Introduction à Maven

Sébastien Nedjar et Fabien Pesci

1 Introduction

L'objectif de ce document est de vous présenter rapidement l'outil de référence pour la gestion et l'automatisation de production des projets logiciels Java : Maven. Au premier abord, Maven semble avoir le même objectif que `Make` sous Unix : la production d'un logiciel à partir de ses sources. Maven est bien plus puissant que cela, par ses possibilités avancés, il est souvent utilisé comme une brique de base pour l'intégration continue et les forges logicielles [1]. Contrairement à ses prédécesseurs, il a une approche moins descriptive de la construction d'un logiciel. Parmi les tâches qu'il aide à automatiser il y a :

- Constructions
- Exécution des tests
- Documentation
- Génération de rapport
- Gestion des dépendances
- Gestion de versions
- Releases
- Distribution

Au lieu d'écrire des scripts particuliers pour chaque projet, Maven essaye de proposer une abstraction standard du cycle de vie d'un logiciel. Sur ce cycle viennent se greffer les ajouts et les particularités propres à chaque projet. Ainsi la configuration d'un nouveau projet sera facile et c'est au fur et à mesure de la complexification des besoins que la configuration devra être adaptée.

Pour arriver à cet objectif de simplicité sans perte de flexibilité, Maven utilise la philosophie "Convention plutôt que configuration". L'idée est de faire décroître le nombre de décisions qu'un développeur doit prendre en lui proposant une convention adaptée au cas d'utilisation le plus classique qu'il pourra amender pour correspondre à ce qu'il veut faire. Ainsi la configuration n'est plus la norme mais l'exception.

1.1 Convention Maven

L'un des grands intérêts de Maven est qu'il encourage la standardisation des pratiques grâce aux conventions qu'il propose. La structure des répertoires d'un projet est la partie la plus visible de cet effort d'uniformisation. Les conventions étant relativement simples et logiques, il y a peu de raison de ne pas les respecter. L'immense avantage est que tous les projets Maven tendant à avoir la même structure, n'importe quel développeur pourra facilement retrouver ses marques dans des projets différents. Voici une liste non-exhaustive des répertoires standard d'un projet Maven :

- `src` : les sources du projet.
- `src/main` : code source et fichiers source principaux.
- `src/main/java` : code source java.
- `src/main/resources` : fichiers de ressources (images, fichiers annexes etc.).
- `src/main/webapp` : webapp du projet.
- `src/main/filters` : Les filtres de ressources, sous forme de fichier de propriétés, qui peuvent être utilisés pour définir des variables connues uniquement au moment du build.

1. http://fr.wikipedia.org/wiki/Forge_%28informatique%29

- `src/test` : fichiers de test.
- `src/test/java` : code source java de test.
- `src/test/resources` : fichiers de ressources de test.
- `src/test/filters` : Les filtres nécessaires aux tests unitaires, qui ne seront pas déployés
- `src/site` : informations sur le projet et/ou les rapports générés suite aux traitements effectués.
- `target` : fichiers résultat, les binaires (du code et des tests), les `jar` générés et les résultats des tests.

1.2 Project Object Model (POM)

Le modèle objet projet ou POM est le fichier central pour la configuration d'un projet avec Maven. Il contient une description détaillée du projet, avec en particulier des informations concernant le versionnage et la gestion des configurations, les dépendances, les ressources de l'application, les tests, les membres de l'équipe, la structure ... Ce POM se matérialise par un fichier pom.xml à la racine du projet. Voici un exemple simple de POM :

```
<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">

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>MSBAI</name>
  <description>Ma Super Belle Application Inutile</description>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding
       >
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.10</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Bien que très simple, ce POM permet de voir les éléments clés dans la définition d'un projet Maven. Chaque élément est décrit par une balise XML reconnaissable par l'usage de chevrons (< >) comme en HTML. Les balises indispensables sont les suivantes :

project Cette balise est la balise de premier niveau dans tous les projets Maven.

modelVersion Cette balise permettent de configurer la version de la syntaxe du POM (à ignorer pour le moment). La version du POM change que très rarement mais il faut impérativement préciser la version pour que les futures versions de Maven puissent continuer à gérer correctement le projet.

groupId Cette balise est un identifiant unique pour l'organisation qui a créé le projet. Le `groupId` est l'un des identifiants clés d'un projet, il est généralement construit à partir du nom de domaine de l'organisation responsable du projet. Par exemple `org.apache.maven.plugins` est le `groupId` de tous les plug-ins Maven.

artifactId Cette balise indique le nom de base unique pour l'artefact principal généré par ce projet. Typiquement en Java cet artefact est un fichier `jar`. Le nom de ce fichier aura la forme suivante `<artifactId>-<version>.<extension>` (`myapp-1.0-SNAPSHOT.jar` pour le projet d'exemple).

version Cet élément indique la version de l'artefact généré par le projet. Permet d'aider à Maven d'aider le développeur à gérer correctement les versions d'un même projet ainsi que les dépendances.

packaging Indique quel est le type d'archive utilisé pour l'artefact principale (`jar`,`war`,`ear`,...). Par défaut Maven génère un `jar`, il est donc inutile de préciser cette balise pour la majorité des projets.

name Indique le nom complet du projet. Il est parfois utile dans la documentation générée grâce à Maven.

url Cette balise indique où est situé le site principal du projet. Comme la précédente cette information sert surtout pour les documents générés par l'outil.

description Cette balise donne une description sommaire du projet. Encore une fois surtout pour la documentation.

1.3 Gestion des dépendances

L'un des premiers bénéfices de Maven que constate le développeur habitué à `Make` ou `Ant` est la gestion simplifiée des dépendances transitives. En effet, Maven étant fortement basé sur le réseau, il est capable de récupérer automatiquement les `jar` dont votre projet dépend ainsi que leurs dépendances. On pourrait comparer cette fonctionnalité de Maven avec les services rendus par un outil comme `apt-get` pour la gestion des paquets sous Debian. Comme pour ces outils de gestion d'installation de logiciels, l'endroit ou sont regroupés les paquets s'appelle un dépôt (ou repository).

Pour ajouter une dépendance à un projet, il suffit de rajouter dans le POM la balise `dependency` avec les bonnes propriétés. Cette balise est imbriquée dans une balise `dependencies` qui regroupe toute les dépendances du projet. Une fois le fichier `pom.xml` enregistré, à la compilation suivante, les `jar` seront automatiquement téléchargés à partir des dépôts. Pour trouver les informations sur les `jar` disponibles dans les dépôts, il existe plusieurs moteurs de recherche (par exemple http://search.maven.org/). L'avantage d'utiliser ces moteurs est qu'ils donnent directement les balises `dependency` à copier dans la section `dependencies` du fichier `pom.xml`.

Par défaut, toute installation de Maven connaît un seul dépôt : *"Central"*. Ce dépôt contient la plupart des bibliothèques Java classiques. Parfois, pour obtenir une version récente ou pour obtenir une bibliothèque exotique, il faudra rajouter des dépôts dans le POM. Cela peut être fait par l'ajout d'une balise `repository` placée dans la section `repositories`.

1.4 Cycle de vie

Par défaut Maven propose une abstraction du cycle de vie standard de la construction d'un projet (appelée en anglais Build Lifecycle). Chaque phase de ce cycle de vie est appelée un but (Goal en anglais). Ces étapes se déroulent dans un ordre déterminée et une étape peut démarrer

uniquement si et seulement si les précédentes se sont bien déroulées. Le cycle par défaut est constitué des étapes de construction suivantes :

validate - Valide que le projet est correct et que toutes les informations nécessaires sont renseignées.

compile - Compile le code source du projet.

test - Teste le code compilé en utilisant un framework de test unitaire.

package - Prend le code compilé et l'empaquette dans une forme distribuable, typiquement un `jar`.

integration-test - Exécute et déploie si nécessaire le package dans un environnement où les tests d'intégration pourront être lancés.

verify - Lance tous les outils de validation qui vérifient que le package satisfait les critères de qualité souhaités.

install - Installe le package dans le dépôt local, pour être utilisé localement comme dépendance par un autre projet.

deploy - Fait dans l'environnement d'intégration où chez le client, elle copie le package terminé dans un dépôt distant pour être partagé avec d'autres développeurs ou d'autres projets.

Pour lancer toutes les phases de ce cycle de vie, il suffit d'appeler la commande suivante :

```
mvn deploy
```

Comme l'exécution d'une phase lance aussi toutes les phases qui précèdent, la commande suivante permet d'exécuter le cycle de vie jusqu'à la création du fichier `jar`.

```
mvn package
```

En plus du cycle par défaut, Maven propose deux autres étapes : `site` et `clean`. Le premier est fait pour générer la documentation en ligne et le second permet de nettoyer un projet de tout ce qui a été généré par les *"builds"* précédents.

1.5 Ressources

Maven sépare le code source des ressources de l'application. Ces ressources sont par exemple les fichiers de configuration, des images ou tout autre fichier utilisé par le code de l'application. Par défaut ces fichiers doivent se situer dans le répertoire `src/main/resources`. Maven fait en sorte que ces fichiers se retrouvent au bon endroit lors de la construction de l'archive. Par exemple, pour une application utilisant JPA, c'est dans ce dossier que devra se situer le fichier `META-INF/persistence.xml`.

1.6 Filtre

Les ressources n'ont pas pour unique intérêt de séparer clairement les fichiers de configuration pour améliorer l'organisation de nos projet. Même si ce n'est pas une fonctionnalité activée par défaut, le principal atout est de pouvoir réaliser du filtrage sur ces fichiers.

Filtrer les fichiers correspond à remplacer à l'intérieur de tous les fichiers et à chaque compilation tous les `${properties}` par une valeur définie par le système ou le développeur. Ce mécanisme permet donc de paramétrer le projet grâce à des propriétés qui seront remplacées par de vrais valeurs à l'exécution. Pour activer le filtrage sur un répertoire de ressources, il faut ajouter au POM le code suivant :

```
<build>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
```

```
      <filtering>true</filtering>
    </resource>
  </resources>
</build>
```

Maven met à disposition un certain nombre de propriétés comme par exemple `${pom.name}` qui contient le nom du projet ou `${pom.version}` qui correspond à la version. Pour définir des nouvelles propriétés le développeur a deux solutions. La première, la plus simple, est d'ajouter les propriétés à la ligne de commande d'exécution de Maven. Par exemple si l'on souhaite créer une propriété `${bd.motDePasse}` contenant le mot de passe de connexion à la base de données de notre machine de développement, il suffit d'exécuter Maven de la manière suivante :

```
mvn package "-Dbd.motDePasse=monmdptoutnul"
```

La seconde solution pour définir des propriétés est de passer par un fichier dédié (dit fichier *properties*) qui sera utilisé pour le filtrage. Cette méthode est à préférer quand on souhaite définir un plus grand nombre de propriétés. Si l'on utilise un système de gestion de version, il faudra veiller à séparer les filtres communs à tous les développeurs des filtres individuels (comme le mot de passe de la BD de dev). Par convention les fichiers *properties* sont situés dans le répertoire `src/main/filters`. Pour configurer ces fichiers, il faut ajouter au POM le code suivant :

```
<build>
  <filters>
    <filter>src/main/filters/filter.properties</filter>
  </filters>
</build>
```

Le fichier `filter.properties` est un fichier "clé=valeur" classique. Pour notre exemple d'application il pourrait ressembler à cela :

```
bd.url=jdbc:oracle:thin:nedjar/nedjar@//allegro.iut.univ-aix.fr:1522/
   orcl.iut.univ-aix.fr
bd.user=onyann
bd.password=monmdptoutnul
bd.driver=com.mysql.jdbc.Driver
```

Il faut noter que si une propriété est présente dans un fichier *properties* et qu'elle est aussi définie dans la ligne de commande, alors c'est cette dernière qui sera utilisée. Ainsi même si un filtre est par défaut commun à tous il pourra être redéfini localement par un développeur pour faire des essais.

Avec nos filtres d'exemple, le fichier `src/main/resources/META-INF/persistence.xml` ressemblerait à ceci :

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/
                persistence/persistence_2_0.xsd" version="2.0"
             xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="maPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</
       provider>
    <properties>
      <property name="javax.persistence.jdbc.url" value="${bd.url}"/>
      <property name="javax.persistence.jdbc.user" value="${bd.user}"
         />
      <property name="javax.persistence.jdbc.password" value="${bd.
         password}"/>
```

```
      <property name="javax.persistence.jdbc.driver" value="${bd.
          driver}"/>
      <property name="eclipselink.ddl-generation" value="create-
          tables"/>
      <property name="javax.persistence.level" value="SEVERE"/>
    </properties>
  </persistence-unit>
</persistence>
```

2 Mise en place de l'environnement de travail

2.1 Installation

Sur les machines du département la version 2.2.1 de Maven est déjà installée, donc pour cette étape il n'y a rien à faire. Si vous utilisez votre propre machine, vous pouvez soit installer la version présente dans les dépôts de votre distribution [2] où vous pouvez aussi l'installer manuellement en suivant les instructions de la page officielle : http://maven.apache.org/download.html.

La version d'eclipse présente sur les machines du département est la dernière disponible. Pour la récupérer pour chez vous, allez sur la page http://www.eclipse.org/downloads/ puis téléchargez "Eclipse IDE for Java Developers" version linux. Cette version contient par défaut les plugin Maven, Git et Junit. Elle vous simplifiera donc bien la vie pour vos projets à réaliser en Java d'ici la fin du S4.

Pour éviter d'utiliser plusieurs versions de Maven pour le même projet, il faut aller dans le menu `Window->Preferences` puis dans la partie `Maven->Installation`. Une fois dans cette fenêtre, il faudra cliquer sur le bouton `Add` et indiquer le répertoire `/usr/share/maven2/`.

2.2 Création d'un premier projet

Avant d'utiliser Maven pour nos projets Java, nous allons voir comment créer un simple projet d'exemple (le helloworld de Maven). Pour créer ce projet, nous allons faire appel au mécanisme d'archétype. Un archétype est défini comme un modèle réutilisable. Dans Maven, un archétype est un template (une trame en français) d'un projet qui est combiné avec des informations entrées par l'utilisateur pour produire un projet Maven qui a été assemblé pour les besoins spécifiques de l'utilisateur. Nous allons voir comment le mécanisme d'archétype fonctionne, pour en savoir plus sur les archétypes, vous pouvez consulter cette page de la documentation officielle : http://maven.apache.org/guides/introduction/introduction-to-archetypes.html.

1. Comme nous voulons que notre projet Maven puisse être importé par Eclipse, nous commençons par nous déplacer dans le workspace.

   ```
   cd ~/workspace
   ```

   Pour éviter de devoir réimporter salement vos projet à chaque début de TP, il est plus qu'intéressant de mettre votre workspace sur une clef USB et de choisir au démarrage d'Eclipse ce workspace là. En plus vous pourrez ainsi travailler de la même façon que ce soit à l'IUT ou chez vous.

2. Ensuite on créé notre projet avec la ligne de commande suivante :

   ```
   mvn archetype:generate -DinteractiveMode=false \
   -DarchetypeArtifactId=maven-archetype-quickstart \
   -DgroupId=com.mycompany.app -DartifactId=my-app
   ```

2. Un développeur n'ayant pas un Unix sur sa machine ne peut pas être raisonnablement considéré comme un vrai développeur.

Les paramètres `-DgroupId` et `-DartifactId` permettent de définir convenablement votre projet (voir le paragraphe sur le POM).

3. Le projet est situé dans un dossier ayant même nom que l'`artifactId`, c'est à dire dans notre exemple "my-app". Ce dossier a la structure suivante :

```
my-app/
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- mycompany
    |               `-- app
    |                   `-- App.java
    `-- test
        `-- java
            `-- com
                `-- mycompany
                    `-- app
                        `-- AppTest.java
```

4. La compilation de ce projet se fait en exécutant les commandes suivantes[3] :

```
cd my-app
mvn package
```

5. Pour rendre le projet compatible avec Eclipse il faut exécuter la commande suivante :

```
mvn eclipse:eclipse
```

   Une fois cette commande exécutée, il suffit d'importer le projet à partir d'Eclipse pour pouvoir commencer à travailler.

Cette introduction à Maven est très succincte mais elle vous permettra de démarrer vos projets Java en utilisant pour sa construction un outil qui vous facilitera la vie. Certains IDE n'ont pas une intégration parfaite de Maven, c'est pour cela qu'il est souvent plus pratique de l'utiliser en ligne de commande.

3. Attention généralement le premier lancement de Maven est très long car il va récupérer tout ce dont il a besoin par le réseau