Maven

Mireille Khalifeh

Architectures Logicielles Java

Sommaire

INTRODUCTION À MAVEN

La réponse à cette question dépend de votre point de vue. La plus grande partie des utilisateurs de Maven vont l'appeler un “outil de build” : c'est-à-dire un outil qui permet de produire des artefacts déployables à partir du code source. Pour les gestionnaires de projet et les ingénieurs en charge du build, Maven ressemble plus à un outil de gestion de projet. Quelle est la différence ? Un outil de build comme Ant se concentre essentiellement sur les tâches de prétraitement, de compilation, de packaging, de test et de distribution. Un outil de gestion de projet comme Maven fournit un ensemble de fonctionnalités qui englobe celles d'un outil de build. Maven apporte, en plus de ses fonctionnalités de build, sa capacité à produire des rapports, générer un site web et ainsi facilite la communication entre les différents membres de l’équipe. Voici une définition plus formelle d'Apache Maven : Maven est un outil de gestion de projet qui comprend un modèle objet pour définir un projet, un ensemble de standards, un cycle de vie, et un système de gestion des dépendances. Il embarque aussi la logique nécessaire à l'exécution d'actions pour des phases bien définies de ce cycle de vie, par le biais de plugins. Lorsque vous utilisez Maven, vous décrivez votre projet selon un modèle objet de projet clair, Maven peut alors lui appliquer la logique transverse d'un ensemble de plugins (partagés ou spécifiques). Ne vous laissez pas impressionner par le fait que Maven est un "outil de gestion de projet". Si vous cherchiez juste un outil de build alors Maven fera l'affaire.

POURQUOI UTILISER MAVEN ?

Nous avons vu qu'il est possible de créer et d'exécuter des projets Java EE directement dans Eclipse. Alors pourquoi utiliser Maven ? Si un IDE peut suffire pour gérer des projets simples, cette solution s'avère rapidement limitée :

Comment partager mon projet avec d'autres développeurs quand ils n'ont pas exactement la même configuration de poste que la mienne et qu'ils n'utilisent pas le même IDE que moi ?
Comment compiler et tester mon projet en dehors d'un IDE (par exemple dans un processus d'intégration continue) ?
Comment automatiser certaines tâches répétitives et limiter ainsi les erreurs ou les oublis ?
Pour résoudre tous ces problèmes (et d'autres encore), le plus simple est d'utiliser un outil tel que Maven.

Caractéristiques principales de Maven

L'outil de construction de projet le plus célèbre est sans aucun doute make. Make permet de définir des tâches avec des commandes associées et des dépendances entre ces tâches.

Maven part d'une approche très différente : il découpe le cycle de construction du projet en phases pré-définies et le développeur peut paramétrer ou ajouter des tâches à effectuer automatiquement pour chacune des phases. Les principales phases dans Maven sont :

compile : compilation du code source du projet
test : compilation du code source des tests et exécution des tests
package : construction du livrable (pour une application Web, il s'agit de l'archive WAR)
Maven ajoute la possibilité de gérer automatiquement les dépendances logicielles. Pour développer des applications Java EE, nous allons avoir besoin de bibliothèques externes (les fichiers .jar en Java). Plutôt que d'aller les télécharger une à une depuis le Web et de les ajouter dans Eclipse, nous allons signaler à Maven l'identifiant des dépendances dont nous aurons besoin et il va se charger pour nous de les télécharger depuis un référentiel centralisé (Maven central repository), de les stocker dans un cache sur la machine et de les associer à notre projet.

Enfin les concepteurs de Maven ont adopté une approche normative afin de garantir une homogénéité entre les projets. Ainsi un projet Maven se conforme à une organisation assez stricte des répertoires et des fichiers.

Il existe d'autres outils de construction de projet en Java. Ant, également promu par la communauté Apache, a été (et reste) beaucoup utilisé. Ant suit un principe très proche de Make. Gradle est un autre outil de construction bien connu des développeurs Android.

Le Modèle conceptuel d'un "Projet"

Avec Maven vous modélisez un projet. Vous ne faites plus simplement de la compilation de code en bytecode, vous décrivez un projet logiciel et vous lui assignez un ensemble unique de coordonnées. Vous définissez les attributs qui lui sont propres. Quelle est sa licence ? Quels sont ses développeurs et ses contributeurs ? De quels autres projets dépend-il ? Maven est plus qu'un simple "outil de build", c'est plus qu'une amélioration des outils tels que Ant et make, c'est une plateforme qui s'appuie sur de nouvelles sémantiques pour les projets logiciels et le développement. La définition d'un modèle pour tous les projets fait émerger de nouvelles caractéristiques telles que :

La gestion des dépendances

Puisque chaque projet est identifié de manière unique par un triplet composé d'un identifiant de groupe, un identifiant d'artefact et un numéro de version, les projets peuvent utiliser ces coordonnées pour déclarer leurs dépendances.

Des dépôts distants

En liaison avec la gestion de dépendance, nous pouvons utiliser les coordonnées définies dans le Project Object Model (POM) de Maven pour construire des dépôts d'artefacts Maven.

Réutilisation universelle de la logique de build

Les plugins contiennent toute la logique de traitement. Ils s'appuient sur les données et paramètres de configuration définit dans le Project Object Model (POM). Ils ne sont pas conçus pour fonctionner avec des fichiers spécifiques à des endroits connus.

Portabilité / Intégration dans des outils

Les outils tels qu'Eclipse, NetBeans, et IntelliJ ont maintenant un endroit unique pour aller récupérer les informations sur un projet. Avant Maven, chaque EDI conservait à sa manière ce qui était, plus ou moins, son propre Project Object Model (POM). Maven a standardisé cette description, et alors que chaque EDI continue à maintenir ses propres fichiers décrivant le projet, ils peuvent être facilement générés à partir du modèle.

Facilités pour la recherche et le filtrage des artefacts d'un projet

Des outils tels que Nexus vous permettent d'indexer et de rechercher les contenus d'un dépôt à partir des informations contenues dans le POM.

Installation de Maven

Plutôt que de supposer que vous sachiez comment installer un logiciel et définir des variables d'environnement, voici une installation détaillés afin de réduire le nombre de problèmes que vous pourriez rencontrer suite à une installation partielle. La seule chose que nous allons supposer est que vous avez déjà installé un JDK (Java Development Kit) approprié. Si vous n'êtes intéressé que par l'utilisation elle-même, vous pouvez passer à la suite du livre après avoir lu les paragraphes "Téléchargement de Maven" et "Installer Maven".

Vérifier votre installation de Java

Utilisez la version stable la plus récente du JDK (Java Development Kit) disponible pour votre système d'exploitation.

% **java -version**

java version "1.8.0_121"

Java(TM) SE Runtime Environment (build 1.8.0_121-b13)

Java HotSpot(TM) 64-Bit Server VM (build 25.121-b13, mixed mode)

Maven fonctionne avec tous les kits de développement Java TM certifiés compatibles et également avec certaines implémentations de Java non certifiées. Les exemples fournis avec ce livre ont été écrits et testés avec les versions officielles du Java Development Kit téléchargées depuis le site internet de Sun Microsystems. Si vous êtes sous GNU/Linux, vous pouvez avoir besoin de télécharger le JDK de Sun vous-même et vérifier qu'il s'agit de la version que vous utilisez (en exécutant la commande java - version). Maintenant que Sun a mis Java en Open Source, cela devrait s'améliorer et nous devrions rapidement trouver la JRE et le JDK de Sun par défaut même dans les distributions puristes de GNU/Linux. En attendant ce jour, vous pouvez avoir à le télécharger.

Téléchargement de Maven

Vous pouvez télécharger Maven depuis le site internet du projet Apache Maven avec l'url suivante http://maven.apache.org/download.html.

Lorsque vous téléchargez Maven, faites attention à choisir la dernière version de Apache Maven disponible sur le site de Maven. Si l'Apache Software License ne vous dit rien, nous vous suggérons de vous familiariser avec http://maven.apache.org/download.html les termes de cette licence avant de commencer à utiliser le produit.

Installer Maven

Il existe de grandes différences entre un système d'exploitation comme Mac OS X et Microsoft Windows, il en est de même entre les différentes versions de Windows. Heureusement, le processus d'installation de Maven est relativement simple et sans douleur sur ces systèmes d'exploitation. Les sections qui suivent mettent en relief les bonnes pratiques pour installer Maven sur une palette de systèmes d'exploitation.

Installer Maven sur Mac OSX

Vous pouvez télécharger une version binaire de Maven depuis http://maven.apache.org/download.html.

Téléchargez la version courante de Maven dans le format qui vous convient le mieux. Choisissez l'emplacement où vous voulez l'installer et décompressez l'archive à cet endroit. Si vous avez décompressé l'archive dans le répertoire /usr/local/apache-maven-3.3.9, vous pouvez vouloir créer un lien symbolique afin de vous faciliter la vie et éviter d'avoir à modifier l'environnement.

```
/usr/local % cd /usr/local
/usr/local % ln -s apache-maven-3.3.9 maven
/usr/local % export M2_HOME=/usr/local/maven
/usr/local % export PATH=${M2_HOME}/bin:${PATH}
```

Une fois Maven installé, il vous reste quelques petites choses à faire afin de pouvoir l'utiliser correctement. Vous devez ajouter le répertoire bin de la distribution (dans notre exemple /usr/local/maven/bin) au PATH de votre système. Vous devez aussi positionner la variable d'environnement M2_HOME sur le répertoire racine de votre installation (pour notre exemple, /usr/local/maven).
Vous allez devoir ajouter les variables M2_HOME et PATH à un script qui sera exécuté à chaque fois que vous vous connecterez à votre système. Pour cela, ajouter les lignes suivantes au .bash_login.

```
export M2_HOME=/usr/local/maven
export PATH=${M2_HOME}/bin:${PATH}
```

Maintenant que vous avez mis à jour votre environnement avec ces quelques lignes, vous pouvez exécuter Maven en ligne de commande.
Pour ces instructions d'installation nous avons supposé que vous utilisiez le shell bash.

Installer Maven sur Microsoft Windows

L'installation de Maven sur Windows est très proche de celle sur Mac OSX. Les principales différences sont le répertoire d'installation et la définition des variables d'environnement. Cela ne change pas grand-chose si vous installez Maven dans un autre répertoire, tant que vous configurez correctement les variables d'environnement. Une fois que vous avez décompressé Maven dans le répertoire d'installation, il vous faut définir deux variables d'environnement : PATH et M2_HOME. En ligne de commande, vous pouvez définir ces variables d'environnement en tapant les instructions suivantes :

```
C:\Users\user >set M2_HOME=c:\Program Files\apache-maven-2.2.1
C:\Users\user >set PATH=%PATH%;%M2_HOME%\bin
```

Définir ces variables d'environnement en ligne de commande va vous permettre d'exécuter Maven dans la même console. Mais à moins de les définir comme des variables du Système avec le Panneau de Configuration, vous devrez exécuter ces deux lignes à chaque fois que vous vous connecterez à votre système. Vous devriez modifier ces deux variables avec le Panneau de Configuration de Microsoft Windows.

Installer Maven sur GNU/Linux

Pour installer Maven sur une machine GNU/Linux, suivez la même procédure « Installer Maven sur Mac OSX ».

Tester une installation Maven

Une fois Maven installé, vous pouvez vérifier sa version en exécutant la commande suivante **mvn -v** en ligne de commande. Si Maven a été correctement installé, vous devriez voir en sortie quelque chose ressemblant à cela.

$mvn -v

Apache Maven 3.3.9 (bb52d8502b132ec0a5a3f4c09453c07478323dc5; 2015-11-10T18:41:47+02:00)
Maven home: /Users/user/Downloads/apache-maven-3.3.9
Java version: 1.8.0_121, vendor: Oracle Corporation
Java home: /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/jre
Default locale: en_US, platform encoding: UTF-8
OS name: "mac os x", version: "10.12", arch: "x86_64", family: “mac"

Si vous obtenez cette sortie, vous savez que Maven est disponible et prêt à être utilisé. Si vous n’obtenez pas cette sortie et que votre système d'exploitation ne peut pas trouver la commande mvn, vérifiez que vos variables d'environnement PATH et M2_HOME sont correctement définies.

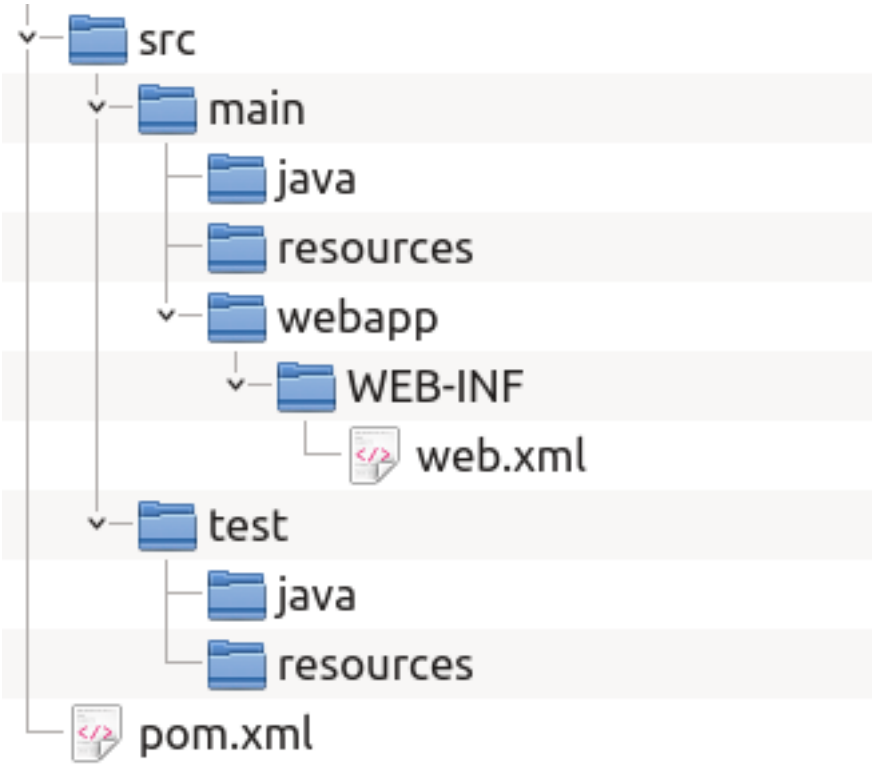
Désinstaller Maven

La plupart des instructions d'installation demandent de décompresser l'archive de la distribution de
Maven dans un répertoire et de définir quelques variables d'environnement. Si vous devez supprimer
Maven de votre ordinateur, tout ce que vous avez à faire est de supprimer le répertoire d'installation de
Maven ainsi que les variables d'environnement. Vous voudrez sûrement supprimer le répertoire ~/.m2
qui contient votre dépôt local

Un premier projet avec Maven

Un projet Maven possède toujours un fichier pom.xml à la racine du projet. Ce fichier XML est le descripteur du projet et contient toutes les informations nécessaires à Maven pour gérer le cycle de vie du projet.

Le nom du fichier pom.xml vient de POM (Project Object Model).
Maven impose une arborescence minimale des fichiers afin de garantir une homogénéité entre tous les projets.



pom.xml
A la racine du projet, on trouve le fichier pom.xml, le descripteur du projet pour Maven.
src/main
Ce répertoire contient les fichiers de l'application. On trouve au moins le sous répertoire java contenant les sources Java. Le sous répertoire resources accueille les fichiers qui ne sont pas des sources Java mais qui doivent être présents avec les fichiers compilés dans l'application finale (il s'agit souvent de fichiers de configuration).
src/test
Ce répertoire contient les fichiers utilisés pour tester l'application. On trouve le sous répertoire Java contenant les sources Java des tests unitaires. Le sous répertoire resources accueille les fichiers qui ne sont pas des sources Java mais qui sont nécessaires à l'exécution des tests (il s'agit souvent de fichiers de configuration pour les tests).
Il existe un dernier répertoire à connaître, le répertoire **target**. Il s'agit du répertoire de travail de Maven. Ce répertoire est créé automatiquement par Maven pour stocker tous les

fichiers de travail. On y trouve les classes compilées, les fichiers sources générés automatiquement, le livrable final, les rapports d'exécution des tests...

Le fichier POM.XML

Le fichier pom.xml est le descripteur de projet pour Maven. Il s'agit d'un fichier XML présent à la racine du projet qui est lu par Maven pour lui fournir les informations du projet.

Le contenu du fichier pom.xml est le suivant :

```
<project
  xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
xsd/maven-4.0.0.xsd">
 <!--
  La version du format du fichier pom.
  Actuellement la dernière version est la 4.0.0.
 -->
 <modelVersion>4.0.0</modelVersion>

 <!--
  Le group ID de l'application. Le group ID
  s'apparente à un package Java mais pour un projet. Il évite
  une colision de nom dans le cas de deux projets ayant le même nom
  puiqu'ils peuvent avoir des group ID différents.
  Ainsi si deux projets s'appelle hello et qu'ils ont des group ID
  différents, ils sont considérés comme étant des projets différents.
 -->
 <groupId>com.demoGroup</groupId>

 <!--
  Le nom du projet
 -->
 <artifactId>hello</artifactId>

 <!--
  La version de notre projet. Maven gère le versionnage
  afin de permettre le suivi des évolutions d'un projet.
  Ici, le suffixe "-SNAPSHOT" indique à Maven que le projet
  est en cours de développement pour cette version.
```

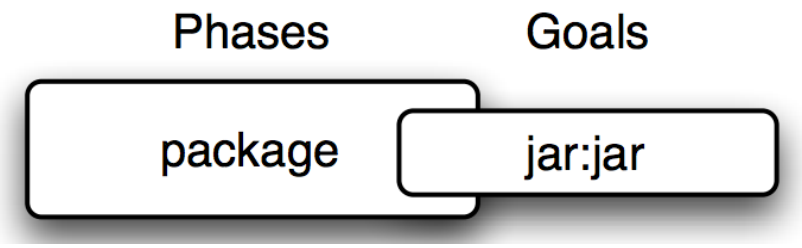
```
  -->
  <version>0.0.1-SNAPSHOT</version>

  <!--
    Le type de packaging, c'est-à-dire le type de projet.
    Ici, on indique à Maven que le projet doit être packagé
    sous la forme d'un WAR. Donc pour Maven, il s'agit d'une
    application Web.
  -->
  <packaging>war</packaging>

  <!--
    Les propriétés de notre projet. On peut définir des propriétés
    spécifiques au projet ou des propriétés standard à Maven pour
    paramétrer la construction du projet.
  -->
  <properties>
    <!--
      Propriété standard définissant la version minimale de Java supportée
      par les fichiers sources (ici 1.8 pour Java 8).
    -->
    <maven.compiler.source>1.8</maven.compiler.source>

    <!--
      Propriété standard définissant la version Java des fichiers compilés
      du projet (ici 1.8 pour Java 8).
    -->
    <maven.compiler.target>1.8</maven.compiler.target>

    <!--
      Le format d'encodage des fichiers source du projet. Attention, l'encodage
      par défaut n'est pas le même sous Windows et sous les systèmes *NIX.
      Il est donc plus sage de toujours positionner cette propriété dans le fichier pom.xml.
    -->
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
</project>
```

Ce fichier pom.xml donne les informations minimales à Maven :

Le projet s'appelle com.demoGroup:hello
La version actuelle est la 0.0.1 et il s'agit d'une version de travail
Le projet est une application Web Java EE (war)
Le projet est écrit en Java 8 et les sources sont encodées en UTF-8

Cycle de vie de Maven

Une phase est une des étapes de ce que Maven appelle "le cycle de vie du build". Le cycle de vie du build est une suite ordonnée de phases aboutissant à la construction d'un projet. Maven peut supporter différents cycles de vie, mais celui qui reste le plus utilisé est le cycle de vie par défaut de Maven, qui commence par une phase de validation de l'intégrité du projet et se termine par une phase qui déploie le projet en production.
Les phases du cycle de vie sont laissées vagues intentionnellement, définies comme validation, test, ou déploiement elles peuvent avoir un sens différent selon le contexte des projets. Par exemple, pour un projet qui produit une archive Java, la phase package va construire un JAR ; pour un projet qui produit une application web, elle produira un fichier WAR. Les goals des plugins peuvent être rattachés à une phase du cycle de vie. Pendant que Maven parcourt les phases du cycle de vie, il exécute les goals rattachés à chacune d'entre elles. On peut rattacher de zéro à plusieurs goals à chaque phase. Dans la section précédente, quand vous avez exécuté **mvn install** atteint la phase package, il exécute le goal jar du plugin Jar. Puisque notre projet Quickstart a (par défaut) un packaging de type jar, le goal jar:jar est rattaché à la phase package.



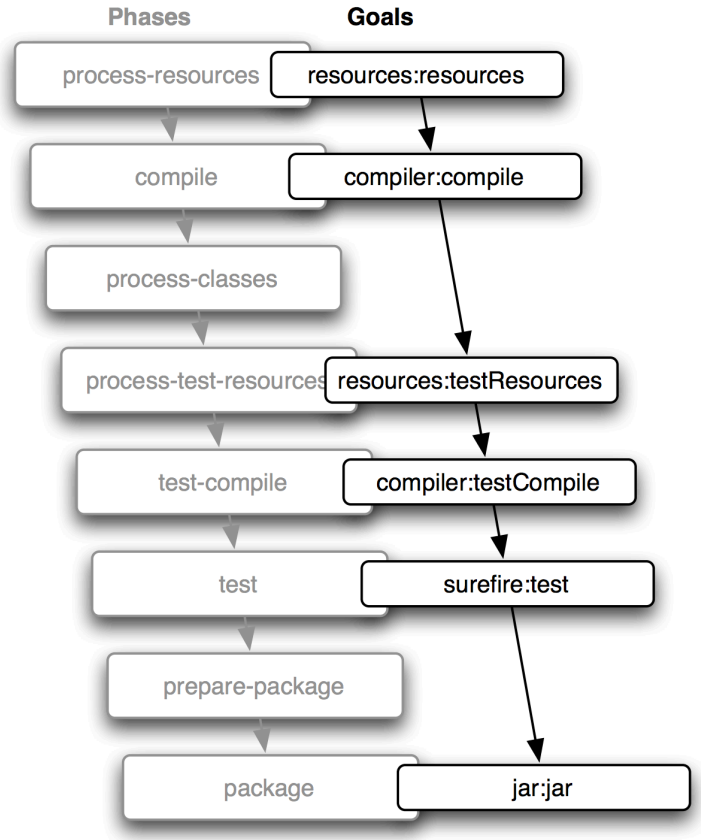
Un goal est rattaché à une phase

Nous savons que la phase package va créer un fichier JAR pour un projet ayant un packaging de type jar. Mais qu'en est-il des goals qui la précèdent, comme compiler:compile et surefire:test ?
Ces goals s'exécutent lors des phases qui précèdent la phase package selon le cycle de vie de Maven ; exécuter une phase provoque l'exécution de l'ensemble des phases qui la précèdent, le tout se terminant par la phase spécifiée sur la ligne de commande. Chaque phase se compose de zéro à plusieurs goals, et puisque nous n'avons configuré ou personnalisé aucun plugin, cet exemple rattache un ensemble de goals standards au cycle de vie par défaut. Les goals suivants ont été exécutés dans l'ordre pendant que Maven parcourait le cycle de vie par défaut jusqu'à la phase package :

resources:resources

Le goal resources du plugin Resources est rattaché à la phase process-resources. Ce goal copie toutes les ressources du répertoire src/main/resources et des autres répertoires configurés comme contenant des ressources dans le répertoire cible.

compiler:compile
Le goal compile du plugin Compiler est lié à la phase compile. Ce goal compile tout le code source du répertoire src/main/java et des autres répertoires configurés comme contenant du code source dans le répertoire cible.
resources:testResources
Le goal testResources du plugin Resources est lié à la phase process-test-resources. Ce goal copie toutes les ressources du répertoire src/test/resources et des autres répertoires configurés comme contenant des ressources de test dans le répertoire cible pour les tests.
compiler:testCompile
Le goal testCompile du plugin Compiler est rattaché à la phase test-compile. Ce goal compile les tests unitaires du répertoire src/test/java et des autres répertoires configurés comme contenant du code source de test dans le répertoire cible pour les tests.
surefire:test
Le goal test du plugin Surefire est rattaché à la phase test. Ce goal exécute tous les tests et produit des fichiers contenant leurs résultats détaillés. Par défaut, le goal arrêtera le build en cas d'échec de l'un des tests.
jar:jar
Le goal jar du plugin Jar est lié à la phase package. Ce goal package le contenu du répertoire cible en un fichier JAR.



Les goals sont lancés à l'exécution de la phase à laquelle ils sont rattachés

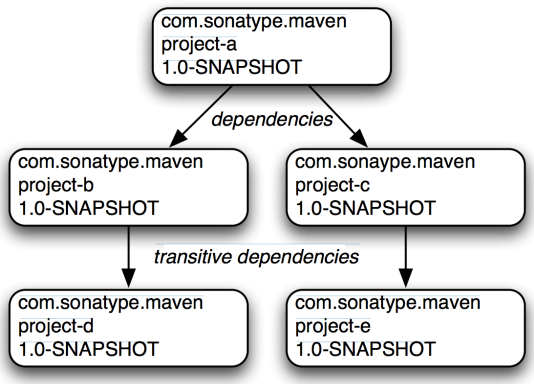
Pour résumer, lorsque nous avons exécuté mvn install, Maven exécute toutes les phases jusqu'à la phase install, et pour cela il parcourt toutes les phases du cycle de vie, exécutant tous les goals liés à chacune de ces phases. Au lieu d'exécuter une des phases du cycle de vie de Maven, vous pourriez obtenir le même résultat en spécifiant une liste de goals de plugin de la manière suivante :

```
mvn resources:resources \
 compiler:compile \
 resources:testResources \
 compiler:testCompile \
 surefire:test \
 jar:jar \
 install:install
```

Il est beaucoup plus simple d'exécuter les phases du cycle de vie plutôt que de lister explicitement les goals à exécuter en ligne de commande, de plus, ce cycle de vie commun permet à chaque projet utilisant Maven d'adhérer à un ensemble de standards bien définis. C'est ce cycle de vie qui permet à un développeur de passer d'un projet à l'autre sans connaître tous les détails du build de chacun d'entre eux. Si vous savez construire un projet Maven, vous savez tous les construire.

La gestion des dépendances de Maven

Si vous examinez le fichier pom.xml du projet simple, vous verrez qu'il comporte une section dependencies pour traiter des dépendances et que cette section contient une seule dépendance — JUnit.
Un projet plus complexe contiendrait sûrement plusieurs dépendances, ou pourrait avoir des dépendances qui dépendent elles-mêmes d'autres artefacts. L'une des principales forces de Maven est sa gestion des dépendances transitives. Supposons que votre projet dépende d'une bibliothèque qui à son tour dépend de 5 ou 10 autres bibliothèques (comme Spring ou Hibernate par exemple).
Au lieu d'avoir à trouver et lister explicitement toutes ces dépendances dans votre fichier pom.xml, vous pouvez ne déclarer que la dépendance à la bibliothèque qui vous intéresse, Maven se chargera d'ajouter ses dépendances à votre projet implicitement. Maven va aussi gérer les conflits de dépendances, vous fournira le moyen de modifier son comportement par défaut et d'exclure certaines dépendances
transitives.
Remarquez que Maven n'a pas juste téléchargé le JAR de JUnit, mais aussi un fichier POM pour les dépendances de JUnit. C'est le téléchargement des fichiers POM en plus des artefacts qui est au cœur de la gestion des dépendances transitives par Maven. Quand vous installez l'artefact produit par votre projet dans le dépôt local, vous noterez que Maven publie une version légèrement modifiée du fichier pom.xml de votre projet dans le répertoire contenant le fichier JAR. Ce fichier POM enregistré dans le dépôt fournit aux autres projets des informations sur ce projet, dont notamment ses dépendances. Si le Projet B dépend du Projet A, il dépend aussi des dépendances du Projet A. Quand Maven résout une dépendance à partir de ses coordonnées, il récupère, en plus de l'artefact, le POM, puis il analyse les dépendances de ce POM pour trouver les dépendances transitives. Ces dépendances transitives sont ensuite ajoutées à la liste des dépendances du projet.
Dans le monde de Maven, une dépendance n'est plus simplement un fichier JAR ; c'est un fichier POM qui à son tour peut déclarer de nouvelles dépendances. Ce sont ces dépendances de dépendances que l'on appelle dépendances transitives et cela est rendu possible par le fait que les dépôts Maven contiennent plus que du bytecode ; ils contiennent des métadonnées sur les artefacts.



Dans le schéma précédent, le projet A dépend des projets B et C. Le Projet B dépend du projet D et le projet C dépend du projet E. L'ensemble des dépendances directes et transitives du projet A serait donc les projets B, C, D et E, mais tout ce que le projet A doit faire, c'est de déclarer ses dépendances aux projets B et C. Les dépendances transitives sont pratiques lorsque votre projet dépend d'autres projets qui ont leurs propres dépendances (comme Hibernate, Apache Struts, ou Spring Framework). Maven vous permet d'exclure certaines dépendances transitives du classpath du projet.
Maven fournit enfin différentes portées pour les dépendances. Le fichier pom.xml du projet simple contient une unique dépendance — junit:junit:jar:3.8.1 — ayant pour portée test indiquée dans la balise scope. Lorsqu'une dépendance Maven a une portée de type test, elle n'est pas disponible pour le goal compile du plugin Compiler. Cette dépendance sera ajoutée au classpath des goals compiler:testCompile et surefire:test.
Durant la création du JAR d'un projet, les dépendances ne sont pas intégrées à l'artefact produit ; elles ne sont utilisées que lors de la compilation. Par contre lorsque vous utilisez Maven pour produire un WAR ou un EAR, vous pouvez le configurer de manière à packager les dépendances avec l'artefact produit et vous pouvez même configurer Maven pour exclure certaines dépendances du fichier WAR par l'utilisation de la portée provided. La portée provided indique à Maven que la dépendance est nécessaire à la compilation, mais qu'elle ne doit pas être intégrée à l'artefact produit par le build. Cette portée est donc très pratique lorsque vous développez une application web. Vous aurez besoin du jar des spécifications Servlet pour compiler, mais vous ne voulez pas inclure le JAR de l'API Servlet dans le répertoire WEB-INF/lib.