DALL'AGNOL Tom
BUISSON Kévin
ROLLIN Antoine

09/04/2016
Groupe 1 - SI4
DevOps

Mr. StataCookie

L'intégration continue avec Artifactory & Jenkins

L'objectif du TD est de transformer l'architecture monolithique du projet The Cookie Factory en une architecture modulaire de façon à rendre sa modification plus flexible et de mettre en place un pipeline d'intégration continue.

I. Intégration continue

1. Qu'est-ce que c'est ?

L'intégration continue est un ensemble de pratiques de génie logiciel permettant d'assurer le bon fonctionnement du code produit, et ce, de manière automatique.

Dans le cadre de Mr.StataCookie nous avons employé les pratiques suivantes :

- compilation automatique via Maven
- test automatique du code grâce à Maven, pour s'assurer que notre code fonctionne comme il le devrait
- découpage du code en modules
- compilation et test du code, automatique lors d'un push, grâce à Jenkins, pour s'assurer de la validité du code avant le déploiement
- déploiement de/des artefacts correspondant au code push
- récupération d'artefacts fonctionnels pour les dépendances de notre module grâce à Maven et Artifactory pour que les autres développeurs aient toujours un build fonctionnel des différents modules
- tests d'intégration à l'aide d'Arquillian et de Jenkins pour vérifier que nos modules fonctionnent bien ensemble

2. Intérêts

Les intérêts de l'utilisation de l'intégration continue sont nombreux.

Le découpage en module du code permet à différentes équipes de travailler sur une partie du projet, et le fait de déployer des artefacts toujours fonctionnels permet à ces différentes équipes de ne pas se gêner entre elles. Lorsque le code push ne fonctionne pas, les autres équipes récupèrent un artefact fonctionnel : le problème ne se propage pas.

Tester unitairement permet de s'assurer que les classes que l'on modifie n'aient pas de comportement aberrant et qu'elles font bien leur travail. Tester l'intégration permet de s'assurer que les modules communiquent correctement entre eux et n'aient pas de comportement insensé entre eux.

La compilation et les tests automatiques permettent de gagner du temps puisque cela n'est plus à faire manuellement. Ainsi, l'intégration implique un gain de temps, tout en s'assurant que le code que nous produisons est correct.

3. Mise en place

Pour mettre en place l'intégration continue en place dans notre projet nous avons tout d'abord employé Maven pour compiler et lancer les tests de manière automatique. Nous avons ensuite découpé le code en module (pour voir le découpage, veuillez-vous référer au document annexe), puis mis en place un repository manager (Artifactory) pour stocker les artefacts de chacun de nos modules et enfin un outil d'intégration continue (Jenkins) qui permet d'automatiser les tâches de compilation, lancement de test et de déploiement (si tout est correct) vers le repository manager.

Artifactory est un repository manager : il nous permettra de stocker le build de chacun des modules et de les mettre à disposition de Jenkins et des utilisateurs ayant besoin des dépendances manquantes. La configuration de Artifactory correspond simplement en la création d'un dépôt de type Maven pour le stockage des modules. L'outil traitera ensuite automatiquement la sauvegarde des builds de chaque module.

Jenkins est un outil d'intégration continue dérivé de Hudson fonctionnant avec le web container Apache TomCat. Nous l'utiliserons pour automatiser les builds à chaque push vers le dépôt contenant le projet. Il enverra les builds en succès vers le dépôt d'Artifactory. Jenkins utilise un système de « job » qui correspond à une tache à effectuer. Nous utiliserons principalement 3 jobs : un job pour compiler et tester unitairement les modules du j2e, un job pour les tests d'intégration et un job pour le lancement et l'arrêt du server .NET.

Jenkins est configuré de façon à pouvoir exécuter 2 jobs simultanément, recompiler seulement les modules qui ont été modifiés et les modules qui en dépendent et envoyer seulement les builds en succès vers Artifactory.

II. Forces & Faiblesses

Dans cette partie nous aborderons les forces et faiblesses de l'architecture utilisée, comparée à la structure monolithique initiale.

1. Points forts

Modularité

- Seuls les modules modifiés et les modules qui en dépendent vont être recompilés. Nous avons ainsi un gain de temps non négligeable surtout dans le cadre d'un projet de grande envergure contenant un grand nombre de tests. Nous ne recompilons et testons que les modules qui sont impactés par les modifications.
- Nous pouvons travailler seulement en disposant du module que nous souhaitons modifier. Grâce à Artifactory, lorsque nous aurons besoin de compiler et/ou de tester, nous récupérons les .jar nécessaires pour faire fonctionner notre module.
- Nous avons un projet fortement découpé. Nous sommes ainsi certains que nous ne téléchargerons pas des parties du code inutile au module que nous sommes actuellement en train de modifier.

Fiabilité

- Lorsqu'un module est push sur github alors qu'il ne compile pas ou qu'un des tests ne passe pas, la modification n'est pas push vers Artifactory grâce à Jenkins. Les erreurs des uns n'empêchent donc pas les autres de travailler puisque Artifactory disposera et desservira toujours des modules fonctionnels.

Gestion de la mémoire

- En raison de l'utilisation d'une machine virtuelle, nous disposons d'une capacité de mémoire limitée. C'est pourquoi nous avons configuré Jenkins de façon à ce qu'il ne garde que les dix derniers builds (dont le dernier build en succès). De ce fait nous réduisons les possibilités d'un manque d'espace mémoire dû à Jenkins.

Intégration continue

- Les tests d'intégrations sont effectués régulièrement pour assurer la cohésion entre les différents serveurs.
- Jenkins est configuré de façon à ce que le serveur .NET soit lancé avant chaque exécution des tests d'intégration, et arrêté ensuite. Ceci est important car nous utilisons une machine virtuelle, ce qui est assez limité. Pour éviter tout problème, il vaut mieux ainsi que le serveur .NET s'arrête dès que nous finissons les tests d'intégration.

2. Points faibles

Modification non prise en compte

- Si nous modifions un module A et mettons un test d'un module B en erreur (A et B indépendants l'un de l'autre), le nouveau module A n'est pas push sur Artifactory à cause de l'erreur de B. Nous ne pouvons ainsi pas agir sur deux modules qui n'ont rien à voir et les push en même temps : si l'un compile et passe les tests mais pas l'autre, le système estime que l'erreur viens des deux donc pour ne pas prendre de risque, il ne push ni A ni B.

Attente active

- Jenkins scrute le projet toute les minutes pour recompiler les modules modifiés. Si nous avons beaucoup de jobs qui font ça, la bande-passante risque d'être saturée.
- S'il y a plusieurs push entre 2 scrutations et qu'un des push provoque une erreur de compilation ou de test, rien n'est envoyé sur Artifactory.

Plus long

- Le téléchargement des dépendances rallonge le temps de build comparé au code monolithique. En effet, ne disposant que du/des module(s) que nous modifions, il nous manque le reste du projet, il est donc récupéré lorsque nous compilons ou testons. Dans le cas du code monolithique, comme nous possédions tout le projet, nous n'avions pas à télécharger à chaque fois le code.

Gestion de la mémoire

- Contrairement à Jenkins, Artifactory garde tous les builds, la mémoire est n'est donc pas nettoyée et il y potentiellement un risque de saturation de mémoire.