

Apache Maven

**Nicolas De loof
et Arnaud Héritier**

Avec la contribution des membres francophones de la communauté Maven,
en particulier Stéphane Nicoll, Vincent Siveston, Raphaël Piéroni, Hervé Boutemy,
Jérôme Van Der Linden, Antonio Goncalves et François Le Droff

PEARSON

The Pearson logo consists of the word "PEARSON" in a white, uppercase, sans-serif font, centered within a dark rectangular background. Below the text is a thin, white, curved line that arches under the letters.

Pearson Education France a apporté le plus grand soin à la réalisation de ce livre afin de vous fournir une information complète et fiable. Cependant, Pearson Education France n'assume de responsabilités, ni pour son utilisation, ni pour les contrefaçons de brevets ou atteintes aux droits de tierces personnes qui pourraient résulter de cette utilisation.

Les exemples ou les programmes présents dans cet ouvrage sont fournis pour illustrer les descriptions théoriques. Ils ne sont en aucun cas destinés à une utilisation commerciale ou professionnelle.

Pearson Education France ne pourra en aucun cas être tenu pour responsable des préjudices ou dommages de quelque nature que ce soit pouvant résulter de l'utilisation de ces exemples ou programmes.

Tous les noms de produits ou marques cités dans ce livre sont des marques déposées par leurs propriétaires respectifs.

Apache, Apache Maven, Maven, and the Apache Maven logo are trademarks of The Apache Software Foundation. Used with permission. No endorsement by The Apache Software Foundation is implied by the use of these marks.

Publié par Pearson Education France
47 bis, rue des Vinaigriers
75010 PARIS
Tél. : 01 72 74 90 00
www.pearson.fr

Mise en pages : TyPAO

ISBN : 978-2-7440-2337-8
Copyright © 2009 Pearson Education France
Tous droits réservés

Aucune représentation ou reproduction, même partielle, autre que celles prévues à l'article L. 122-5 2° et 3° a) du code de la propriété intellectuelle ne peut être faite sans l'autorisation expresse de Pearson Education France ou, le cas échéant, sans le respect des modalités prévues à l'article L. 122-10 dudit code.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Le jour J : la livraison

La vie d'un projet est ponctuée de moments forts, et le jour de la livraison en est un particulièrement éprouvant. Après de longues semaines de développement et de mise au point, notre logiciel est mis à la disposition de ses utilisateurs, soumis à leur jugement et à leur manque de compassion pour les erreurs que nous aurions pu commettre.

Stratégie de livraison

Première livraison



La toute première version publique de *noubliepaslalistedescourses* a été particulièrement héroïque. Tout commence par plusieurs semaines de tests intensifs pour identifier tous les bogues qui auraient pu nous échapper. Les derniers correctifs sont appliqués, avant de se lancer dans une ultime journée de vérification pour être sûr de ne rien laisser au hasard. Après quoi, il nous faut préparer le logiciel enfin stabilisé et validé pour la diffusion. Emmanuel prend en charge la préparation de cette toute première mouture.

Il commence par parcourir tous les fichiers de configuration du projet pour indiquer comme numéro de version la fameuse valeur `1.0.0`. Le projet, reconstruit avec cette nouvelle configuration, est testé une dernière fois avant d'être rendu public. Emmanuel marque dans notre gestionnaire de code source ce moment crucial, ce qui se traduit par la pose d'un "tag". La journée se termine donc tard autour d'une bonne bouteille pour fêter le succès de notre belle aventure. *noubliepaslalistedescourses* est désormais en ligne et attend de pied ferme ses utilisateurs.

Deuxième livraison

Une mauvaise surprise nous tombe dessus lorsque nous devons en urgence appliquer une correction pour un problème de performances. Soumise à un succès inattendu, l'application a commencé à cafouiller et il a fallu trouver des palliatifs immédiats. Le tag posé par Emmanuel nous permet de récupérer le code source associé à cette fameuse version 1.0.0 et de préparer aussi vite que possible une version 1.0.1 corrigeant ce problème. Et là, impossible de reconstruire le projet. Lors de la première livraison, les modifications n'avaient pas été complètement intégrées dans le gestionnaire de code source lors de la pose du tag. Emmanuel doit donc gérer dans l'urgence deux problèmes là où un seul lui suffisait déjà largement.

Pour ne pas reproduire la même erreur, il décide de poser le tag de la version 1.0.1, **puis** d'extraire du gestionnaire de sources le code associé au tag encore tout frais et de s'en servir pour construire le logiciel qui sera validé une toute dernière fois avant d'être publié. Si jamais nous détectons un problème de dernière minute, il nous suffirait d'apporter les corrections nécessaires, de poser un nouveau tag et de reprendre la procédure.

Troisième livraison

Les choses semblent enfin maîtrisées, mais nous déchantons vite quand d'autres problèmes apparaissent sur la version 1.0.1. Après analyse, on comprend que ces problèmes auraient dû être détectés par notre outillage de tests. Alors, que s'est-il passé ?

Ces tests sont malheureusement associés à un profil qui doit être activé à la demande. Comme ils portent sur les accès à la base de données, et pour ne pas pénaliser les autres développeurs, nous les avons isolés dans un profil dédié, comme nous l'avons vu au Chapitre 4. Était-ce une fausse bonne idée ?

L'équipe de développement, malgré le fiasco de cette version corrective, n'en est pas convaincue. Les profils permettent à chacun d'être correctement outillé sur la partie qui le concerne sans être handicapé par des temps de construction interminables ou par des prérequis sur l'environnement de développement.

Lors de la construction de notre projet avant livraison, nous devons tout simplement ne pas négliger d'activer tous les profils adéquats pour valider au maximum l'application. Nous pouvons même ajouter, à cette phase cruciale, des profils spécifiques pour adjoindre des informations dans le livrable : date de livraison, auteur de la livraison, numéro de révision dans le gestionnaire de code source...

La version 1.0.2 sera donc la bonne, maintenant qu'Emmanuel tient enfin une procédure de livraison fiable et... connue de tous ?

Documentation

La procédure peut être aussi bétonnée qu'on le voudra, elle n'est fiable que si elle est appliquée scrupuleusement. Autrement dit, le facteur humain reste comme toujours l'épée de Damoclès qui menace les projets.

La plupart des entreprises répondent à ce problème par une documentation lourdement contrôlée, répondant à des règles très strictes de contrôle qualité et à des nomenclatures précises. En supposant que tout le monde soit rompu aux pratiques de qualité et vive dans un monde labellisé AFAQ et ISO-900x, cela devrait garantir le succès du projet en toute circonstance. Seulement, nous ne vivons pas dans ce monde – est-ce réellement regrettable ?

D'autres préfèrent une approche plus souple et réactive en faisant appel à l'auto-organisation, par exemple *via* une documentation supportée par un wiki.

INFO

Pour ceux qui ne le sauraient pas, un wiki est un site web dont le contenu peut être modifié, corrigé ou complété par ses utilisateurs. Il est donc très dynamique et peut s'adapter à la réalité des problèmes rencontrés et s'organiser en fonction des informations réellement utiles.

Un wiki n'est pas du tout incompatible avec une gestion "classique" de la documentation. Si vous n'en avez pas un sur votre projet, faites un petit essai avec votre équipe, vous serez surpris du résultat. Les solutions ne manquent pas, du wiki PHP de base au très professionnel Confluence, en passant par l'incontournable XWiki cher à Vincent.

Emmanuel documente ainsi la procédure de livraison dans notre wiki, accessible rapidement et sans les ambiguïtés d'une classification documentaire complexe. Elle a aussi l'énorme mérite de permettre à celui qui applique la procédure de venir compléter la documentation pour préciser un point qu'il trouve mal expliqué, ou donner des indications sur un point pas assez détaillé. Certains seront très à l'aise avec les commandes Unix alors que d'autres auront besoin qu'on les prenne par la main.



Est-ce vraiment la panacée ? Pourquoi acceptons-nous que le moment de la livraison – le plus crucial de la vie du projet – soit le seul à ne pas bénéficier de l'approche que Maven a apportée au projet : l'automatisation aussi complète que possible *via* une commande unique. Emmanuel a pris le soin de documenter dans le détail la procédure, et le principe du wiki permet à chacun de la compléter si un point restait obscur. Cependant, nous construisons tous nos projets, quelles que soient leur complexité ou leur technologie, par la commande universelle `mvn install`. La procédure de livraison ne pourrait-elle pas elle aussi être homogénéisée ? Emmanuel part donc à la pêche aux bonnes pratiques du côté de Maven et de ses plugins...

Le plugin release

Le plugin release de Maven a été conçu dans cet esprit. Il regroupe en une seule commande toutes les bonnes pratiques de livraison apprises sur de nombreux projets. En l'utilisant, vous n'aurez pas comme nous à apprendre à vos dépens que l'établissement d'une procédure fiable ne s'improvise pas. Surtout, vous n'aurez pas besoin de 20 pages de documentation pour indiquer la procédure à suivre.

Pour vous donner une idée de ce que le plugin propose, voici le processus qu'il applique pour produire un livrable.

Étape 1 : préparation

- Il contrôle l'environnement de l'utilisateur qui ne doit présenter **aucune modification non sauvegardée** dans le gestionnaire de code source. Optionnellement, un accès exclusif au gestionnaire peut être demandé pour les plus paranoïaques d'entre nous.
- Il contrôle le projet qui ne doit référencer **aucune dépendance en SNAPSHOT**, dont l'état serait par nature non reproductible.
- Il modifie les **indications de version** dans les fichiers POM des modules du projet. Le plugin peut être configuré pour affecter la même version à toutes les branches d'un projet multimodule, sinon il demandera de saisir une version pour chacun d'eux.
- Il indique dans le POM du projet le **tag appliqué à la version à produire**. Le projet comptera donc une référence explicite à l'emplacement où son code source est stocké.
- Il effectue une construction complète du projet pour **contrôler les modifications appliquées automatiquement** et s'assurer qu'elles n'ont pas un impact négatif.
- Il **sauvegarde** les modifications appliquées dans le gestionnaire de code source.
- Il **pose un tag** dans le gestionnaire de code source pour la livraison en cours de préparation.
- Il modifie à nouveau les indications de version pour pointer vers la **version de développement suivante** et les sauvegarde.

Comme vous pouvez le constater, la démarche est plus que rigoureuse et elle est entièrement automatisée. Au final, le projet dispose, dans son gestionnaire de versions, d'un

tag pour la version à livrer, et la version courante (*trunk*) correspond à la prochaine version à développer. L'état "version livrée" n'est apparu que furtivement dans l'historique du gestionnaire de code source, ce qui correspond bien à la réalité de cet événement aussi ponctuel que capital.

Étape 2 : livraison

La production du logiciel livrable est réalisée à partir du tag placé dans le gestionnaire de sources. Nous pourrions effectuer cette manœuvre manuellement, mais ici encore le plugin `release` simplifie la tâche :

- **extraction du code** correspondant au tag dans un répertoire dédié, vierge de toute modification locale malencontreuse ;
- construction du projet à partir d'une liste de cibles, d'options et de profils **indiqués dans le POM**.

Le point capital ici est que la description de la configuration et des commandes nécessaires à la construction correcte du livrable est entièrement documentée dans le POM lui-même. Pas de document à connaître, à retrouver dans la bonne version ou à ne surtout pas oublier de mettre à jour.

Un projet Maven utilisant le plugin `release` permet de réaliser une livraison totalement maîtrisée, contrôlée et synchronisée avec le gestionnaire de code source en une simple commande :

```
mvn release:prepare release:perform
```

Emmanuel est plus que satisfait par cette simplicité. Maven prouve ici son efficacité : une commande unique pour réaliser un traitement loin d'être simple et applicable sur tous nos projets, toutes complexités et technologies confondues. Reprenez le document `PROCx589002-02.1-Procédure de livraison.doc` que vous aviez jusqu'ici et servez-vous-en pour caler votre armoire bancaire. Et si votre responsable qualité vous réclame un document de procédure, faites-lui une photocopie de la Figure 10.1 !

ASTUCE

Cette procédure est loin d'être élémentaire et il y a donc de nombreuses raisons qu'elle plante en cours de route avant d'être complètement au point. Le plugin propose un mode **dryRun** qui permet de l'exécuter à blanc, sans qu'aucune modification soit appliquée dans le gestionnaire de code source, le temps de bien mettre au point la configuration du plugin.

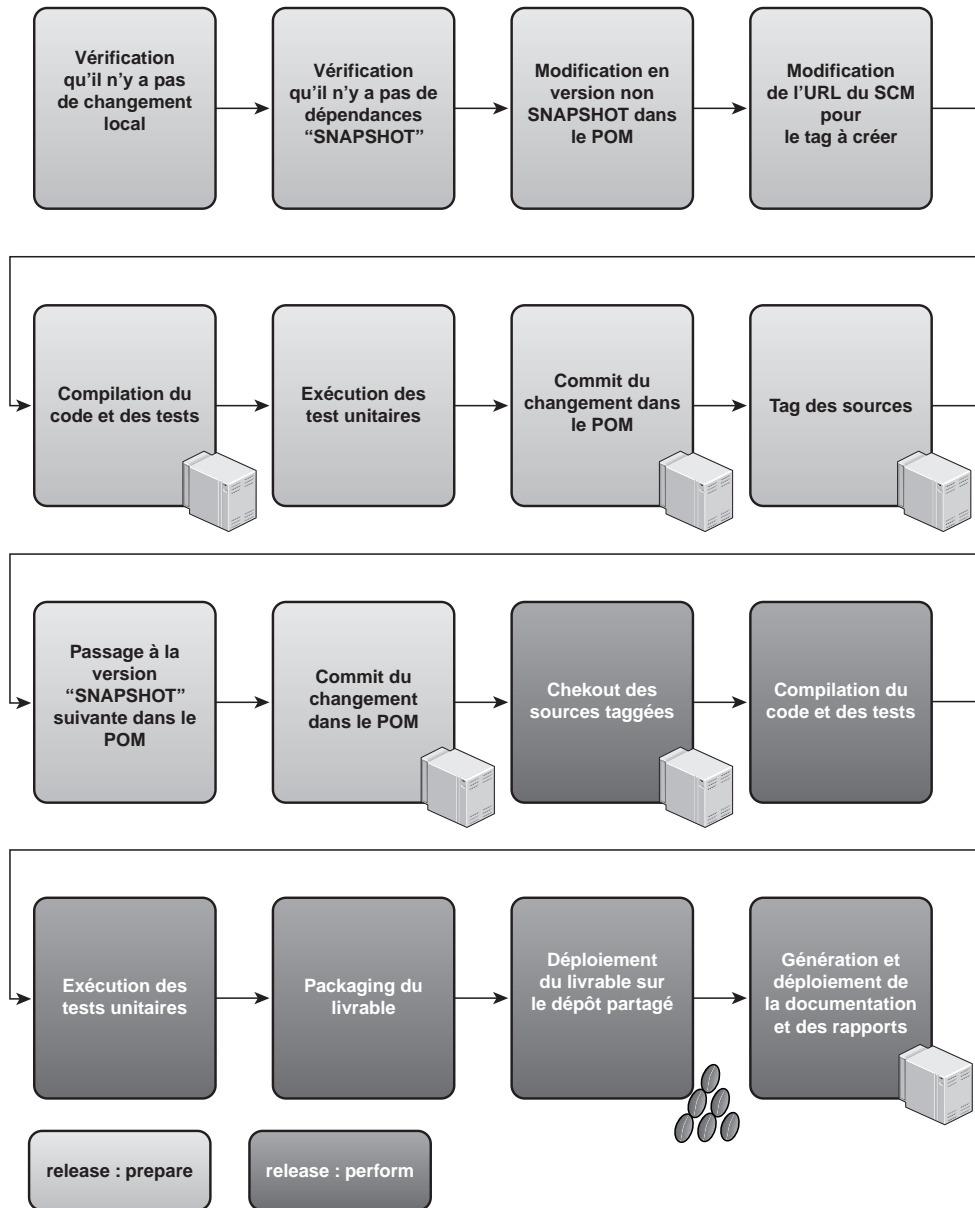


Figure 10.1

Étapes réalisées par le plugin `release`.

Et si ça foire ?

La préparation de la livraison comprend de nombreuses étapes et contrôles. Si l'une d'elles échoue, le plugin va évidemment s'interrompre. Après avoir identifié et corrigé le problème, nous avons deux options :

- Reprendre toute la procédure à son point de départ. Le plugin `release` propose alors de faire un retour arrière complet dans l'état initial, dans l'esprit de ce que propose une base de données au cours d'une transaction :

```
mvn release:rollback
```

- Reprendre la procédure à partir du point où elle s'est arrêtée précédemment.

La seconde solution est évidemment moins stricte ; cependant, vous pouvez parfois être bloqué par un problème purement local et indépendant de la stabilité du projet, comme un contrôle de votre environnement par le plugin `enforcer` qui échoue, ou la construction qui n'aboutit pas parce que vous n'avez pas attribué suffisamment de mémoire à l'exécution de Maven.

Dans les deux cas, la résolution du problème ne nécessite pas de modifier le projet lui-même, et le plugin `release` permet de reprendre le processus : par défaut, lancé une deuxième fois sur le même projet, il va reprendre sa tâche là où il en était resté, sauf si on lui indique explicitement de tout reprendre de zéro *via* l'option `-Dresume=false`.

Notre prochaine version

La livraison de la version 1.2.0 de *noubliepasalistedescourses* n'a pas été une de ces journées de stress sans fin, qui se terminent bien après le coucher du soleil. Elle n'a pas non plus été menée par un expert rompu à tous nos outils et spécialiste de la procédure. C'est notre stagiaire qui l'a réalisée, le plus naturellement du monde, et sans même se rendre compte de ce qu'avaient pu être les livraisons précédentes.

L'approche par convention et l'excellente extensibilité de Maven montrent toute la force de celui-ci dans le plugin `release`. Un seul fichier, utilisant un formalisme XML certes verbeux mais simple, permet de décrire toutes les étapes de construction et toutes les options nécessaires pour obtenir un résultat fiable.

Tester des candidats

Jusqu'à présent, nous avons utilisé notre outillage de tests automatisé et une série de tests en amont pour valider le fonctionnement de notre logiciel avant livraison. Nous ne sommes cependant jamais à l'abri d'une coquille de dernière minute qui passerait au

travers du processus de livraison. Aussi, nous voulons tester une dernière fois avant de mettre le coup de tampon "bon pour le service".



C'est à nouveau Emmanuel qui s'y colle, maintenant qu'il est devenu notre gourou de la livraison. Pour notre version 1.3.0, il propose de compléter notre procédure techniquement bien rodée d'une phase supplémentaire de tests. La version marquée 1.3.0 sera installée sur notre serveur de validation et subira nos tests les plus redoutables. Comment gérer cependant les versions dans les essais successifs que nous allons probablement enchaîner ? Chaque candidat au titre de livraison officielle a sa propre identité et ne doit pas être confondu avec les autres.

Première option, utiliser un nombre supplémentaire dans le numéro de version, indiquant le numéro du candidat dans la course au logiciel sans faille. La 1.3.0.1 sera probablement imparfaite, la 1.3.0.2 aura son lot de régressions inacceptables, la 1.3.0.3 sera peut-être la bonne. Certains logiciels suivent cette option et ne diffusent publiquement que les sous-versions qui ont satisfait tous les critères de qualité. Les utilisateurs peuvent cependant être surpris de constater des manques dans la série des numéros de version.

Emmanuel n'aime pas trop cette solution et voudrait pouvoir redéfinir la version de notre meilleur candidat en "1.3.0". Seulement, le code marqué dans notre gestionnaire de code source ne porterait pas la bonne version, et le tag serait inadapté. Une fois de plus, Maven et son plugin `release` viennent à notre secours. Lorsque nous exécutons la commande `mvn release:perform`, nous demandons à Maven de construire le livrable et de le diffuser sur notre dépôt public. Une autre commande `mvn release:stage` est très comparable mais remplace automatiquement la phase de déploiement pour pointer sur notre dépôt de validation. Le logiciel installé sur celui-ci sera donc strictement identique, au bit près, à une livraison classique mais aura juste changé de destination. Nous pourrions donc le tester attentivement en toute tranquillité.

La seconde différence qu'introduit `stage` est qu'il conserve la possibilité de lancer un `rollback`. Il est donc possible, si la version en cours de test ne nous satisfait pas, de revenir en arrière sur notre gestion de version et de relancer plus tard une nouvelle version avec le même numéro de version.

Si effectivement notre premier "release candidate" est insuffisant pour répondre à nos exigences, il nous suffit de revenir *via* un `release:rollback` en version `1.3.0-SNAPSHOT`. La seule chose qui restera du candidat malheureux est le tag, que nous pouvons renommer en `1.3.0-RC1`. Si, par malheur, toutes nos tentatives ne produisaient qu'un logiciel encore pire, nous pourrions toujours, pressés par le temps, repartir de ce tag. Celui-ci présente bien la capacité de produire un projet en version 1.3.0 !



Carlos est très intrigué par les explications d'Emmanuel sur cette procédure de mise sur étagère d'une version candidate. Il propose alors d'utiliser un autre mécanisme, en se fondant sur le gestionnaire de dépôt dont il a la responsabilité. Notre procédure de livraison va produire un livrable dans la version cible 1.3.0 et le diffuser sur le gestionnaire de dépôt. Par contre, l'emplacement dans ce dépôt sera choisi dans une sous-catégorie spéciale, dédiée aux préversions candidates. Nos bêta-testeurs pourront pointer dessus pour récupérer le binaire et le tester en connaissance de cause, alors que les autres utilisateurs n'en auront pas connaissance. Une fois cet artefact validé, il suffira de demander au gestionnaire de dépôt de le **promouvoir** dans le dépôt public (sous Nexus, dans la version *pro*, voir la gestion des *staging repositories*).

Emmanuel et Carlos nous proposent ainsi, par le biais d'outils propres au monde Maven, une procédure complète, fiabilisée et totalement automatisée pour produire notre livrable, le tester puis le diffuser.

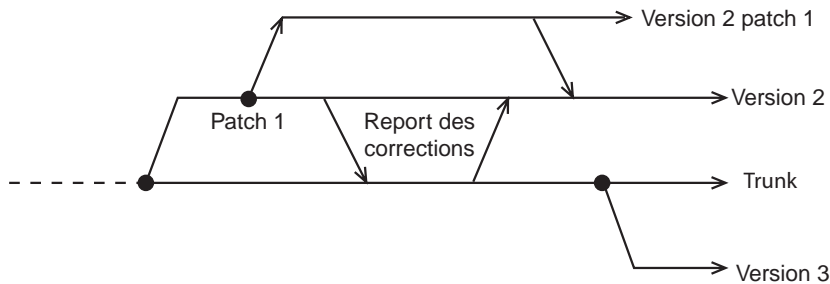
Urgence !

Notre version 1.3.0 n'est pas encore complètement stabilisée que nous avons un retour alarmant d'un de nos clients sur la version 1.2.0. Un bogue très gênant, qui nécessite de notre part une intervention immédiate. Impossible de proposer une migration dans la nouvelle version qui n'est même pas encore prête et apporte de toute façon des changements significatifs dont notre client n'a que faire dans son état d'esprit du moment.

Il nous faut un correctif, une 1.2.1, et le plus vite sera le mieux. Et c'est bien sûr sur Emmanuel que ça tombe une fois de plus !



Mais, cette fois, Emmanuel ne s'est pas laissé prendre de vitesse et a prévu le coup. Notre gestionnaire de code source prend ce problème en charge à travers la notion de **branche**. En parallèle, vont commencer à croître d'un côté le tronc du projet, correspondant à la version 1.3.0 qui continue son chemin, et, d'un autre côté, une branche 1.2.1, dont le point de départ est le moment exact de création de la version 1.2.0. Les divergences entre les codes de ces deux versions pourront être fusionnées plus tard, ou peut-être resteront-elles définitivement séparées, l'avenir nous le dira. La Figure 10.2 donne une idée de la gestion en parallèle de versions, avec un point de branchement et un point de fusion quelques semaines plus tard. Cette approche peut être utilisée pour des besoins correctifs ou pour expérimenter à l'abri d'un petit coin tranquille une évolution complexe, qui impacterait trop le reste de l'équipe.

**Figure 10.2**

Utilisation d'une branche de développement parallèle.

Encore une fois, le plugin `release` va nous faciliter la tâche, ce qui est la moindre des choses lorsqu'il faut en plus gérer l'énervement de notre client au bout du fil et lui prouver notre réactivité. `mvn release:branch` nous permet de créer rapidement une branche de développement, avec mise à jour de nos POM pour indiquer la version corrective et les références adéquates à notre gestionnaire de code source. Nous pouvons alors commencer à travailler dans la branche fraîchement créée sans perdre de temps.

Il faudra juste configurer le serveur d'intégration continue pour scruter l'avancement de ce développement parallèle. Sous Hudson, nous pouvons simplement copier la configuration existante pour créer un nouveau *job*. Seul l'emplacement du gestionnaire de code source sera modifié.

Une fois notre développement terminé, le couple `release:prepare` `release:perform` reprendra du service pour produire ce livrable correctif très attendu. Une fois de plus, toute cette procédure aurait pu être traitée manuellement, dans une ambiance de stress et de précipitation dont les résultats peuvent être désastreux. Maven propose de les prendre en charge de manière totalement structurée et automatisée.

Au-delà de l'intégration continue

Nous avons déjà mis en œuvre l'automatisation de notre construction de projet au sein d'un serveur d'intégration continue. Maintenant que nous savons tout aussi bien automatiser la construction de nos livrables, il est naturel de franchir un pas supplémentaire.

On parle parfois de **production continue** pour décrire cette nouvelle étape dans l'automatisation, et c'est une pratique qui s'inscrit dans la démarche d'usine logicielle que de nombreuses entreprises cherchent à appliquer à leurs développements informatiques pour échapper à un monde où règnent le bricolage et les manipulations manuelles hasardeuses. La Figure 10.3 présente le principe général de cette approche.

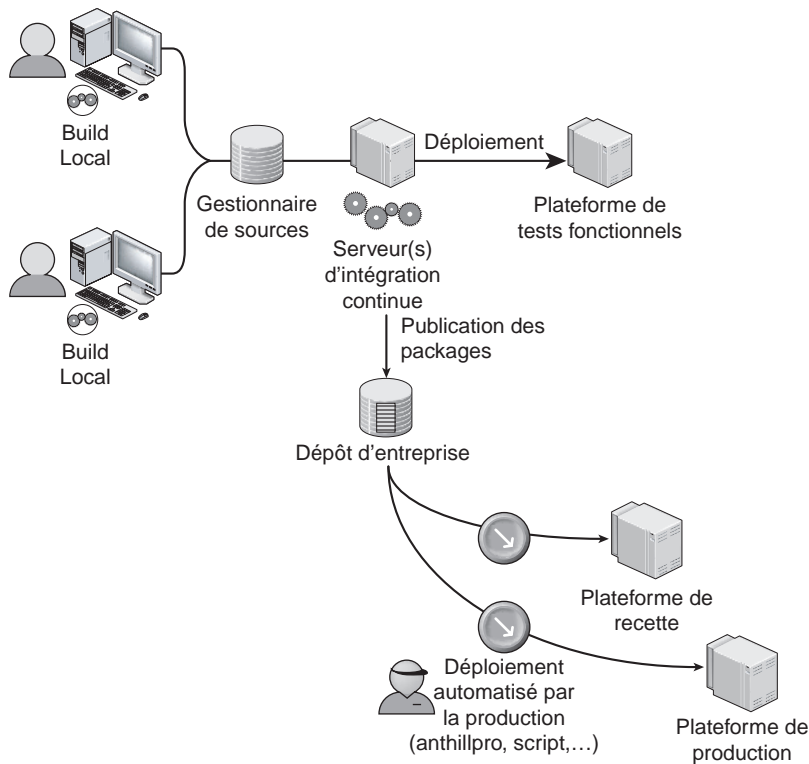


Figure 10.3

Production continue.

Le concept est au final relativement simple. Nous disposons d'une automatisation capable de construire, tester en profondeur et publier les versions stables de notre logiciel, avec toute la rigueur et la traçabilité dont peut rêver n'importe quel ingénieur qualité. La seule étape qui reste encore plus ou moins aléatoire est l'installation sur nos plateformes de recette puis de production. Pourtant, comme nous l'avons vu au Chapitre 8, Maven – assisté de Cargo – est tout à fait capable de déployer nos applications JEE sur un serveur.

En production continue, nous allons ajouter dans la description du projet (dans notre POM), ces étapes recette et de mise en production. Il ne restera donc entre le développeur, qui saisit son code, et l'administrateur, qui installe la nouvelle version, aucune inconnue plus ou moins bien documentée. Bien sûr, nous devons aussi intégrer dans le processus la gestion des migrations de données, la possibilité d'un retour arrière, mais ce n'est après tout qu'un problème d'automatisation d'un processus que nous gérons jusqu'ici à la main.

L'intégralité du processus de réalisation de notre logiciel est ainsi embarquée dans le POM. Les tâches de construction, de test, de validation de nos règles de développement, de qualification, de contrôle fonctionnel et d'installation sont entièrement automatisées et ne dépendent plus que de la décision de l'administrateur de cliquer sur le bouton vert, ou plutôt de lancer un `mvn -Pproduction monserver:deploy`.



Emmanuel est plus que satisfait et raconte avec un sourire nostalgique à nos stagiaires les journées rocambolesques des premières versions.

Conclusion

L'automatisation des processus de développement est une tâche qui nécessite un gros effort initial, non seulement pour dépasser les contraintes techniques mais surtout pour faire passer les mauvaises habitudes et inculquer une nouvelle vision. N'espérez pas créer en quelques jours une **usine logicielle** mettant en œuvre des tests unitaires, d'intégration et fonctionnels sur une grille de machines. Commencez petit sur des projets pilotes, essayez les plâtres de vos premières erreurs et attendez un peu pour récolter les fruits d'un processus enfin compris et dont les utilisateurs seront les meilleurs évangélistes.

La livraison fait partie de ces étapes qui concernent tout le monde avec un haut niveau de stress. L'automatisation y prend toute sa force et démontre son intérêt. Maven peut vous apporter de nombreux services, structurer votre projet, mais c'est certainement sur un point aussi stratégique qu'il s'imposera comme outil incontournable. Chaque spécificité de votre projet étant automatisée, documentée et historisée *via* votre fichier POM, vous pourrez enfin vous focaliser sur la seule chose qui compte vraiment : le fonctionnement de votre application.