Université Paris Dauphine

Mémoire de recherche

Septembre 2016

Améliorer la qualité des logiciels avec l'Intégration Continue

Gaëtan Meynier

Supervisé par

Khalid Belhajjame, Maître de conférence à l'Université Paris Dauphine,
Arthur Szumovicz, Consultant en système d'information à AXA.

Table des matières

Chapitre 1

Introduction

1.1 Motivation

1.2 Scope

1.3 Problèmes de recherche

1.4 Methodes

1.4.1 Méthodes de recherche

1.4.2 Collecte de données

1.5 Structure du mémoire

Chapitre 2

DevOps

Chapitre 3

Intégration Continue

3.1 Histoire

L'idée de l'Intégration Continue a été développée par la communauté Extreme Programming et a été décrit par Kent Beck dans son livre « Extreme programming explained » [Bec09]. Martin Fowler a également été l'un des premiers contributeurs qui ont écrit sur l'intégration continue [Fow00]. Plus tard, le travail a été poursuivi par Paul Duvall qui a écrit tout un livre à propos de l'intégration [Duv07].

3.2 L'Intégration Continue comme un processus

L'Intégration Continue est un processus où le logiciel est construit à chaque changement. Cela signifie que lorsqu'une modification apportée par un développeur est détectée dans le code source, une construction automatique est déclenchée sur une machine de construction séparé. La construction contient plusieurs étapes prédéfinies comme la compilation, les tests, la génération de métrique du code source et le déploiement - entre autres. Une fois la construction terminé un rapport est envoyé aux membres du projet spécifié. Le rapport de compilation indique le résultat de chaque étape de la construction avec des informations détaillées sur les erreurs possibles qui ont pu survenir.

Martin Fowler [Fow00] décrit l'Intégration Continue comme :

> *« Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that*

this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly. »

« L'Intégration Continue est une pratique de développement logiciels où les membres d'une équipe intègre leur travail régulièrement, chaque développeur intègre au moins quotidienne une version - conduisant à de multiples intégrations journalière. Chaque intégration est vérifiée par une construction automatique (y compris le test) pour détecter les erreurs d'intégration le plus rapidement possible. De nombreuses équipes trouvent que cette approche conduit à réduire considérablement les problèmes d'intégration et permet une équipe pour développer des logiciels cohésifs plus rapidement. »

3.2.1 Les bénéfices de l'Intégration Continue

Selon Paul Duvall [Duv07], l'intégration logiciels n'est pas un problème dans les petites équipe, un à deux développeurs, mais quand le nombre de collaborateur se multiplie ou que diverses équipes commencent à travailler ensemble sur un projet, l'intégration logiciels devient un problème, car plusieurs personnes modifient simultanément des morceaux de code devant fonctionner ensemble. Vérifier que les divers composants logiciels interdépendants continuent de fonctionner correctement ensemble soulève la nécessité d'intégrer plus tôt et plus souvent. Les points suivants décrivent les différents effets bénéfiques que Paul Duvall a été en mesure d'identifier.

Réduire les risques

En intégrant plusieurs fois par jour, les risques de dysfonctionnement sont considérablement réduits. Les problèmes sont remontés dès leur intégration et peuvent même être la cause d'un rejet d'intégration. Ceci étant possible par l'intégration et l'exécution de tests et l'inspections automatiquement du code source après chaque modification.

Générer des logiciels déployables

L'un des objectifs du développement logiciel agile est de déployer tôt et souvent. L'Intégration Continue aide à atteindre cet objectif en automatisant les étapes de production des logiciels déployables. Des logiciels déployables et fonctionnels est l'avantage le plus évident de l'Intégration Continue du point de vue extérieur, car le client ou l'utilisateur final est généralement pas intéressés par le fait que l'Intégration Continue ait été utilisé dans le cadre de l'assurance qualité. Il est aussi l'atout le plus tangible, étant le résultat final de l'Intégration Continue.

Permettre une meilleure visibilité du projet

Le fait que le processus d'Intégration Continue s'exécute régulièrement fournit la capacité à remarquer les tendances et à prendre des décisions sur la base d'informations réelles. Sans Intégration Continue, les informations doivent être recueillies manuellement requérant du temps et des efforts. Le processus d'Intégration Continue fourni en temps réel les informations sur l'état de la construction et de la qualité des dernières mesures tels que la couverture de test ou le nombre de codage violations de la convention.

Une plus grande confiance du produit

En ayant une Intégration Continue en place, l'équipe projet se protège contre certaines actions négatives portées aux codes sources. L'Intégration Continue agit comme un filet de sécurité, il repère les erreurs tôt et régulièrement. Ce qui se traduit par une plus grande confiance à l'équipe. Même des changements importants peuvent être faits avec confiance.

3.2.2 Le cycle de travail de l'Intégration Continue

Alors que l'Intégration Continue est généralement orchestrée par un serveur d'Intégration Continue, c'est aussi une façon de travailler. L'idée principale est d'intégrer les changements au code source aussi souvent qu'il y ait du nouveau code ajoutant de la valeur à l'application, à savoir plusieurs fois par jour. Cela rend l'intégration du code plus facile parce qu'il y a moins de code à intégrer à la fois. Il contribue également dans une situation où plusieurs personnes modifient la même base de code, car les développeurs auront ainsi les modifications apportées par d'autres. L'intégration des changements au code source de base comporte plusieurs étapes. Le cycle de travail est illustré dans le schéma suivant (Voir Figure 3.1). Il est basé sur les idées présentées par Martin Fowler [Fow00] avec une phase de mise à jour supplémentaire, après l'exécution des tests. Cette phase de mise à jour est nécessaire dans la situation où le test prend du temps de sorte qu'il pourrait déjà y avoir de nouvelles modifications intégrées pendant notre phase de test.

1. récupérer le code source du référentiel (Check-out),
2. modifier le code et créer les tests pour les modifications,
3. vérifiez si quelqu'un d'autre a modifié le code source,
4. exécuter tous les tests afin de vérifier que les modifications n'aient pas altéré le bon fonctionnement de l'application,
5. vérifiez à nouveau si quelqu'un d'autre ont modifié le code source,
6. valider les modifications dans le référentiel (Check-in),
7. démarrer un nouveau cycle.

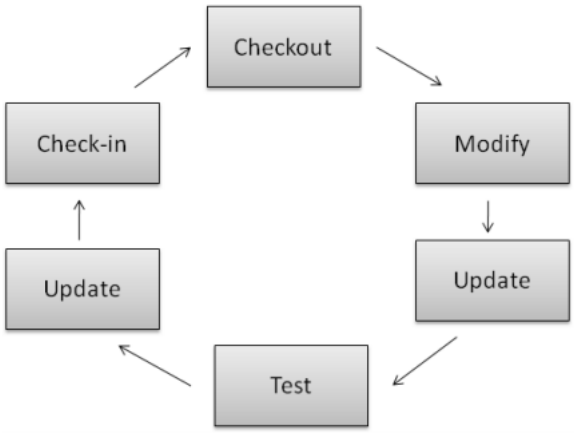

FIGURE 3.1 – Cycle de travail de l'Intégration Continue

Le cycle de travail commence créer une copie du code source hébergé sur le serveur de gestion de code qui est sur le point d'être modifié. Si le code avait précédemment récupéré, une mise à jour est faite à la place. Le développeur effectue les changements appropriés à sa tâche (nouvelle fonctionnalité, réfactorisation du code...) et implémente les tests appropriés afin de vérifier la conformité de son développement. Un test de développement vérifie que la sortie d'une méthode, en fonction de ses entrées, soit en adéquation avec le résultat attendu. Il est recommandé d'écrire les tests avant le code réel de l'application, ceci est appelé Test Driven Development (TDD).

Dans un projet de logiciel, il est courant que de nombreux développeurs travaillent simultanément sur une même partie de code. Par conséquent avant d'intégrer notre partie en code source, il est nécessaire de vérifier qu'aucune modification n'ait été faite par un développeur tierce sur par le biais d'une nouvelle mise à jour. Dans certains cas, il est possible que d'autres développeurs aient modifié exactement les mêmes lignes de code que nous produisant un ou plusieurs conflits. Dans ce cas, les conflits doivent être immédiatement résolus et tous les tests ré-effectués. Il est possible que dans cet intervalle une nouvelle version soit disponible sur le référentiel, l'étape précédente doit être répétée jusqu'à ce qu'aucuns nouveaux changements ne soient détectés. Une fois cette synchronisation effectué le check-in peut être fait.

3.2.3 Comment l'Intégration Continue s'appuie-t-elle sur d'autres pratiques de développement

L'Intégration Continue comprend un ensemble de règles qui tous développeurs devraient suivre :

1. effectuer régulièrement des commits,
2. ne pas commit des codes buggés,
3. régler les problèmes et builder immédiatement,
4. écrire des tests de développement automatisés,
5. tous les tests et métriques doivent être valides,
6. exécuter en local ses builds,
7. éviter de travailler avec du code buggés.

Ces règles ne sont pas nouvelles dans le monde du développement logiciel et sont aussi adopté d'autres pratiques de développement. Par conséquent, si des pratiques telles que les tests de développeur, les normes de codage, le refactoring et la propriété collective sont déjà mises en place au sein du projet, il est facile de commencer à utiliser l'Intégration Continue. Ces pratiques doivent être rigoureusement appliquées au travers de l'Intégration Continue sous peine d'empêcher les autres collaborateurs du projet de travailler. Prenons par exemple le cas d'un développeur ayant remanié une partie du code source et cassé quelques tests. Ne l'ayant pas remarqué, faute de n'avoir pas exécuté les tests, il valide ses changements dans le référentiel. La build éclate du faite que la règle « tous les tests doivent être au vert » n'est pas suivie. Maintenant si un autre développeur commence à travailler avec le code du référentiel, la première chose qu'il doit faire est de fixer ce qui avait été cassé par son collègue. Et cela peut prendre un beaucoup de temps, si cette personne ne connaît pas la partie de code qui provoque l'échec des tests.

3.3 L'Intégration Continue est un mélange de personnes et de systèmes

3.3.1 Le développeur

!!!!!!!!!!!!!!!! You build it, you run it !!!!!!!!!!!!!!!!

Pratiquer l'Intégration Continue exige de la discipline de la part des développeurs. Ils devront appliquer avec rigueur les pratiques de développement vus précédemment. Une fois le développement de la tâche effectué, le développeur doit exécuter une build sur sa propre machine de développement. On appelle cela une build privée. Cette étape permet de vérifier que les modifications apportées n'ont pas endommagé l'intégration avec le reste du code source. Il est important d'exécuter la build privée avant de valider les changements dans le référentiel de contrôle de version, car soumettre un code cassé peut empêcher les autres développeurs de travailler. Une fois l'exécution de la build privée avec succès, le développeur peut valider les modifications, et les tests. Si l'intégration de la build échoue malgré ses précautions, réparer cette build est la priorité numéro un.

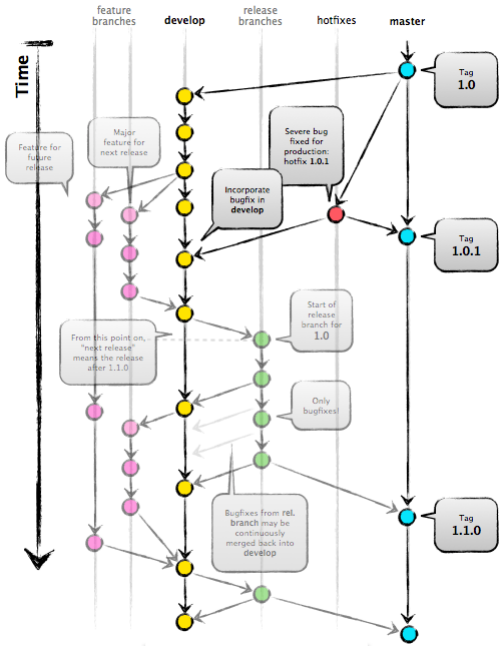

FIGURE 3.2 – Ligne de temps d'un référentiel de contrôle de version

3.3.2 Le référentiel de contrôle de version

L'intégration continue ne peut pas se faire sans référentiel de contrôle de version. Le référentiel de contrôle de version, également connu sous le nom de gestionnaire de code source (SCM), est un système utilisé pour stocker le code source et d'autres aspects du logiciel (comme la documentation) de manière centralisée. Il assure également le suivi de l'historique des versions et modifications effectuées au cours du développement. Les développeurs de logiciels ont la possibilité de revenir à une version antérieure ou la révision d'un logiciel, ou de prendre connaissances des changements apportés sur toute révision donnée. Ce référentiel fournit un point d'accès unique au code source pour les développeurs et le système d'Intégration Continue. Il peut être constitué de différentes branches du logiciel stocké. Une branche peut être créée pour la réécriture majeure d'un morceau de code ou pour le prototypage d'une idée intéressante qui pourrait ne pas se retrouver dans le produit final. La build d'intégration est exécuté sur la branche principale du référentiel de contrôle de version [Duv07]. « Master » est la branche de code source où la plupart de la mise au point a lieu. Certains systèmes de contrôle de version appellent cela aussi « tunk » ou « head ». La ligne principale se doit d'être toujours stable et la build d'intégration ne doit jamais échouée quand elle est intégrée au référentiel.

3.3.3 Le serveur d'Intégration Continue

Le serveur d'Intégration Continue est l'orchestrateur de l'ensemble du processus. Il exécute la build d'intégration lorsqu'une modification a été apportée au référentiel. Quatre approches sont à prendre en compte.

La première est la configuration d'un « post commit hook » au niveau du gestionnaire de code source. Le référentiel de contrôle de version peut-alors avertir immédiatement le serveur d'Intégration Continue qu'une modification a été ajoutée et validée. De cette façon une build d'intégration est exécutée pour chaque commit.

Une autre approche, dénommée « polling approach » [Duv07] est de vérifier les changements à intervalles réguliers (de l'ordre de la minute). De ce fait plusieurs changements peuvent être effectués entre chaque build.

Une troisième option est d'exécuter une build d'intégration à intervalles réguliers, mais si l'intervalle est trop long le bénéfice de la rétroaction est rapidement perdu.

La quatrième et dernière option, consiste à intégrer une copie du référentiel principal, accessible uniquement par le serveur d'Intégration Continue, au niveau du serveur lui-même 3.3. Les développeurs n'ont ainsi accès qu'au référentiel du serveur d'Intégration Continue. Ce dernier peut alors être configuré afin de rejeter les modifications apporté au référentiel ne respectant pas les tests ou les métriques qualités prédéfinies. Garantissant ainsi la qualité de l'application.

Le serveur d'Intégration Continue fournit également une vue, généralement une page web, qui expose l'état de santé de tous les « build jobs »[1] et afficher leurs résultats en temps réel configurés (Voir Figure 3.4). Ce tableau de bord peut être affiché, par exemple, sur un grand écran dans la salle de l'équipe de développement pour donner un aperçu rapide et en temps réel des tâches effectués sur le serveur et voir ainsi s'il y a des constructions en cours d'exécution. De nombreux serveurs d'Intégration Continue proposent également un type de visualisation basé sur les principes des feux de circulation ou de météo de connaitre l'état des builds d'Intégration.

Toutes les fonctionnalités d'un serveur d'Intégration Continue ne sont pas nécessaires pour faire de l'Intégration Continue. De nombreux scripts personnalisés peuvent effectuer les mêmes tâches, mais avoir un serveur conçu à cet effet aide beaucoup [Duv07]. De plus en plus de solutions propriétaires ou open source[2] performe le marché et offre un environnement d'Intégration Continue

1. Build jobs : cela correspond aux différentes tâches du processus de build.
2. Open source : logiciel libre redistribution, d'accès au code source et de création de

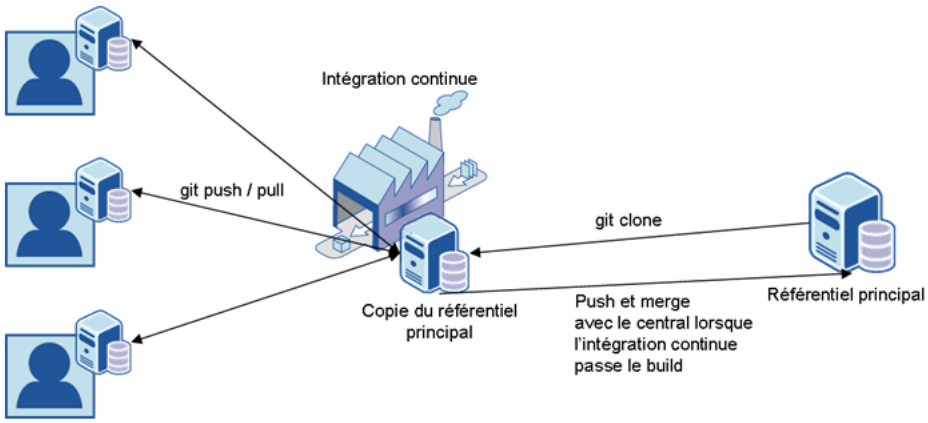

FIGURE 3.3 – Organisation d'un serveur d'Intégration Continue

stable et complet.

Dans sa forme la plus simple, l'Intégration Continue pourrait être mise en place avec un seul ordinateur dédié exécutant des scripts afin de vérifier le code source du référentiel, lancer une build d'Intégration et envoyer des rapports une fois la build terminée. Le serveur d'Intégration Continue offre une autre possibilité en fournissant une interface utilisateur afin de configurer les multiples « build jobs ».

3.3.4 Les scripts de construction (build)

Généralement la plupart des étapes d'une build sont définies en utilisant un script de compilation. Un script de compilation peut être constitué d'un ou plusieurs scripts et il est utilisé, par exemple, pour compiler, tester, contrôler et déployer des logiciels. Toutes les mesures qui peuvent être automatisées pour construire et déployer des logiciels doivent être automatisées. Cela économise du temps et les nerfs des développeurs. Il existe de nombreuses techniques disponibles comme Ant (Java), Make (C/C++) ou Scons (Python).

Certains développeurs utilisent leur environnement de développement intégré (IDE) pour builder leurs logiciels. Dans ce cas la build ne pourra être encadré par l'Intégration Continue. L'intégration Continue nécessite que la build puisse être exécutée indépendamment de tout IDE [Duv07].

travaux dérivés.

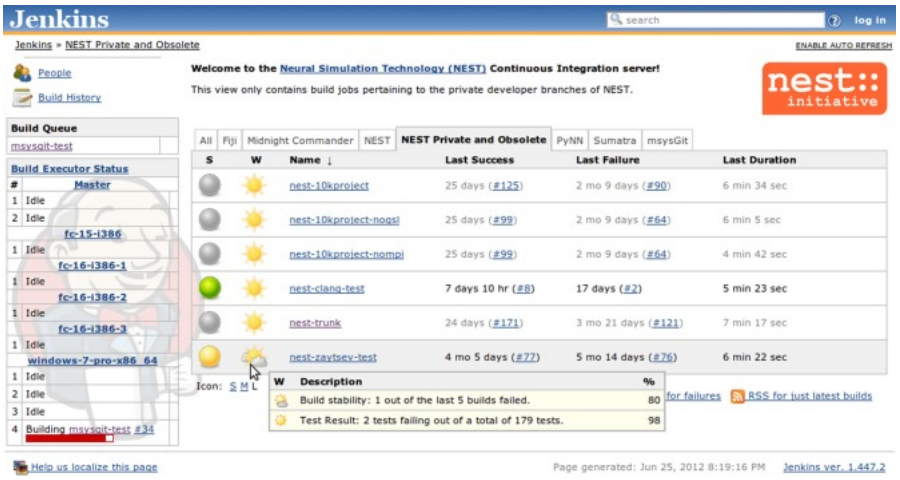

FIGURE 3.4 – Tableau de bord d'un serveur d'Intégration Continue Jenkins

3.3.5 Les mécanismes de feedback

Lorsque la build d'intégration est terminée, les résultats doivent être accessibles dès que possible. La capacité à fournir un feedback rapide est l'un des avantages de l'Intégration Continue. La rétroaction est disponible immédiatement une fois la build terminée. La rétroaction peut être diffusée par différent canaux ; tableau de bord, courrier électronique, flux RSS. En cas de build défectueuse, la réparation peut démarrer immédiatement après réception de l'avis.

Certain pionnier commence même à intégrer le terme de « monitoring continue » dans l'ingénierie logicielle en complément de l'Intégration Continue. Le monitoring continu consiste à avoir un affichage visible par tous les membres de l'équipe de développement, actualisé en temps réel, donnant un feedback direct sur l'état des différentes builds simplifié et directement interprétable. Cet affichage est dans la plupart du temps un moniteur, mais d'autres solutions plus amusantes, telles que la lampe à lave ou une « ambient orb » commencent à s'imposer [Swa04].

3.3.6 Les machines de build d'intégration

Un serveur d'Intégration Continue à besoin d'un hôte pour fonctionner. La machine de build d'intégration (ou nœud) est une machine distincte qui doit imiter l'environnement de production. Si possible, elle doit fonctionner avec le même système d'exploitation, la même version de serveur de base de données et les mêmes versions de librairies doivent être utilisées, comme il est prévu pour en production. Chaque différence augmente le risque des tests de ne pas détecter les problèmes liés à l'environnement de la production [Fow06].

Dans le cas où de nombreux « build jobs » sont configurés pour l'application, pour réduire la durée de la build et ainsi augmenté la rapidité de la rétroaction il peut être nécessaire de paralléliser les tâches sur plusieurs machines (scalabilité horizontal). Certains logiciels de serveur d'Intégration Continue fournissent une architecture maître-esclave qui permet ainsi de diviser la charge de travail sur plusieurs hôtes.

Parfois plusieurs environnements sont nécessaires pour builder sur différentes plates-formes. La virtualisation des serveurs apportent la réponse à ce problème. En utilisant une infrastructure de virtualisation bien établis, il est assez facile d'exécuter des instances esclaves multiples sur un seul nœud physique. Ces instances pouvant fonctionner sous différents systèmes d'exploitation. Certains logiciels de serveur d'Intégration Continue offre la possibilité à la build d'intégration de fonctionner simultanément sur ces différentes instances et de collecter les résultats pour chaque environnement.

Puis virtualisation est quelque chose de valeur à l'aide de la peine de la virtualisation diminue [Bar03]. Certains logiciels de serveur de CI comme Hudson permet même de construction de travail afin de fonctionner simultanément sur plusieurs serveurs virtuels et de collecter des résultats pour chaque environnement.

3.4 Caractéristique de l'Intégration Continue

Selon Paul Duvall [Duv07] seules quatre caractéristiques sont nécessaires à l'Intégration Continue :

— une connexion à un référentiel de contrôle de version,
— un script de compilation,
— un mécanisme de rétroaction,
— un processus pour intégrer les modifications au code source (manuelles ou serveur d'Intégration Continue).

Ces éléments seuls sont nécessaires à la construction d'un système d'Intégration Continue efficace, qui est expliqué plus en détail dans les sections suivantes.

3.4.1 Compilation du code source

La compilation du code source est l'une des caractéristiques de base du système d'Intégration Continue. La compilation crée des exécutables binaires à partir de source lisible (pour les développeurs). Lors de l'utilisation des langages dynamiques comme Python ou Ruby la compilation est différente. Les binaires ne sont pas compilés, à la place les développeurs ont la possibilité d'effectuer

```
testchoice (__main__.TestSequenceFunctions) ... ok
testsample (__main__.TestSequenceFunctions) ... ok
testshuffle (__main__.TestSequenceFunctions) ... ok

----------------------------------------------------------
Ran 3 tests in 0.110s

OK
```

FIGURE 3.5 – xUnit output

un checking strict, qui peut être considéré comme de la compilation dans le contexte de ces langues [Duv07].

3.4.2 Tests

Les tests sont la partie la plus vitale de l'Intégration Continue. Beaucoup considèrent qu'une Intégration Continue sans automatisation du contrôle continu ne peut être un Contrôle Continue [Duv07]. Il est difficile d'avoir confiance dans les changements du code source sans une bonne couverture de test. Les tests peuvent être automatisés en utilisant des outils de tests unitaires tels que JUnit (Java), NUnit (C#), ou d'autres framework [3] de xUnit. Certains de ces frameworks peuvent également générer des rapports machines lisibles, qui peuvent être analysés et utilisés pour générer des représentations graphiques telles que des pages Web ou des tableaux (Voir Figure 3.5).

Niveaux de tests

Le test peut être effectué à différents niveaux 3.6. Le plus bas niveau de test est appelé test unitaire. Une unité est la plus petite partie d'une application testable. Cela correspond une fonction ou une méthode dans une classe. Le but des tests unitaires est de vérifier que les différentes parties du code fonctionnent comme elles le devraient. Ils assurent la stabilité du code en testant chaque unité unitairement. Les régressions seront ainsi remontées très rapidement ce qui permet de manipuler le code source avec confiance. Les tests unitaires sont généralement écrits par le développeur qui a également écrit le code. Une bonne pratique des tests unitaires est de commencer par écrire les tests et d'ensuite les valider par le code. C'est ce qu'on appelle le « Tests Driven Development » (TDD) ou Développement Dirigé par les Tests ».

Le niveau suivant est le test d'intégration. Dans ce contexte d'Intégration nous devons vérifier que les modules individuels du logiciel fonctionnent aussi en tant que groupe.

3. Framework : ensemble d'outils et de composants logiciles.

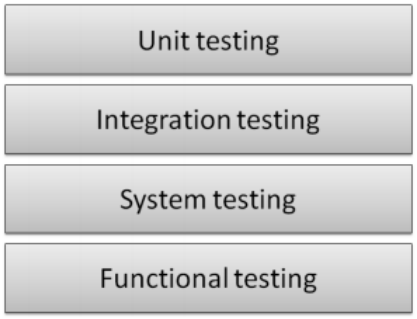

FIGURE 3.6 – Les principaux niveaux de test

> *« Le test d'intégration identifie les problèmes qui se produisent lors de la combinaison d'unités. En utilisant un plan de test exigeant que vous testiez chaque unité et que vous vérifiez la viabilité de chacune d'elles avant de les combiner, vous savez que les erreurs découvertes lors de la combinaison d'unités concernent probablement l'interface entre les unités. Cette méthode réduit le nombre de possibilités à un niveau beaucoup plus simple d'analyse. »* [Mic16]

Le troisième niveau teste les API d'un point de vue externe, sans se préoccuper de fonctionnement interne du système. Le test système analyse le flux de retour de l'API en fonction de son flux d'entrée afin de détecter les défauts à la fois dans les inter-assemblages mais également au sein du système dans son ensemble. Cette méthode est appelé « boîte noire ».

Le test fonctionnel est le quatrième niveau de test majeur. Il assure la stabilité de l'application en reproduisant le parcours d'un utilisateur sur le navigateur. Il teste le bon fonctionnement de l'application et remontent les régressions fonctionnelles.

D'autres niveaux de test existent, tel que le test applicatif qui assure la sécurité et la compatibilité, le test d'IHM qui fiabilise l'ergonomie et la visibilité, le test de charge qui veille à la performance et à la robustesse de l'application...

Exécuter les tests plus rapides en première

Ecrire des tests d'échecs

3.4.3 Base de données d'intégration

3.4.4 Inspection Continue

Différences entre inspection et test

Rapport d'inspection

3.5 Déploiement Continue

3.5.1 Les bonnes pratiques du Déploiement Continue

Etiqueter les actifs d'un référentiel

Produire un environnement propre

Etiqueter chaque build

Exécuter tous les tests

Créer des build feedback reports

Capacité à effectuer un rollback

3.5.2 La « bêta perpétuelle »

Le principe de la « bêta perpétuelle » a été introduit pour la première fois par Tim O'Reilly dans le manifeste du Web 2.0, qui expose le principe selon lequel :

> *« Les utilisateurs doivent être considérés comme des co-développeurs, en suivant les principes Open Source (. . .). Avec le Web 2.0, ce principe évolue vers une position plus radicale, la bêta perpétuelle, dans laquelle le produit est développé de manière ouverte, avec de nouvelles fonctionnalités offertes sur une base mensuelle, hebdomadaire, ou même quotidienne. »*

Le terme « bêta perpétuelle » désigne le fait que notre application n'est jamais finalisée. Elle s'absout des contraintes habituelles liées au cycle de développement en « release » au profit d'une livraison en continu des nouvelles fonctionnalités.

Release early, release often

Derrière ce nouveau concept se cache un concept déjà bien en place chez les agilistes (du point de vue de l'itération courte) et dans le monde de l'Open Source (du point de vue de la récolte continue du feedback), le « Release early,

release often », traduit en français par « Publiez tôt, publiez souvent ». Cette pratique a été décrite par Eric Steven Raymond dans “La cathédrale et le bazar” où il formulait explicitement :

« Publiez tôt. Publiez souvent. Et écoutez vos clients »

Cette méthodologie vise à réduire les temps de développement et améliorer l'implication de l'utilisateur dans la conception du logiciel afin de créer un produit correspondant à ces attentes et ainsi éviter la création d'un logiciel que personne n'utilisera.

Les services en ligne (Software As A Service)

Le concept de « bêta perpétuelle » a été rendu possible grâce au Cloud Computing et plus particulièrement à la généralisation du service en ligne ou « Software As A Service ». L'hébergement de l'application par l'éditeur permet d'absoudre ce dernier au traditionnel cycle de déploiement d'un logiciel et de ne gérer qu'une seule version de son application. Les services en ligne sont continuellement mise à jour sans pour autant en informer l'utilisateur. Les nouvelles fonctionnalités, découverte au fur et à mesure par l'utilisateur, permettent un apprentissage progressif des nouveautés applicatives.

La « Customer driven roadmap »

L'hébergement de l'application sur serveur offre à l'éditeur une maîtrise total de sa plateforme de production. Il peut ainsi mettre en place des sondes analytiques afin de récolter des informations sur l'usage de notre application et l'accueil réservé à nos nouvelles fonctionnalités par l'utilisateur.

Les pré-requis

La mise en place d'une stratégie de « bêta perpétuelle » requiert certains pré-requis pour en garantir le succès :

- une intégration continu,
- une livraison continue,
- un déploiement continu,
- une stratégie de type « One click deployment / Rollback » pour une restauration rapide de l'application au dernier état stable.

Conclusion

Le concept de la « bêta perpétuelle » est présent chez de nombreux géants du Web tel que Google, Facebook, Amazon, Twitter, Flickr... Peu en font mention dû à la mauvaise image du terme « bêta », qui pour la conscience collective, se

réfère à un produit non fini et peu fiable. Prenons en exemple Gmail, la boite aux lettres mails développé par Google, qui jusqu'en 2009 intégrait la mention « bêta » dans son logo. De petites fonctionnalités unitaires sont fréquemment proposés aux utilisateurs. En fonction de leur niveau d'adoption Google les intègrent ou non à la version standard de son service.

3.5.3 Le « Zero downtime deployment »

Les patterns

Blue/Green Deployment

Canary Release

Dark Launch

La mise en oeuvre

Load balancer

Les sessions

La modification du schéma de base de données

3.5.4 Le « Flipping feature »

Chapitre 4

Etude de l'art

4.1 Logiciel en développement

4.2 Processus de développement logiciel

4.2.1 Scrum dans l'AgilLab

4.2.2 Comment l'Intégration Continue s'intègre-t-elle dans le processus de développement logiciel

4.2.3 Les équipes de développement

4.3 La sécurité à l'esprit

Chapitre 5

Implémentation

5.1 Choisir ses outils

5.1.1 Scaling

5.1.2 Le choix du serveur d'Intégration Continue

Support du langage de programmation

Support du Source Code Management

Gestion des builds

Sécurité

Autres caractéristiques

Extensibilité

Installation et configuration

Autres questions à examiner

5.2 Logiciels et outils utilisés

5.2.1 Serveur d'Intégration Continue

5.2.2 Logiciel de gestion de configuration

5.2.3 Outils de build

5.2.4 Automatisation des tests et de la couverture de code

5.2.5 Documentation Continue

5.2.6 Déploiement Continue

5.2.7 Feedback Continue

5.3 Architecture

5.3.1 Serveur d'Intégration Continue

5.3.2 Logiciel de gestion de configuration

5.3.3 Réseaux et déploiement

5.4 Configuration

Table des figures

Bibliographie

[Bar03] Paul Barham. *Xen and the Art of Virtualization.* ACM SIGOPS Operating Systems Review, 164-177, 2003.

[Bec09] Cynthia Andres Kent Beck. *Extreme programming explained.* Addison Wesley, 2009.

[Duv07] Paul M. Duvall. *Continuous Integration : Improving Software Quality and Reducing Risk.* Addison Wesley, 2007.

[Fow00] Martin Fowler. Continuous integration, 2000. http ://www.martinfowler.com/articles/originalContinuousIntegration.html.

[Fow06] Martin Fowler. Continuous integration, 2006. http ://www.martinfowler.com/articles/continuousIntegration.html.

[Mic16] Microsoft. Test d'intégration, 2016. https ://msdn.microsoft.com/fr-fr/library/aa292128(v=vs.71).aspx.

[Swa04] Michael Swanson. Automated continous integration and the ambient orb, 2004. http ://blogs.msdn.com/b/mswanson/archive/2004/06/29/169058.aspx.