importante communauté s'est formée autour de celui-ci et a contribué à son amélioration grâce au système de plugins. Plusieurs centaines de plugins sont disponibles, accessibles directement grâce à l'interface de Jenkins.

Il est très facile d'installer de nouveaux plugins : Un simple cocher/décocher permet d'installer ou de désinstaller les plugins souhaités. Des notifications sont ensuite affichées lorsque des mises à jour sont disponibles.

Les plugins sont l'atout principal de Jenkins, et pour cause : Jenkins seul ne fait pas grand chose. Ces principales fonctionnalités de base reposent sur la compilation de projets ainsi que leur planification. Il est ensuite possible d'afficher des logs sur le déroulement de la compilation.

Dans le cadre de l'amélioration globale de la qualité de code, des plugins permettent par exemple d'afficher des résultats produits par la compilation directement sur l'interface de Jenkins, et ainsi avoir une vue détaillée sur l'état actuel du projet. Ces résultats sont la mise en forme graphique de rapports produits par diverses librairies, regroupant ainsi l'ensemble des rapports et simplifiant grandement leur lecture. Des exemples de librairie ainsi que de leur utilisation seront vus par la suite.

3.2 Vers l'amélioration de la qualité du code

La section suivante tâchera de présenter les pratiques ainsi que les plugins pouvant être intéressants pour les projets futurs du laboratoire.

3.2.1 Les tests unitaires

Afin d'améliorer la qualité globale d'un projet, la rédaction de tests unitaires est un point essentiel. Les tests unitaires se réfèrent à la pratique de tester certaines fonctions ou parties du code. Cela nous donne la possibilité de vérifier que celles-ci fonctionnent comme prévu [10]. Il faut par exemple tester pour diverses valeurs d'entrées si le résultat obtenu correspond bien à ce qui était attendu.

Grâce à Jenkins et certains plugins, il est possible d'obtenir visuellement les résultats de tests unitaires. Des détails sont ainsi générés sur leurs résultats, et des courbes indiquant les tests réussis et échoués sont établies. Associé à des librairies mesurant la couverture du code, il est possible d'avoir une idée précise du pourcentage de code parcouru par les tests unitaires.

Les outils XUnit Les outils XUnit sont des outils permettant de réaliser des tests unitaires dans différents langages informatiques. Le *X* contenu dans *XUnit* est généralement remplacé par le langage concerné. Par exemple, l'outil de tests unitaires le plus connu est JUnit qui concerne le langage Java.

JUnit plugin et XUnit plugin Les outils XUnit permettent pour la plupart d'exporter les résultats des tests unitaires sous forme de fichiers XML. Naturellement, Jenkins ne permet pas de réaliser des tests unitaires. Cependant, les fichiers XML générées lors de ces tests sont lisibles grâce à deux plugins *JUnit plugin*[11] et *XUnit plugin*[12]. Ces deux plugins lisent le contenu des fichiers XML et fournissent la liste des tests exécutés ainsi que le nombre de tests réussis et échoués sous forme de graphe.

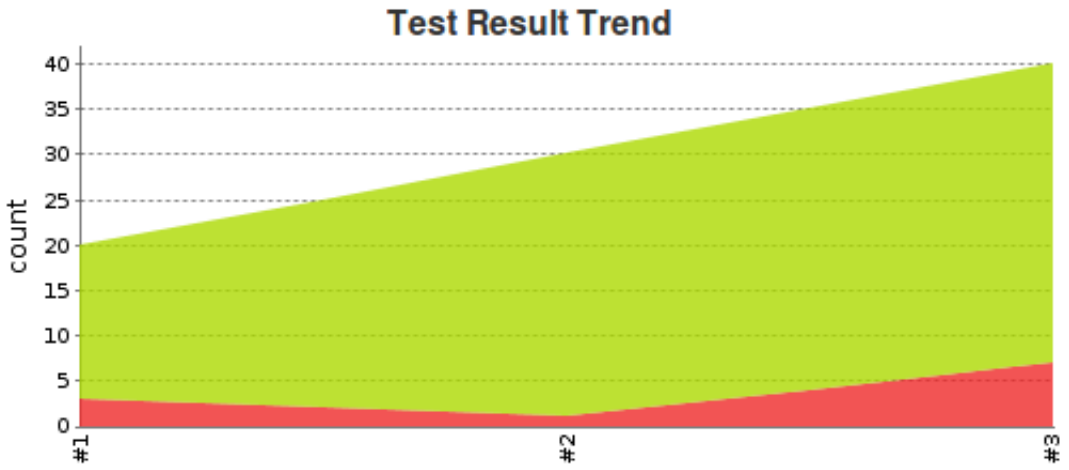

FIGURE 3.2 – Exemple de résultats de tests unitaires

Ces graphes sont très utiles pour évaluer l'évolution d'un projet. Il est ainsi possible d'observer une éventuelle augmentation du nombre de tests échoués, pouvant par exemple indiquer qu'un ajout de code récent a modifié le comportement d'une fonctionnalité du programme. L'évolution du nombre de ces tests peut également donner des indications sur le code ajouté : En effet, si de nombreux ajouts ont été effectués mais que le nombre de tests n'a pas changé, il est fort probable que ces lignes de code n'aient pas été testées.

3.2.2 La couverture de code

La couverture de code est associée aux tests du code et permet de s'assurer que les tests parcourent bien l'ensemble du programme testé. Les tests permettent de vérifier que les résultats obtenus sont ceux qui étaient attendus. La couverture du code quant à elle permet de mesurer quelles parties du code ont bien été testées. En effet, disposer de tests, même s'ils sont tout à fait corrects, n'est pas grandement utile s'ils ne s'adressent qu'à une infime partie du code du projet. Ainsi, l'association de tests de qualité couplée à la mesure de leur couverture garantit que l'ensemble du programme a bel et bien été testé.

Pour obtenir les résultats concernant la couverture du code, il est nécessaire d'utiliser des librairies qui effectuent ces mesures. Parmi ces librairies, on peut par exemple citer :

— *Coverage.py*, fonctionnant pour le langage Python.
— *gcovr*, inspiré de *Coverage.py* et fonctionnant pour divers langages basés sur le compilateur GNU comme le C ou le C++.

Cobertura plugin D'autres librairies existent et peuvent être utilisées, mais une condition est cependant nécessaire afin de garantir leur association avec Jenkins : il faut qu'il soit possible de générer les résultats sous forme de fichiers XML de type Cobertura [2].

Afin de lire les résultats de couverture de code, il est nécessaire d'utiliser un plugin Jenkins. Le plugin *Cobertura plugin*[13] permet ainsi de lire un fichier XML de type Cobertura et ainsi obtenir le détail au travers de l'interface Jenkins. Parmi ce détail se trouve l'ensemble des lignes de code

2. Cobertura est un outil Java qui mesure le pourcentage de code parcouru en Java.

parcourues ainsi que des courbes indiquant le pourcentage du parcours de classes, conditions, fichiers, lignes etc.

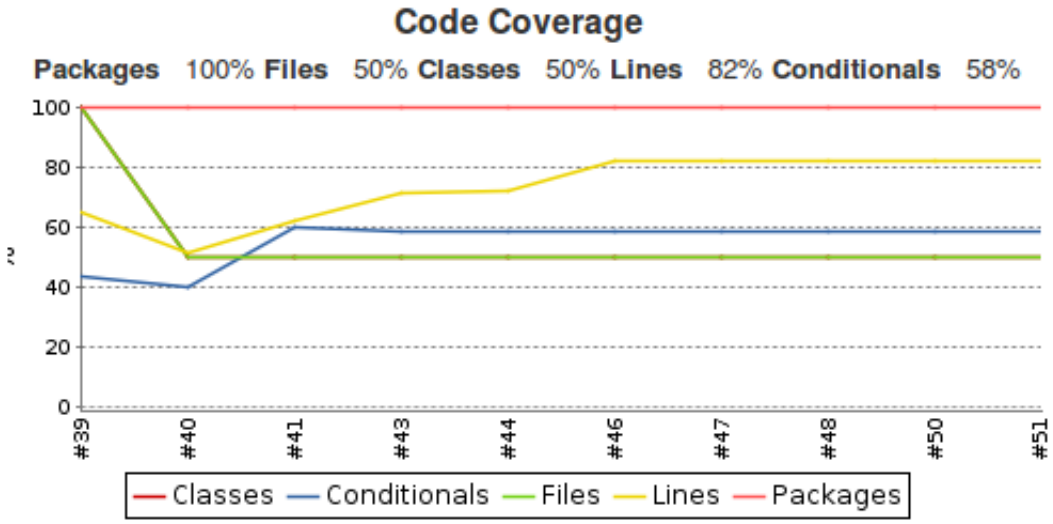

FIGURE 3.3 – Exemple de résultats de couverture de code par les tests

Associée aux tests unitaires, la couverture du code est essentielle dans la garantie de la qualité du code. Les courbes obtenues peuvent permettre de déceler au plus tôt une modification des habitudes dans le code comme par exemple :

- La qualité des tests unitaires diminue, les tests récents ne parcourant pas convenablement le code ajouté.
- La non-rédaction de tests unitaires.
- La venue de nouveaux collaborateurs, possédant des habitudes différentes, qui n'ont pas rédigé de tests unitaires.

Il est donc plus facile de détecter et de résoudre rapidement de mauvaises habitudes afin de maintenir tout au long du projet un code de qualité.

3.2.3 La duplication de code

Parmi les erreurs courantes en programmation se trouve la duplication de code. C'est "[...] une erreur courante de conception de logiciels où une suite d'instructions similaires (voire identiques) existe en plusieurs endroits du code source d'un logiciel."[14] Ces duplications apparaissent souvent suite à l'utilisation du copier/coller. Cela peut provoquer d'importants problèmes si un problème est détecté dans cette partie du code ou une modification est nécessaire. En effet, il faut alors corriger toutes les parties copiées, en prenant le risque d'en oublier et ainsi allonger la durée de correction de bogues.

Afin de continuer l'effort effectué concernant la qualité de code, il convient donc de détecter et corriger toutes ces duplications de code. Une solution est de réaliser une factorisation du code, c'est à dire regrouper les instructions dupliquées à l'intérieur d'une fonction.

Pour détecter où se trouve les duplications, l'utilisation d'une librairie facilite grandement la tâche. La librairie CPD (pour *Copy/Paste Detector*) permet de parcourir le code à la recherche des duplications et génère les résultats au format XML.

DRY plugin Le plugin *DRY plugin* (pour *Don't Repeat Yourself*)[15] permet de lire le fichier XML généré grâce à la librairie CPD et ainsi obtenir le résultat détaillé sur l'interface de Jenkins. Ces résultats sont textuels et indiquent les lignes où se situent les duplications, ainsi qu'une coloration des lignes de code concernées.

Cet ensemble de démarches permettent d'effectuer une observation continue de l'état des projets, de sorte à garantir une qualité tout au long de leurs développements.

Chapitre 4

Environnement de tests avec Docker

Cette partie donne un aperçu de l'utilisation de Docker, ainsi que son intérêt pour les projets futurs menés au sein du LOA.

4.1 Présentation de Docker

Les projets sur lesquels le LOA travaille sont nombreux et variés, et sont développés par de nombreuses personnes. Il arrive souvent que ces projets soient également développés en coopération avec d'autres équipes ou laboratoires. Par exemple, le LOA travaille très souvent avec le laboratoire ICARE.

C'est ici que l'utilisation de Docker devient intéressante. En effet, le fait de travailler avec un nombre important de personnes (et qui possède donc tous des machines ou systèmes d'exploitation différents) entraîne souvent des problèmes. Il n'est pas rare de se retrouver avec un projet contenant un bogue important, qui n'a pourtant pas été détecté à l'origine par celui qui l'a introduit car le problème n'est tout simplement pas apparu sur sa machine lors de ses essais.

FIGURE 4.1 – Logo de Docker

Docker est un logiciel libre qui permet de déployer des applications dans des conteneurs logiciels. Il repose sur le kernel Linux et sur des conteneurs Linux LXC, qui permet de faire fonctionner des environnements Linux isolés les uns des autres, mais partageant le noyau et les ressources du système hôte. Ces conteneurs, isolés les uns des autres et du reste de système, permettent donc d'embarquer et de travailler sur des applications avec un environnement similaire.

4.2 Utilisation de Docker

4.2.1 Scénario d'utilisation

Pour comprendre plus facilement l'intérêt que peut avoir Docker, un scénario d'utilisation peut s'avérer utile.

Annie et Bob travaille ensemble sur un projet PHP.
Annie possède une machine avec la dernière version du système d'exploitation Ubuntu, ainsi que la toute dernière version de PHP.
Bob quant à lui possède une machine Debian, et travaille toujours sur une version plus ancienne de PHP.
Résultat : Il est fort possible que des problèmes de compatibilité surviennent dans leurs codes respectifs, même en présence de tests. De plus, il est possible que le serveur de production sur lequel est déployé le code possède un environnement également différent d'Annie et Bob.
A l'aide d'une image Docker, construit à partir d'un Dockerfile par exemple (qui sera vu plus en détail dans la partie suivante), Annie et Bob peuvent désormais tous les deux travailler dans un environnement similaire à leur environnement de production.

4.2.2 Conteneurs et images

Comme indiqué précédemment, les conteneurs Docker permettent d'embarquer des applications dans des environnements personnalisés isolés du système. Ces conteneurs sont construits à partir d'une image Docker. Par exemple, une image Docker peut contenir un système d'exploitation Ubuntu avec les différentes librairies nécessaires au bon fonctionnement de l'application. Les images Docker reposent sur l'héritage. En effet, une image peut être construite à partir d'une autre image, qui elle-même a été construite grâce à une troisième image.

4.2.3 Construction d'une image

Plusieurs méthodes permettent de créer une image : Il est possible de construire une image en lançant un conteneur, effectuer une suite d'actions à l'intérieur, puis sauvegarder les changements apportés. Une autre méthode plus simple existe également : le Dockerfile. Contrairement à la première méthode, le Dockerfile permet de créer une image à partir d'une image initiale, sans avoir à lancer un conteneur au préalable. On y ajoute une série d'actions, qui permettront la construction de l'image.

Voici quelques exemples de commandes que l'on peut trouver dans un Dockerfile. Il en existe évidemment bien d'autres.

- FROM : La commande *FROM* permet de définir l'image à partir de laquelle on part. Dans l'exemple, on part d'une image contenant le système d'exploitation Ubuntu.
- RUN : La commande *RUN* permet d'exécuter des commandes, qui le seront lors de la construction de l'image. On peut par exemple penser à l'installation de librairies nécessaires au bon fonctionnement de l'application embarquée.
- ADD : La commande *ADD* permet d'ajouter des fichiers à l'intérieur de l'image.
- WORKDIR : La commande *WORKDIR* permet de définir le répertoire courant de l'image, où seront exécutées les commandes qui suivent.
- CMD : La commande *CMD* permet de définir la commande qui sera exécuté au lancement du conteneur.

Un exemple de Dockerfile est présent en annexe. Il permet de compiler dans un conteneur un projet actuellement en cours de développement au LOA.

4.3 L'association de Docker avec Jenkins

Afin de compiler un projet dans un environnement défini et contrôlé, il est possible d'utiliser le système de conteneurs de Docker directement à l'intérieur de Jenkins.

Cloudbees Docker Custom Build Environment Plugin Il existe plusieurs plugins autour de Docker. Le plugin *Cloudbees Docker Custom Build Environment Plugin*[16] a été choisi car il permet, simplement à l'aide d'un fichier Dockerfile, de lancer la compilation à l'intérieur d'un conteneur créé à partir de ce fichier.

Cette association entre Jenkins et Docker peut s'avérer très utile pour tester un projet dans un environnement choisi, sans avoir à modifier l'environnement sur lequel se trouve le serveur Jenkins. Cela évite également d'installer toutes les librairies nécessaires aux différents projets présents sur le serveur Jenkins : les librairies sont alors installées uniquement dans les conteneurs produits grâce au Dockerfile.

Chapitre 5

Conclusion

5.1 Synthèse

Grâce à l'utilisation de l'intégration continue avec le serveur Jenkins, il est possible d'observer l'évolution des projets de manière continue. Il est ainsi possible de détecter au plus vite d'éventuels problèmes afin d'y remédier dès leur apparition. Les nombreux plugins de Jenkins assurent le suivi de la qualité du projet, grâce aux rapports facile à lire ainsi qu'aux courbes et graphes générés.

Cette solution permet d'aider le LOA dans ses efforts pour renforcer la qualité de ces projets futurs et l'accès à ces résultats par l'ensemble des membres du projet permet de sensibiliser tous les collaborateurs à la qualité de leur code.

De plus, l'utilisation de Docker via Jenkins est possible, permettant de tester le code dans n'importe quel environnement choisi, et isolé du système. Il est ensuite facile grâce à son système d'images de compiler le code sur différentes machines sans avoir à se soucier des différentes installations de librairies nécessaires à son fonctionnement. Cette solution facilite ainsi la collaboration et l'échange de code du LOA avec les différents laboratoires et sociétés.

Afin de continuer à garantir la production d'un code de qualité croissante, l'utilisation d'autres plugins Jenkins associée à des librairies pourrait être envisagée afin de détecter par exemple des erreurs de compilation ou de syntaxe spécifiques à chaque langage.

5.2 Bilan personnel

Ce stage a été bénéfique pour moi, tant au niveau personnel que professionnel :

Au niveau personnel, j'ai appris de nouvelles choses et amélioré mes compétences au cours de ce stage. Je connaissais l'intégration continue et Jenkins uniquement de nom auparavant, j'ai maintenant une idée précise ce qu'est l'intégration continue ainsi que de son intérêt certain pour la gestion de projets informatiques. J'ai également découvert ce à quoi pouvait ressembler un serveur d'intégration continue via Jenkins, ainsi que les possibilités qu'il pouvait apporter. Pour essayer les différentes fonctionnalités de Jenkins sur des programmes basiques, j'ai utilisé Git. Bien que je l'avais déjà utilisé quelques fois auparavant, j'ai désormais une meilleure connaissance de ses possibilités ainsi que des différentes commandes principales grâce au terminal des systèmes Linux. Enfin, j'ai acquis des compétences sur Docker, qui m'était totalement inconnu jusque là.

Au niveau professionnel, j'ai pu avoir un aperçu sur le travail réalisé au travers d'un laboratoire de recherche. Par le biais de cette mission au caractère individuel, ce stage m'a également permis d'apprendre à travailler en autonomie, tout en discutant et fournissant régulièrement des retours vers mon tuteur sur mes avancées. Grâce à tout ce que j'ai appris, j'ai élargi mon spectre à d'autres compétences informatiques, ce qui peut être un atout si de futures missions autour de ce domaine me sont confiées à l'avenir.

Bibliographie

[1] Intégration continue, Juillet 2016. https://fr.wikipedia.org/wiki/Intégration_continue.

[2] What are the best continuous integration tools. https://www.slant.co/topics/799/~continuous-integration-tools.

[3] Jenkins (site officiel). https://jenkins.io/.

[4] BERG Alan Mark. *Jenkins Continuous Integration Cookbook - Second edition*. Packt Publishing Ltd., 2015.

[5] Teamcity (site officiel). https://www.jetbrains.com/teamcity/.

[6] Travis ci (site officiel). https://travis-ci.org/.

[7] Environnement (informatique), Février 2012. https://fr.wikipedia.org/wiki/Environnement_(informatique).

[8] Docker (site officiel). https://www.docker.com/.

[9] Plugin, Juin 2016. https://fr.wikipedia.org/wiki/Plugin.

[10] McFARLIN Tom. The beginner's guide to unit testing : What is unit testing ?, Juin 2012. http://code.tutsplus.com/articles/the-beginners-guide-to-unit-testing-what-is-unit-testing--wp-25728.

[11] GLICK Jesse. Junit plugin - jenkins wiki, Août 2016. https://wiki.jenkins-ci.org/display/JENKINS/JUnit+Plugin.

[12] BOISSINOT Gregory. Xunit plugin - jenkins wiki, Juin 2016. https://wiki.jenkins-ci.org/display/JENKINS/xUnit+Plugin.

[13] CONNOLLY Stephen. Cobertura plugin - jenkins wiki, Novembre 2015. https://wiki.jenkins-ci.org/display/JENKINS/Cobertura+Plugin.

[14] Duplication de code, Janvier 2016. https://fr.wikipedia.org/wiki/Duplication_de_code.

[15] HAFNER Ulli. Dry plugin - jenkins wiki, Juin 2016. https://wiki.jenkins-ci.org/display/JENKINS/DRY+Plugin.

[16] DE LOOF Nicolas. Cloudbees docker custom build environment plugin, Janvier 2016. https://wiki.jenkins-ci.org/display/JENKINS/CloudBees+Docker+Custom+Build+Environment+Plugin.

Annexe A

Exemple de Dockerfile

Exemple de Dockerfile qui compile un projet Python

```
FROM ubuntu
RUN apt-get update && apt-get install -y python \
                python-pytest \
                python-coverage \
                openjdk-8-jdk
ADD . /HelloPythonDocker
WORKDIR /HelloPythonDocker
CMD sh ./run.sh
```