Zeste de savoir

La méthode Continuous Delivery

12 août 2019

Table des matières

? Je viens d'avoir une idée géniale , mais comment réussir à la rendre visible aux utilisateurs le plus vite possible ?

C'est une question que bon nombre de développeurs se sont souvent posés et à laquelle **Jez Humble** and **David Farley**, apportent une réponse en deux mots (qui a fait l'objet d'un ouvrage) : *Continuous Delivery*

FIGURE 0. – Livraison continue

Le *Continuous Delivery*, selon Martin Fowler, est une discipline du génie logiciel où vous construisez une application de telle manière qu'elle puisse être envoyée en production à tout moment. Cette façon de travailler de manière agile est très prisée par le mouvement DevOps dont la philosophie est : *« You build it, you run it »*.

Vous l'aurez compris, l'enjeu principal de la méthode *Continous Delivery* est de réduire le plus possible le temps entre une idée et sa mise à disposition. Nous verrons donc dans la suite, les concepts principaux de la méthode ainsi que les bonnes pratiques de mise en place. Vous découvrirez ainsi qu'au-delà du défi technique, la méthode relève aussi et surtout un défi social.

1. Les concepts clés de la méthode

1.1. La gestion de la configuration

Pour dire qu'on gère correctement la configuration de l'application il faut pouvoir répondre positivement aux questions suivantes :

— Est-ce que je peux reproduire sur n'importe quel environnement de déploiement mon application (même niveau de patch, même configuration, même dépendances logicielles, etc.) ?
— Est-ce que je peux facilement faire un changement dans mon application ou sa configuration et déployer le changement sur mes environnements ?
— Est-ce que je peux facilement voir les différences de configuration entre mes environnements (développement, recette, pré-production, production), avec une trace de qui et quand ont été fait les changements ?

1.1.1. Le versioning

En *Continuous Delivery*, tout doit être versionné. On doit pouvoir à n'importe quel moment charger une version précise de l'application développée. Pour ce faire, de nombreux gestionnaires de versions existent. Parmi les plus populaires nous avons Git , SVN et Mercurial .

La plupart du temps, quand on parle de *versioning* de l'application, on pense uniquement à versionner le code source. D'autres éléments doivent aussi être versionnés tel que les dépendances, l'environnement sur lequel est déployé l'application, ou la documentation associée.

1.1.2. La gestion de dépendances

De nos jours, plus personne ne réinvente la roue, on se base de plus en plus sur des bibliothèques existantes pour travailler et donc se concentrer sur notre cœur de métier. Cependant, les bibliothèques tierces ont elles aussi leur cycle de vie qu'il faut prendre en compte. Il serait impensable de stocker les exécutables des bibliothèques tierces dans les sources du code (pour éviter de l'alourdir).

Pour gérer ces problématiques, il existe des gestionnaires de dépendances pour langages de programmation. Parmi les plus connus, on a pip pour python, npm pour Javascript, Maven ou Gradle pour Java/Groovy, RubyGems pour ruby, Composer pour php, etc. Ils permettent tous de déclarer un ensemble des dépendances tierces de l'application dans le code source. Les bibliothèques tierces seront téléchargés et installés au moment du *build* de l'application.

1.1.3. La gestion de l'environnement

Il est important de distinguer une application de l'environnement dans lequel il doit être déployé. Les sources d'une application sont généralement constituées du code qui sera transformé en exécutable, alors que les sources d'un environnement décrivent les dépendances plus bas niveau.

Si on prend l'exemple de l'application ZdS , réalisée en python/django. L'environnement prêt à l'accueillir nécessite les dépendances plus bas niveau que sont, un serveur web (Apache, Nginx, etc.), une base de données (MySQL, PostgreSQL, Oracle, etc.), un moteur d'indexation (Solr, Elasticsearch, etc.). Toutes ces dépendances, ainsi que les valeurs associées (adresse du serveur web, nom de la base de données, ports de connexion, etc.) doivent êtres versionnés pour chaque environnement (développement, recette, pré-production, production).

Il existe des outils très connus pour gérer les configurations d'environnements. On retrouve parmi les plus gros : Puppet , Chef , Capistrano , Ansible . Ils permettent, lorsque vous recevez une machine vierge, d'installer un environnement de travail particulier sur lequel sera ensuite déployé l'application. L'intérêt est donc de pouvoir répéter l'action sur n'importe quel machine.

1.2. Le Continuous Provisioning

C'est une étape importante, mais pas forcément nécessaire du *Continuous Delivery*. L'idée est d'avoir dans le circuit *Continuous* un outil de *provisioning* qui permet de créer une machine vierge de travail sur laquelle vous allez installer votre environnement. Dans la plupart des cas, le *provisioning* consiste à créer une nouvelle VM rattachée à une adresse IP, ainsi que les compte et profils de base.

Le monde du *provisioning* qui était trusté jusqu'ici par des grands tel que Vagrant, ou encore VM Ware, se voit aujourd'hui presque révolutionné par Docker . Les Startups et même Google, s'accordent à dire que Docker est l'avenir du *provisioning*.

1.3. Le Continuous Integration

Figure 1. – Cycle d'intégration

L'intégration continue est une pratique en génie logiciel, qui consiste à faire en sorte que les développeurs intègrent fréquemment leurs travaux à l'application. Chaque intégration doit être testée de manière automatique afin de détecter les erreurs le plus rapidement possible. Si les tests échouent, le travail du développeur n'a pas à être intégré, d'où l'importance d'avoir un taux de tests automatiques très élevé dans votre code.

L'environnement d'intégration se rapproche de l'environnement de production. Si vous utilisez par exemple une base de données Oracle en production et que vous souhaitez installer un SGBD plus léger pour les développeurs, votre environnement d'intégration devra utiliser le SGBD cible de production qui est Oracle.

Autrement dit, en intégration continue, si nous devons développer une fonctionnalité, les étapes suivantes sont nécessaires :

— Se positionner sur la version à partir de laquelle on souhaite travailler (utilisation du gestionnaire de version)
— Développer la fonctionnalité voulue
— Ajouter et/ou modifier des tests automatiques qui vérifient que la fonctionnalité marche comme prévu
— Compiler l'application et dérouler l'ensemble des tests sur notre environnement de développement
— Si les tests sont positifs en environnement de développement, les envoyer sur l'environnement d'intégration
— L'environnement d'intégration (qui se rapproche de celui de production) va compiler à nouveau les sources et exécuter la batterie de tests
— Si les tests sont positifs en environnement d'intégration, nos modifications peuvent être *mergées* dans le référentiel principal.

1.4. Le Continuous Deployment

Le déploiement continu consiste à déployer automatiquement les changements survenus dans le code. Le déploiement peut être complètement automatique (dès que le commit est *mergé*) ou en un clic (nécessite l'intervention humaine). Pour mettre en œuvre cette étape ultime dans la démarche *Continuous*, il faut au préalable s'assurer de plusieurs choses :

— Toutes les autres étapes sont mises en œuvre
— Le taux de couverture des tests doit être très bon
— Un déploiement doit être réversible (*rollback*)

Le monde du génie logiciel a vu apparaître de nombreux outils d'automatisation du déploiement. On peut citer par exemple XL Deploy ou encore Kwatee qui permettent de déployer, via un simple clic, votre application sur l'environnement que vous avez défini.

http://zestedesavoir.com/media/galleries/1393/

2. Les bonnes pratiques Continuous

2.1. Maintenir un unique référentiel pour les sources

Il est important de choisir le gestionnaire de source le plus adapté à vos besoins et s'assurer que l'ensemble des développeurs connaissent l'outil (car une erreur est très vite arrivée).

Si l'utilisation d'un gestionnaire de source est devenue un standard de nos jours, nombreux sont des projets qui ne versionnent pas **tout**. Des scripts d'installation aux fichiers de configuration, en passant par les scripts de chargement de la base de données, scripts de tests, bibliothèques tierces, etc. Tout doit être dans le gestionnaire de sources. L'objectif étant de pouvoir, à partir d'une machine vierge, déployer une version précise de l'application.

2.2. Automatiser le build

Obtenir l'exécutable d'une application à partir de ses sources peut facilement devenir complexe. Demander aux développeurs de taper une dizaine de commandes pour compiler l'application peut rapidement devenir source d'erreurs/oublis. Dans la plupart des cas, ces tâches peuvent être automatisées et donc, **doivent être automatisées**.

Les outils qui vous permettent de *builder* votre application sont nombreux. On en trouve au moins un pour chaque langage de programmation. Parmi les plus gros, on a Ant , Maven ou Gradle pour Java, Rake pour Ruby, CMake pour C/C++, Grunt ou Gulp pour JavaScript, Sphinx pour RsT, etc.

2.3. Des tests et encore des tests, le médicament contre le stress

Builder une application revient à charger les dépendances tierces, compiler, *linker*, etc. Cependant, *builder* ne vous assure pas que votre application fonctionne et est exempt de bugs. Une bonne pratique est donc d'inclure dans le processus de *build* de votre application des tests automatiques.

Tester une fonction, une procédure ou un module revient à dire que si l'on a telles ou telles valeurs en entrée, on devrait obtenir telles ou telles valeurs en sortie. On distingue plusieurs niveaux de tests :

— Les tests unitaires : chargés de vérifier qu'un bout de code fonctionne comme prévu par le développeur
— Les tests d'intégration : chargés de vérifier qu'un bout de code s'intègre bien avec le reste du code
— Les tests fonctionnels : chargés de vérifier que l'application dans son ensemble répond aux besoins spécifiés par les utilisateurs
— La recette : souvent réalisée par une partie des futurs utilisateurs et à la main, elle permet de s'assurer qu'effectivement l'objectif est atteint.

Les tests unitaires et d'intégrations doivent être automatisés. Aujourd'hui, il existe des outils pour ce faire (Junit en Java, Unittest en python, etc.). Quant aux tests fonctionnels, de nouveaux concepts tels que le TDD et le BDD ont vu le jour pour pouvoir les automatiser au maximum.

Vous ne pourrez jamais vous reposer entièrement sur les tests automatiques, mais dans la majorité des cas, ils permettent d'éviter de nombreuses régressions et font gagner du temps aux testeurs finaux.

Dans certains cas, on peut avoir besoin de mettre en place des tests de charges.

2.4. Builder chaque commit sur l'environnement d'intégration

Chaque fois qu'un *commit* est fait, le serveur doit vérifier que l'application *builde* et passe l'ensemble des tests sur l'environnement d'intégration. Le serveur d'intégration doit, dans la mesure du possible, notifier l'auteur du *commit* de l'état (succès ou échec) de son code.

Il existe bon nombre de serveurs d'intégration disponibles dans le *cloud* qui émulent des systèmes de type Linux (Travis , CircleCI , Shippable , etc.) ou encore de type Windows (Appveyor). Il suffit en général de leur transmettre les instructions de *build* et de lancement des tests pour qu'ils le fassent pour vous. Il est aussi possible d'héberger votre propre serveur d'intégration chez vous (avec Jenkins par exemple).

> Le *build* doit être rapide : une application qui met plus d'une demi-heure à *builder* réduit considérablement l'intérêt de la démarche *Continuous*. Il faut tout faire pour réduire le temps de *build* au minimum. Chaque minute gagnée sur le temps de *build* est une minute gagnée pour chaque développeur (et de l'argent économisé pour le boss).

2.5. S'assurer que tout le monde peut récupérer facilement le dernier exécutable

Parce qu'il est souvent difficile de spécifier à l'avance de manière correcte le besoin, nombreux sont ceux qui ne savent ce qu'ils veulent qu'une fois qu'ils ont quelque chose à voir : une base de travail.

Pour prévenir ce genre de problème, toute personne impliquée dans le projet devrait avoir accès à la dernière version de l'application prête à l'emploi.

2.6. Indicateurs, métriques et tableaux de bord pour tous

Un projet est rarement développé tout seul, et dans la plupart des cas, nombreux sont les acteurs qui y participent. Afin d'avoir une bonne vision de l'évolution du projet, il est indispensable d'avoir des indicateurs de suivi.

Tous les indicateurs n'intéressent pas tous les acteurs du projet. Il faut des tableaux de bord à plusieurs facettes. Connaitre le taux de couverture des tests (jusqu'à quel pourcentage peut-on faire confiance aux tests) intéressera certainement le directeur technique. Connaitre les apports de la dernière *release* intéressera surtout les testeurs, etc. Tous ces indicateurs doivent être produits de manières automatiques et disponibles tout le temps.

FIGURE 2. – KPI : Key Performance Indicator

2.7. Automatiser le déploiement … et la marche arrière

Dans une démarche *Continuous*, vous constaterez que l'on a besoin de déployer sur plusieurs environnements et souvent plusieurs fois par jour. Il faut donc pouvoir automatiser le déploiement pour ne pas mobiliser une ressource uniquement pour ça. Le script de déploiement doit permettre de déployer sur n'importe quel environnement cible. Vous pourrez même utiliser le même script pour déployer en production.

Qui dit déploiement automatique, dit procédure de *rollback* automatique. Car si vous avez besoin de revenir à un état antérieur, vous devez pouvoir y arriver. Une procédure de *rollback* automatique réduit souvent pas mal de tension lors du déploiement et encourage donc à déployer plus souvent et plus sereinement.

3. Le Continuous Delivery, qu'en penser ?

Mettre en place une démarche *Continuous* n'est pas aussi difficile que ça peut en avoir l'air, surtout sur un projet tout neuf. La principale difficulté réside dans la rigueur et la complétude des tests.

Le *Continuous Delivery* offre naturellement de nombreux avantages :

— **La réduction du risque au déploiement** : on déploie de petites modifications, il y a moins de chance de se tromper et il est plus simple de corriger un problème qui apparaît. Et si jamais on découvre un bug, on sait toujours revenir en arrière.
— **La crédibilité des mises à jour** : beaucoup d'utilisateurs jugent les progrès sur pièce. Si «**fait**» signifie «les développeurs déclarent que c'est fait», c'est beaucoup moins crédible que si c'est déployé en préproduction.
— **Les retours utilisateurs** : plus vite les utilisateurs sont exposés aux mises à jour, plus tôt vous avez des retours sur les fonctionnalités et plus vite vous saurez quels ajustements mettre en œuvre. Attention tout de même, les changements doivent être exposés d'abord à une population pilote, les *early adopters*.

La démarche *Continuous Delivery* permet donc de responsabiliser au mieux tous les acteurs de la chaîne de développement (développeurs, testeurs, administrateurs systèmes, administrateurs de base de données, etc.) et d'améliorer la relation entre elles. Tout ceci a pour conséquence directe d'encourager la prise d'initiative au sein de l'équipe.

Liste des abréviations

BDD Behavior Driven Developement. 5

SGBD Système de Gestion de Base de Données. 3

TDD Test Driven Developement. 5

VM Machine Virtuelle. 3