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 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.

2

Au-delà de `java.lang`

Des JAR sous CVS

Avec une équipe qui se compose désormais de cinq développeurs motivés, il n'est plus question de s'envoyer par e-mail des archives du projet pour transmettre aux autres les nouvelles fonctions que l'on vient de développer. Un projet en mode collaboratif utilise un outil de gestion de sources pour partager le code, synchroniser les développements et gérer les conflits lorsque deux personnes travaillent sur le même fichier. Ce gestionnaire de sources (SCM – *Source Control Management*) est typiquement CVS (*Concurrent Version System*), Subversion ou, plus récemment, Git.

Comme son nom l'indique, cet outil est prévu pour contenir des fichiers sources et non des binaires issus d'une compilation. Pourtant, de nombreux projets placent les bibliothèques et les outils nécessaires au projet dans leur gestionnaire de sources. L'idée peut sembler bonne *a priori*, car elle vise à gérer avec un unique outil et, de manière homogène, tous les éléments nécessaires au développement du projet. Sauvegarder les bibliothèques Java dans le SCM est donc une garantie de retrouver à tout moment la version exacte qui est utilisée par le projet.

Notre prototype ne déroge pas à cette "bonne idée" et possède comme tant d'autres un répertoire `lib` avec l'ensemble des bibliothèques utilisées.

Quand le répertoire `lib` explose

La croissance de l'équipe nous permet de rapidement améliorer notre prototype. Le nombre de bibliothèques nécessaires au projet augmente. Nous commençons par introduire Spring pour rendre le code plus évolutif avec l'utilisation des concepts de l'injection de dépendances. Ensuite, nous remplaçons tout le code écrit en JDBC par Hibernate et Java Persistence API. Nous développons une interface web sympathique

basée sur Wicket et, enfin, nous faisons appel à Apache CXF pour exposer nos services à d'autres applications sous forme de services web.

Le nombre de bibliothèques croit exponentiellement car, au-delà de la gestion de celles que nous utilisons explicitement au sein du projet, il faut gérer toutes les bibliothèques qui leur sont nécessaires. Rapidement, le répertoire `lib` se retrouve chargé de dizaines de fichiers `JAR` avec des noms plus ou moins hétéroclites.

Les choses se compliquent alors significativement et la moindre mise à jour d'une bibliothèque relève d'un casse-tête chinois.

D'une part, cette pratique encourage à utiliser ces bibliothèques telles quelles, sans chercher à s'assurer de leur origine ou de la fiabilité de leur téléchargement. Comme il est délicat de comparer deux versions d'un fichier binaire, il nous est impossible de savoir en quoi notre fichier `util.jar` diffère de celui utilisé sur un autre projet comparable, dont nous voulons importer des classes intéressantes. Même si ces deux fichiers portent le même nom et ont la même taille, cela ne signifie pas qu'ils soient identiques. Seule une comparaison binaire pourrait nous en assurer.

Autant dire qu'avec les dizaines de bibliothèques embarquées dans notre projet, plus personne ne fait scrupuleusement cette vérification et nous nous contentons de lire le nom de l'archive `mail-1.2.jar` pour identifier la bibliothèque JavaMail.

Cela nous amène à un second problème possible. Supposons que cette bibliothèque ait été corrompue lors de son téléchargement depuis le site de SUN qui la diffuse ou de son enregistrement dans notre SCM. Un transfert réseau n'est jamais 100 % garanti, et un seul bit modifié peut rendre la bibliothèque inutilisable, sans parler de ces charmants petits virus qui peuvent traîner un peu partout. L'identification du problème peut être extrêmement complexe, car la remise en cause de la bibliothèque sera probablement la toute dernière hypothèse que nous évoquerons pour justifier un dysfonctionnement.

Un bogue est détecté

Après quelques heures de tests et de recherche d'informations sur Internet, nous devons nous rendre à l'évidence, nous rencontrons un bogue connu de la bibliothèque JavaMail utilisée sur le projet. Seule solution viable : la mise à jour de cette bibliothèque dans une version plus récente.

Le téléchargement de la distribution JavaMail depuis le site de SUN[1] donne un fichier ZIP contenant à la fois les binaires et la documentation de cette API ; les binaires, car JavaMail regroupe en fait plusieurs archives `JAR`, à savoir `mail.jar`, mais aussi

1. **http://java.sun.com/products/javamail/**.

`mailapi.jar`. La première contient l'ensemble du code public JavaMail alors que la seconde ne comprend que les API de programmation, et pas la gestion des protocoles de transfert de messages (pop, smtp, imap…) qui sont optionnels. Lequel utilisons-nous actuellement ? Par quoi le remplacer ?

En supposant que nous sachions répondre sans ambiguïté à cette question, nous devons supprimer le `mail-1.2.jar` utilisé jusqu'ici et ajouter le nouveau `mail-1.4.1.jar`. Cela nous impose de modifier tous nos scripts de gestion du projet (scripts de compilation et de lancement, fichiers de configuration Eclipse, NetBeans ou IntelliJ Idea…) pour tenir compte de ce changement, avec le risque d'introduire, par mégarde, des erreurs. Ce simple changement nous oblige donc à la fois à faire preuve de beaucoup de soin et à vérifier le fonctionnement de nos scripts.

Pour éviter ces risques, une seconde option consiste à ne pas indiquer de numéro de version pour les bibliothèques. Nous utilisons le nom de fichier `mail.jar` et le remplaçons purement et simplement par le nouveau fichier en cas de mise à jour. Ayons alors une pensée compatissante pour les équipes de maintenance qui, dans quelques années, devront deviner la version exacte des bibliothèques utilisées sur notre projet, dont certaines seront devenues plus ou moins obsolètes et connues pour certains bogues graves. Le problème devient encore plus complexe lorsqu'on doit utiliser une version modifiée d'une bibliothèque, par exemple parce qu'on y a intégré un correctif qui n'est pas encore pris en compte dans une version officielle.

INFO

Le format d'archive `JAR` prévoit un fichier de métadonnées, `META-INF/MANIFEST.MF`, décrivant théoriquement la bibliothèque, et en particulier sa version précise. Celle-ci est cependant régulièrement non documentée lorsque ce fichier MANIFEST n'est pas tout simplement absent ou quasiment vide.

lib/.jar*

Pour ne plus rencontrer ce problème, nous décidons "d'assouplir" nos scripts de compilation en utilisant l'intégralité du répertoire `lib` comme chemin d'accès aux classes, plutôt qu'une liste explicite de bibliothèques. Placer une nouvelle bibliothèque dans ce répertoire ou en remplacer une par une autre version ne nécessitera alors aucune modification des scripts.

Ce qui pourrait ressembler à la solution miracle n'est pas aussi parfait qu'il y paraît. D'une part, cela ne résout pas la configuration de notre environnement de développement qui continue de réclamer une liste précise de bibliothèques à inclure dans le ClassPath. Ensuite, une manipulation malheureuse de nos fichiers `JAR` ne se verra pas

immédiatement – un glisser-déposer est si vite arrivé ! Il faudra attendre qu'elle ait un impact visible pour devoir ensuite remonter à l'origine du problème.

Enfin, ce n'est pas une solution d'une grande élégance. Pour éviter de devoir traiter le problème, nous avons ouvert les portes en grand à tout ce qui passe. Difficile de parler de "maîtrise" de nos bibliothèques dans de telles conditions.

Identification univoque

Maven propose une approche à l'opposé de ces pratiques hasardeuses. Il se focalise sur l'identification exacte des bibliothèques utilisées. Des référentiels de bibliothèques sur Internet lui sont dédiés et permettent de télécharger les bibliothèques précises, utilisées dans le projet, sans ambiguïté. Les risques d'erreur de transfert sont éliminés par un contrôle automatique basé sur des fonctions de hachage (une sorte d'empreinte digitale du binaire, qui sera invalidée au moindre bit invalide).

L'identification d'une bibliothèque utilisée par un projet s'appuie sur un triplet (`identifiant de groupe`, `identifiant d'artefact`, `version précise`), lequel est construit sur le même principe que celui que nous avons déclaré pour notre projet. Dans le fichier POM de Jason, nous référençons l'artefact `mail` de la bibliothèque standard `javaMail` dans sa version 1.4.

```
<dependency>
  <groupId>javax.mail</groupId>
  <artifactId>mail</artifactId>
  <version>1.4</version>
</dependency>
```

Il n'y a ici aucune équivoque possible. Toute autre variante de JavaMail possédera dans le dépôt Maven un numéro de version différent. Si nous devions nous-mêmes appliquer un correctif, nous devrions utiliser un numéro de version adéquat, comme `1.4-patch-1234`. Dans ce cas, cette bibliothèque modifiée serait placée dans notre dépôt privé, comme nous le verrons au Chapitre 6.

Notre projet inclut également une mystérieuse bibliothèque `util.jar`. Nicolas ne se souvient pas du tout de l'origine de ce fichier. Les équipes de maintenance, confrontées à ce cas de figure, auraient du fil à retordre. Comment gérer une mise à niveau ou un bogue rencontré dans la bibliothèque considérée si on est incapable de l'identifier avec précision ?

Dans le contenu de cette archive java, les packages utilisés, `org.apache.commons.io`, nous mettent sur la piste, et c'est ce qui a inspiré à Jason la déclaration d'une dépendance vers Apache Commons-io. Cependant, il pourrait s'agir d'une version modifiée, pour une quelconque raison, avec je ne sais quel impact possible sur l'application.

L'identification exacte réclamée par Maven oblige à préciser quelle version est utilisée et à définir des numéros de version pour chaque variante de la bibliothèque ou version modifiée que l'on voudrait utiliser.

Ajouter une bibliothèque à un projet Maven se traduit simplement par l'ajout d'un bloc `<dependency>` comparable à notre exemple, identifiant sans équivoque notre intention. Pas de script à éditer, pas de fichier JAR à télécharger et donc pas de validation du fichier téléchargé ; pas de répertoire de bibliothèques à modifier, avec les risques d'erreur de synchronisation qui en découleraient. Mettre à jour une bibliothèque consiste tout simplement à modifier l'information de version qui lui est associée.

Astuce

Les bibliothèques standard de Java sont hébergées par SUN et devraient donc être placées sous le groupe `com.sun.java`. Elles ne peuvent cependant pas être considérées comme des fournitures appartenant à cet éditeur. Aussi, la convention pour ce cas particulier veut qu'on utilise le nom de package `javax.*` qui caractérise ces API. Par ailleurs, il existe de nombreuses exceptions pour des raisons historiques liées à la première mouture de Maven.

Dépôt de bibliothèques

La configuration par défaut de Maven utilise le dépôt (ou *référentiel*) de bibliothèques **http://repo1.maven.org/maven2/**. Ce site, maintenu par la communauté Maven, compte plusieurs dizaines de gigaoctets de bibliothèques libres de diffusion et est mis à jour plusieurs fois par jour. Nous verrons au fil des prochains chapitres comment utiliser d'autres dépôts et en construire un pour ses besoins propres.

À partir de notre déclaration de dépendance, Maven va construire l'URL du sous-répertoire dédié à la bibliothèque indiquée :

```
<URL du dépôt> / <groupId en tant que chemin> / <artifactId> / <version>
```

Pour notre dépendance à JavaMail, nous obtenons : **http://repo1.maven.org/maven2/javax/mail/mail/1.4/**.

En plus du fichier JAR de la bibliothèque attendue, nous trouvons de nombreux autres fichiers dans ce répertoire :

- Chaque fichier présent est accompagné de deux partenaires, avec respectivement l'extension `.md5` et `.sha`. Il s'agit des empreintes de contrôle associées au fichier, que Maven exploitera pour s'assurer que le fichier n'a subi aucune altération au cours du téléchargement.
- Un fichier porte le même nom que la bibliothèque avec le suffixe `-sources`. Il s'agit, comme on pourrait s'en douter, d'une archive des sources Java de la bibliothèque,

ce qui pourra se montrer fort utile depuis votre environnement de développement intégré préféré pour utiliser un débogueur et parcourir le code de cette bibliothèque. Il pourrait également y avoir un autre fichier avec le suffixe `-javadoc` contenant la documentation technique de la bibliothèque.

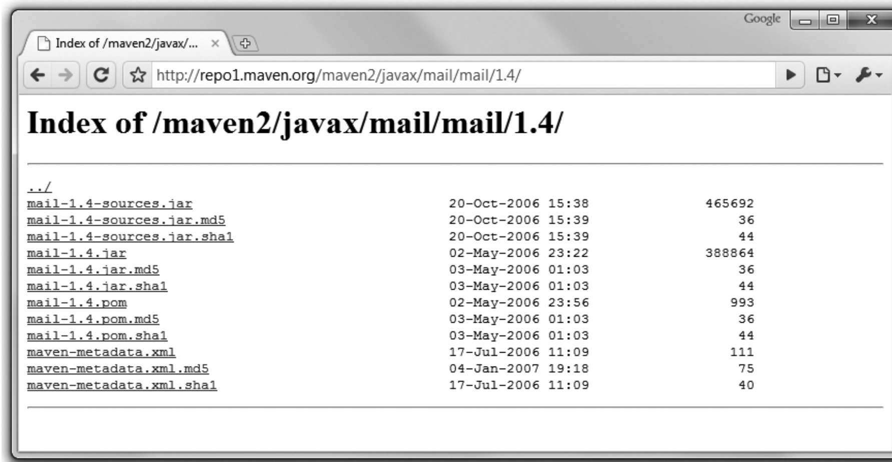

Figure 2.1

Le sous-répertoire dédié à JavaMail 1.4 sur le dépôt de bibliothèques.

- Un autre fichier ayant le même nom que la bibliothèque avec l'extension `.pom`. Il s'agit bien de l'acronyme du *Project Object Model* que nous connaissons déjà. Chaque bibliothèque dans le dépôt Maven possède un fichier de ce type. Soit parce que la bibliothèque a été développée en utilisant Maven, soit parce qu'un fichier minimal a été écrit pour fournir une description de la bibliothèque aux utilisateurs de Maven.
- Un fichier de métadonnées, propre à Maven comme son nom l'indique clairement.

Avis aux amateurs de casse-tête

Notre projet, issu d'un code antédiluvien auquel chacun est venu apporter sa contribution, est constitué de bric et de broc. Le répertoire `lib` devient un sacré fourre-tout, et c'est réellement compliqué de savoir pour quelle raison nous avons dû introduire `commons-net-1.3.jar` dans le projet. Aucune de nos classes ne fait référence à ce package !

Lorsqu'on fait appel à une bibliothèque pour prendre en charge certaines fonctions techniques, il est rare qu'elle se suffise à elle-même. Au même titre que notre projet, elle fait appel à d'autres bibliothèques spécialisées pour lui fournir des composants de haut niveau qui lui facilitent la tâche. Sa documentation précise, bien évidemment, ces prérequis, ce qui nous a permis lors de son introduction dans le projet de connaître la liste de bibliothèques à ajouter pour avoir un ensemble fonctionnel. Certaines étaient déjà intégrées, et il a fallu nous assurer que la version demandée était compatible avec celle que nous utilisions et, éventuellement, faire la mise à jour qui s'imposait.

Les bibliothèques de haut niveau, telles que le framework Spring, introduisent dans le projet un nombre important de bibliothèques. Les choses se compliquent lorsqu'on désire changer de version pour profiter de nouvelles fonctionnalités ou d'un correctif. Nous devons retracer à la main la chaîne complète des bibliothèques pour identifier ce qui a changé, en nous fondant sur la documentation respective de chaque bibliothèque rencontrée pour connaître ses prérequis et ses éventuelles incompatibilités.

Pour nous épargner une migraine, les développeurs de bibliothèques ont heureusement pris la bonne habitude de ne jamais briser la compatibilité avec les versions précédentes sans un avertissement visible. La pratique la plus courante consiste à utiliser le numéro de version et à passer à une version "majeure" supérieure. Entre la version 1.4 et la version 2.0, il est assez probable que des modifications lourdes ont été apportées, limitant fortement la compatibilité, ce qui justifie le changement de version. Par contre, nous pouvons être plus confiants dans une migration vers une 1.4.2 ou une 1.5, et relâcher (dans la limite du raisonnable) notre surveillance pour passer d'une 1.4.2 à une 1.4.3.

Malgré cette pratique courante, la gestion de la chaîne de dépendances entre bibliothèques peut devenir réellement complexe, si on ne veut oublier personne en route. L'absence d'une bibliothèque peut provoquer des erreurs non évidentes et qui n'apparaîtront pas nécessairement au premier démarrage de l'application. Quant à lire attentivement la documentation de chaque bibliothèque, aucun développeur ne trouve le courage de le faire systématiquement.

La plupart du temps, on se contente donc de prendre la distribution binaire de la bibliothèque et de fusionner son répertoire `lib` avec celui du projet, en tentant d'identifier les doublons. Même si cela fonctionne relativement bien dans de nombreux cas, il est certain qu'on part au petit bonheur la chance en espérant ne rien laisser traîner en route.

L'ami de mon ami…

Que propose Maven pour cette situation ? Nous avons vu qu'il demande de déclarer les dépendances plutôt que de fournir nous-mêmes les binaires ; aussi, notre dernière

option – prendre la distribution telle quelle et la fusionner avec notre répertoire `lib` – n'est pas applicable. Maven va-t-il nous obliger à éplucher la documentation de chaque bibliothèque utilisée ?

Maven est autrement plus subtil : jetez un coup d'œil quelques pages en arrière, sur le contenu du répertoire `lib` de notre projet initial :

```
\lib
    \mail.jar
    \activation.jar
    \util.jar
```

Nous utilisons trois bibliothèques, la première est l'API JavaMail, la deuxième le Bean Activation Framework, nécessaire au bon fonctionnement de JavaMail, et enfin le mystérieux `util.jar` qui s'est avéré être Apache commons-io. Le fichier POM.xml ne compte que deux entrées `<dependency>`, là où notre projet nécessite trois bibliothèques. Jason aurait-il été un peu trop vite ?

Si vous jetez à nouveau un œil aux traces de téléchargement dont Maven nous a abreuvés au premier lancement, vous constaterez qu'il télécharge à la fois des fichiers `POM` et des fichiers `JAR` – comme s'il ne téléchargeait pas déjà assez de choses !

Ces fichiers POM, au même titre que celui de notre projet, décrivent les bibliothèques auxquelles ils sont associés. Pour JavaMail, l'archive `mail-1.4.jar` est ainsi accompagnée d'un `mail-1.4.pom`. Il s'agit bien d'un fichier Project Object Model, au même format XML que pour notre projet et qui comprend des déclarations comparables, en particulier des dépendances. C'est ici qu'est indiqué le lien entre JavaMail et le Bean Activation Framework. Cela permet à Maven de savoir que tout projet qui utilisera l'API JavaMail aura nécessairement besoin du `JAR activation`. Si celui-ci a aussi des dépendances, la chaîne se poursuivra, jusqu'à ce qu'un graphe complet de bibliothèques interdépendantes soit construit.

On parle pour ces données qui décrivent la bibliothèque de "métadonnées". Il s'agit d'une version compacte et normalisée au format POM des informations que nous aurions pu obtenir en lisant la documentation de la bibliothèque : sa licence, le site web qui l'héberge, et ses prérequis. L'exploitation automatisée de ces données permet à Maven de construire l'arbre des dépendances du projet, chaque nouvelle feuille pouvant, par ses propres métadonnées, introduire de nouvelles branches.

Cet arbre, extrêmement difficile à construire à la main et douloureux à maintenir, est analysé automatiquement par Maven à chaque exécution. Il s'assure que l'ensemble des bibliothèques nécessaires est présent et construit ainsi le chemin de classes utilisé par le compilateur. Maven va également gérer les problèmes de conflit de version, lorsque l'arbre fait apparaître plusieurs fois la même bibliothèque dans des versions différentes.

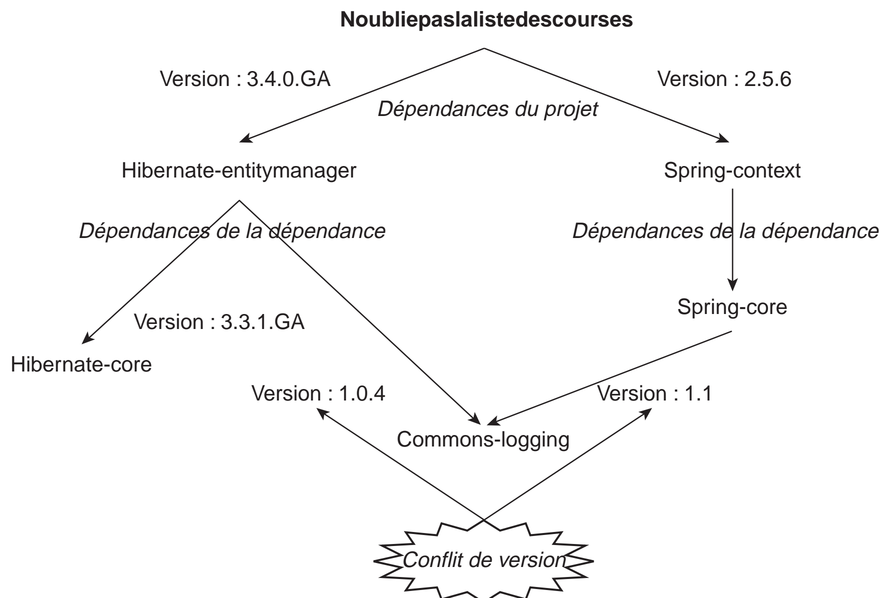

Figure 2.2

Arbre de dépendances transitives.

Le mécanisme utilisé est cependant limité par la liberté laissée aux numéros de version qui rend délicat une comparaison 100 % déterministe.

INFO

L'algorithme de résolution des conflits se fonde sur le principe de "proximité" : Maven compte, dans l'arbre des dépendances, combien de branches séparent la bibliothèque du projet. Celle qui est déclarée au plus près gagne, et, en cas d'égalité, la plus récente l'emporte – sur la base d'une comparaison des numéros de version. Les versions futures de Maven intégreront un mécanisme configurable de dépendance, qui permettra de choisir une politique de gestion de conflit, par exemple pour faire face à des numéros de version exotiques pour lesquels Maven est incapable d'effectuer correctement une comparaison.

Cette dernière fonctionnalité finit par nous convaincre définitivement. Aussi, nous abandonnons nos différents scripts et adoptons les conventions de Maven pour la suite du développement de *noubliepaslalistedescourses*. Les développeurs sont nombreux à choisir Maven pour sa gestion des dépendances. N'oubliez pas, cependant, tous les points que nous avons déjà vus, et en quoi cela différencie Maven d'autres outils de construction de projet. Maven n'est pas juste un outil de gestion des dépendances, pour

lesquelles il existe d'autres très bons outils comme Apache Ivy qui sont utilisables depuis un script Ant.

Ayez bien en tête les points forts et la philosophie de Maven, si vous envisagez de convertir un projet existant, car vous devrez probablement en repenser l'organisation, et pas juste écrire quelques fichiers POM pour déclarer vos dépendances.

Testons un peu

Vincent est un fanatique de la qualité logicielle, aussi a-t-il fait un gros travail d'évangélisation pour nous convaincre d'outiller notre projet de tests automatisés (nous en reparlerons au Chapitre 4). Ceux-ci permettent de contrôler à tout moment que les fonctionnalités de notre projet ne sont pas impactées par une modification, ce qui constitue une sécurité et un gain de temps appréciables.

Nous étions sur le point de décerner à Vincent le prix très convoité de "développeur du mois", quand nous avons rencontré un bogue étrange sur l'application, signalant l'absence de la classe `org.junit.Assert` dans l'environnement d'exécution. Voilà un problème bien curieux.

Après une rapide recherche, nous constatons qu'une erreur d'import dans une classe a fait utiliser `org.junit.Assert#assertNotNull()` à la place de la classe similaire de Spring `org.springframework.util.Assert#notNull()`. La gestion automatique des imports par notre environnement de développement intégré est bien pratique mais elle peut parfois avoir des effets pervers[2]. Comment se fait-il que cette erreur d'étourderie soit passée au travers des mailles de notre (excellent) suivi qualité ? Ou plutôt, comment se fait-il que notre outillage qualité ait pu ajouter des bogues à notre application ?

La réponse tient en un mot : dépendances. Notre gestion des dépendances à la hussarde, avec un répertoire `lib` dont nous utilisons tous les `JAR` sans distinction, ne sait pas différencier les bibliothèques nécessaires à la compilation de celles utilisées par les outils de test.

Nous pourrions fiabiliser les choses en séparant nos bibliothèques en `/lib/runtime` et `/lib/test`, mais Jason nous arrête net : que penser des API servlet, que nous utilisons pour compiler notre interface de gestion web (l'application a pas mal évolué depuis le prototype en ligne de commande !). Ces bibliothèques sont nécessaires pour compiler mais elles ne doivent pas être intégrées à l'application pour respecter les règles JEE, car elles sont déjà présentes dans notre serveur d'application.

---

2. Ne riez pas, il s'agit d'un cas bien réel, identifié lors de la migration du projet sous Maven !

Cela se complique. Peut-être qu'avec un troisième sous-répertoire dans `/lib`… Stop ! Fini de jouer, interrompt Jason, avant de nous expliquer comment Maven traite de manière globale ce problème.

Les "scopes"

Notre problème vient de la nécessité d'associer à chaque dépendance du projet le contexte dans lequel elle doit intervenir. S'agit-il d'un élément indispensable à l'exécution du logiciel ? Est-il utilisé uniquement à des fins de test ? Doit-il être inclus dans l'application ou est-il intégré dans l'environnement d'exécution ?

La déclaration d'une dépendance Maven permet de définir un élément supplémentaire, le "scope" dans lequel la dépendance devra s'appliquer. Nous pouvons ainsi préciser que la bibliothèque jUnit n'est utilisée que durant la phase de test et que l'API `servlet` ne doit pas être intégrée dans notre archive web.

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>servlet</artifactId>
  <version>2.3</version>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.7</version>
  <scope>test</scope>
</dependency>
```

Maven exploite ces données supplémentaires lorsqu'il exécute les commandes de construction du projet. Avec cette précision, jUnit ne sera pas inclus sur la liste des bibliothèques référencées par la commande de compilation. Maven aurait ainsi identifié notre bogue immédiatement.

Une arme à double tranchant

La facilité avec laquelle Maven permet de marier les bibliothèques, gérant dépendances et conflits, nous retire une (autre) sacrée épine du pied. Nous pouvons ainsi nous focaliser sur le développement du logiciel sans être freiné par d'ennuyeux problèmes techniques liés aux bibliothèques.

Notre application de gestion de liste de courses s'étoffe donc rapidement. Elle contient désormais des frameworks de toutes sortes comme Spring, Hibernate, Apache CXF ou Wicket pour prendre en charge les différents aspects de notre architecture.

Maven construit pour nous l'archive web WAR de l'application que nous pouvons déployer sur notre serveur de test.

Un coup d'œil au répertoire `WEB-INF/lib` de l'application web nous fait cependant déchanter : plus de quarante bibliothèques s'y trouvent (qui a demandé tout ça ?). Il y a, par exemple, la bibliothèque `avalon`, un framework ancien que plus personne n'utilise. Plus grave, nous trouvons dans ce répertoire des bibliothèques redondantes, comme un `commons-logging-1.0.4` et un `commons-logging-api-1.1`. Voilà qui est bien troublant. Maven se serait-il emmêlé les pinceaux dans ses dépendances ?

La réponse à nos interrogations est cependant simple : Maven n'est pas un magicien et il ne peut gérer les dépendances entre bibliothèques que grâce aux métadonnées qu'il extrait des fichiers POM de chacune. La qualité de ces informations est déterminante pour obtenir une gestion fine et sans accrocs des dépendances. Il arrive malheureusement qu'une bibliothèque déclare des dépendances qui ne sont pas indispensables à son fonctionnement, ou bien propose plusieurs variantes. Dans ce cas, Maven a bien du mal à s'y retrouver.

La bibliothèque `commons-logging` en est une bonne illustration. Il s'agit d'une bibliothèque qui sert de façade pour passer de manière transparente d'un outil de log à un autre, par exemple de `log4j` au mécanisme intégré dans java à partir de la version 1.4, ou encore à `logkit`, un autre outil comparable.

Le fichier POM de `commons-logging` déclare donc des dépendances vers toutes les bibliothèques de log qu'il supporte. La déclaration Maven correcte devrait être :

```
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.12</version>
  <optional>true</optional>
</dependency>
<dependency>
  <groupId>logkit</groupId>
  <artifactId>logkit</artifactId>
  <version>1.0.1</version>
  <optional>true</optional>
</dependency>
...
```

L'élément `<optional>` permet de préciser la version de la bibliothèque pour laquelle le projet a été conçu, mais que son utilisation n'est pas nécessaire et ne correspond qu'à un cas particulier. Pour le malheur de nombreux utilisateurs, les développeurs de commons-logging ont cependant "oublié" de préciser ce caractère optionnel jusqu'à la version 1.1.1. C'est pour cette raison que nous retrouvons `avalon-framework-4.1.3.jar` dans nos bibliothèques.

Info

La qualité des métadonnées a longtemps été un point faible de Maven, qui se corrige heureusement avec le temps et les nouvelles versions des bibliothèques incriminées. Les projets, même ceux qui n'utilisent pas Maven pour leurs propres besoins, sont aujourd'hui sensibilisés à ce besoin et prennent plus de soin à définir des dépendances fiables.

Pour les versions anciennes cependant, une mise à jour n'est pas possible, car la politique de l'équipe qui gère le dépôt de bibliothèques de référence est de ne jamais modifier un POM qui a été publié, en raison du grand nombre de miroirs et de caches utilisés par la communauté : un fichier modifié signifierait qu'un miroir pourrait ne pas fournir la même version que le dépôt de référence, ce qui pourrait introduire des bogues insurmontables dans les projets. Sans compter que chaque utilisateur devrait manuellement purger son dépôt local pour forcer Maven à récupérer la version corrigée !

Maven possède heureusement une solution de contournement. Lorsque nous définissons une dépendance, nous pouvons exclure certains éléments de la transitivité. Ainsi, si nous voulons empêcher Spring – qui utilise commons-logging – d'introduire sur notre projet ce fameux `JAR` `avalon-framework`, nous pouvons écrire :

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-beans</artifactId>
  <version>2.5.6</version>
  <exclusions>
    <exclusion>
      <groupId>avalon-framework</groupId>
      <artifactId>avalon-framework</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Un autre problème que nous avons identifié est ce doublon `commons-logging + commons-logging-api`. Ce cas est plus subtil. Les développeurs de commons-logging proposent à leurs utilisateurs une série de classes qui masquent d'autres outils de log. Leur projet est donc séparé en une API et des adaptateurs vers chaque outil supporté. Pour éviter à certains d'utiliser par mégarde une classe d'un adaptateur et pas seulement celles de l'API, ils ont mis à disposition une archive `JAR` ne contenant que les classes utilisables par les développeurs : `commons-logging-api`.

Là où les choses se corsent, c'est que ces mêmes classes se retrouvent dans le commons-logging classique, et Maven n'est pas en mesure de le deviner. Aussi, de son point de vue, il s'agit de deux dépendances indépendantes pour lesquelles aucune règle de conflit ne peut s'appliquer.

Il n'existe malheureusement aucune solution miracle pour indiquer qu'une bibliothèque donnée est en réalité un sous-ensemble d'une autre et gérer des conflits de

version entre elles. Une gestion propre du développement de la bibliothèque aurait dû aboutir à la séparation de `commons-logging-api` et d'un second artefact complémentaire, mais pas redondant. À vouloir répondre aux demandes contradictoires des utilisateurs (un seul `JAR` avec tout le nécessaire, des `JAR` focalisés sur un aspect donné…) le projet perd en cohésion et les utilisateurs rencontrent au final des difficultés qu'on aurait dû leur épargner.

Nous avons vu que les `<exclusions>` permettent de corriger ces erreurs de métadonnées. Reste à faire le tri dans nos dépendances pour trouver ces erreurs. Dans notre cas, nous voulons conserver `commons-logging` et exclure `commons-logging-api`, mais aucune solution automatique n'est possible.

L'analyse des dépendances

Avec le nombre de frameworks que nous avons intégrés à l'application, il devient difficile de savoir qui introduit quelle dépendance. Même si les exclusions peuvent permettre de corriger le tir, encore faut-il savoir sur quelles dépendances les déclarer.

Maven propose un outillage complet pour analyser nos dépendances, *via* les plugins `dependency` et `project-info-reports`.

La commande `mvn dependency:list` permet d'établir la liste des dépendances du projet, soit l'équivalent de notre répertoire `WEB-INF/lib`. Elle sera utile pour vérifier l'impact de nos exclusions au fur et à mesure que nous les déclarerons.

La commande `project-info-reports:dependencies` est analogue à la précédente mais elle génère un fichier HTML contenant la liste des dépendances.

Plus intéressante, `mvn dependency:tree` trace un arbre, où chaque branche est une dépendance qui introduit par transitivité d'autres dépendances.

Listing 2.1 : Exécution de `mvn dependency:tree`

```
[INFO] [dependency:tree]
[INFO] fr.noubliepaslalistedescourses: noubliepaslalistedescourses:war:1.0.0-SNAPSHOT
[INFO] +- org.apache.cxf:cxf-rt-frontend-jaxws:jar:2.1.4:compile
[INFO] |  +- org.apache.geronimo.specs:geronimo-jaxws_2.1_spec:jar:1.0:compile
[INFO] |  +- org.apache.geronimo.specs:geronimo-ws-metadata_2.0_spec:jar:1.1.2:compile
[INFO] |  +- asm:asm:jar:2.2.3:compile
[INFO] |  +- org.apache.cxf:cxf-rt-bindings-xml:jar:2.1.4:compile
[INFO] |  +- org.apache.cxf:cxf-rt-frontend-simple:jar:2.1.4:compile
[INFO] |  +- org.apache.cxf:cxf-rt-ws-addr:jar:2.1.4:compile
[INFO] |  +- javax.xml.soap:saaj-api:jar:1.3:compile
[INFO] |  \- com.sun.xml.messaging.saaj:saaj-impl:jar:1.3.2:compile
```

```
[INFO] |  \- javax.xml.ws:jaxws-api:jar:2.1:compile
[INFO] |     +- javax.annotation:jsr250-api:jar:1.0:compile
[INFO] |     \- javax.jws:jsr181-api:jar:1.0-MR1:compile
[INFO] +- org.springframework:spring-aspects:jar:2.5.6:compile
[INFO] |  \- org.aspectj:aspectjweaver:jar:1.6.2:compile
[INFO] +- org.hibernate:hibernate-annotations:jar:3.4.0.GA:compile
[INFO] |  \- org.hibernate:ejb3-persistence:jar:1.0.2.GA:compile
...
```

L'analyse de cet arbre permet d'identifier les bibliothèques qui font appel à `commons-logging-api` et d'exclure cet intrus de notre projet.

Ces commandes bien pratiques restent assez spartiates, cantonnées dans la console. L'intégration de Maven dans les environnements de développement en offre une version nettement plus ergonomique. La Figure 2.3 présente le plugin Maven pour Eclipse (`m2eclipse`) et sa fenêtre d'analyse des dépendances. Si on sélectionne une dépendance dans la zone de droite, il nous indique tous les chemins de dépendance qui y mènent. Un simple clic permet de placer les exclusions qui s'imposent sans éditer manuellement le fichier POM.

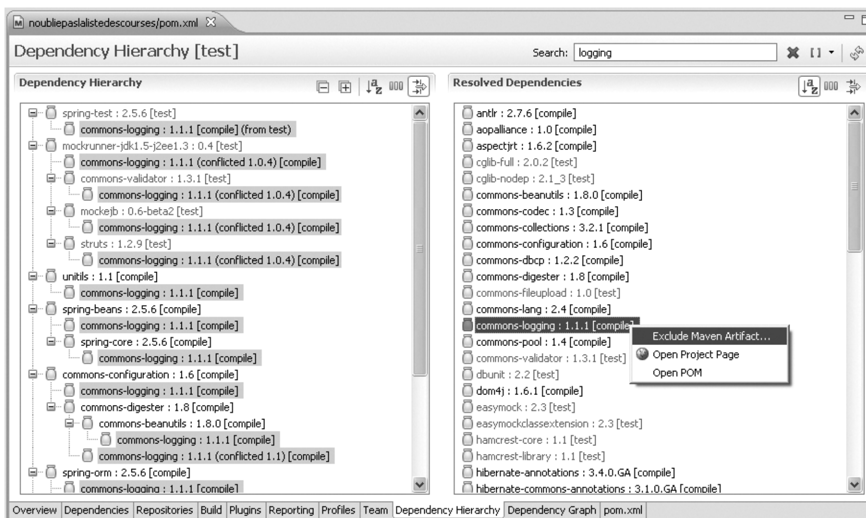

Figure 2.3

Plugin Maven pour Eclipse.

Conclusion

La gestion des bibliothèques et de leurs dépendances est une fonctionnalité de Maven très régulièrement mise en avant. Manuellement, cette gestion peut en effet devenir un réel casse-tête, et la réponse apportée par Maven est à la fois élégante et évolutive. Sur de gros projets, nous avons vu que celle-ci peut cependant déraper et introduire involontairement des bibliothèques inutiles ou redondantes mais, heureusement, Maven permet de corriger ces problèmes. La qualité des métadonnées est donc primordiale, pensez-y si vous participez à un projet qui diffuse ses binaires sur un dépôt Maven.