

ADELE/LIG

« Mon 1er Bundle Avec Maven »

Version 1.1.3

Eric SIMON



« Mon 1^{er} Bundle Avec Maven » de Eric Simon est mis à disposition selon les termes de la [licence Creative Commons Paternité 3.0 non transcrit](#).

1 But du document

Ce document a pour but d'expliquer en quelques lignes comment définir et produire un bundle OSGi [1.][2.] avec Maven[3.].

2 Qu'est-ce que Maven

Maven est un outil de gestion de projet logiciel couvrant un large spectre d'activités du cycle de vie du logiciel telles que la conception, le développement, le déploiement, la documentation et l'évaluation. Plus formellement, 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*. (Section 1.1 de [5.][6.])

Cependant, dans ce document, nous nous intéresserons avant tout à la production d'un artefact spécifique : un bundle OSGi. Pour plus d'information au sujet de Maven, il faut consulter les références [4.] [6.] [6.].

Maven permet de modéliser un projet qui permettra de produire un artefact ; celui-ci sera transféré dans un dépôt pour réutilisation ou pour installation. Pour cela, nous nous intéresserons à quatre concepts : les *archétypes*, les *pom*, les *artefacts* et les *dépôts*.

2.1 Archétypes

Maven se base sur la structure d'un projet pour automatiser un ensemble de tâches. Cependant, un projet sert à produire un type d'artefact spécifique et est donc **typé**. Chaque type de projet va avoir sa propre structure. Par exemple, la Figure 1 montre deux structures de projet : une structure pour un projet de composant iPOJO et une autre pour un projet de génération de site Web.

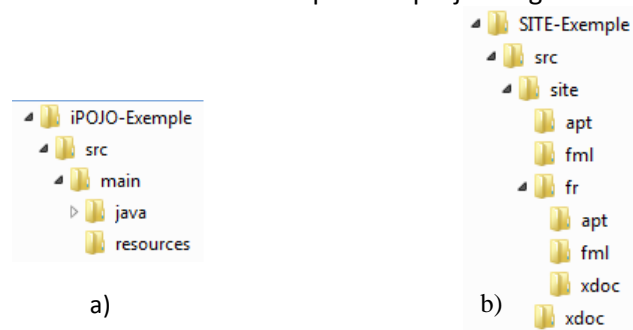


Figure 1 Structures : a) artefact iPOJO et b) artefact site

Maven a défini un type d'artefact qui permet de définir des types de projet : les *archétypes*. Les *archétypes* permettent de générer une structure pour un type de projet donné.

L'exemple de la Figure 1 a) a été obtenu à l'aide de la commande :

```
D:\Mon-1er-Bundle-Avec-Maven\exemples> mvn org.apache.maven.plugins:maven-archetype-plugin:2.1:generate -DarchetypeArtifactId=maven-ipojo-plugin -DarchetypeVersion=1.6 -DarchetypeGroupId=org.apache.felix -DgroupId=exemple.archetype -DartifactId=iPOJO-Exemple -Dversion=0.0.1-SNAPSHOT -Dpackage=exemple.archetype.ipojo
```

Cette commande utilise l'archetype pour un projet iPOJO (options : *archetypeGroupId*, *archetypeArtifactId* et *archetypeVersion*). Il génère la structure du projet `exemple.archetype:iPOJO-Exemple:0.0.1-SNAPSHOT` soit respectivement `<groupId>`:`<artifactId>`:`<version>`.

Alors que la structure du site a été obtenue à l'aide de la commande :

```
D:\Mon-1er-Bundle-Avec-Maven\exemples> mvn org.apache.maven.plugins:maven-archetype-plugin:2.1:generate -DarchetypeGroupId=org.apache.maven.archetypes -DarchetypeArtifactId=maven-archetype-site -DarchetypeVersion=1.1 -DgroupId=exemple.archetype -DartifactId=SITE-Exemple -Dversion=0.0.1-SNAPSHOT -Dpackage=exemple.archetype.site
```

Exemple Helloworld : génération de la structure du projet

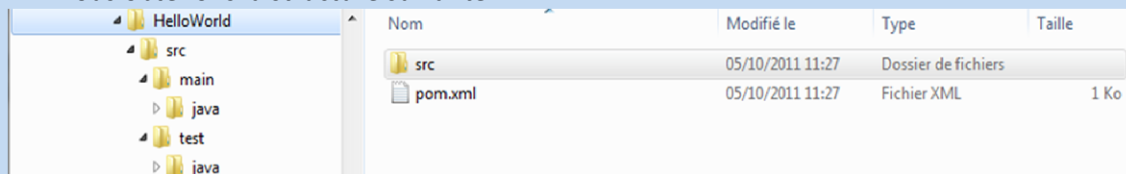
Dans notre cas, nous souhaitons produire un bundle OSGi. Cependant, il n'existe pas d'archétype spécifique, il y a donc deux solutions :

- utiliser l'archétype d'iPOJO qui est un sous-projet OSGi et enlever les propriétés propres à iPOJO ;
- utiliser l'archétype du projet générique et compléter avec les propriétés OSGi.

Nous allons opter pour la deuxième solution, soit la commande :

```
➤ mvn org.apache.maven.plugins:maven-archetype-plugin:2.1:create -DgroupId=mon.premier.bundle -DartifactId=HelloWorld -Dversion=0.0.1-SNAPSHOT -Dpackage=mon.premier.bundle
```

Nous obtenons la structure suivante :



The screenshot shows a file explorer window with the 'HelloWorld' project selected. The left pane shows the directory structure: HelloWorld > src > main > java, HelloWorld > test > java. The right pane shows a table of files:

Nom	Modifié le	Type	Taille
src	05/10/2011 11:27	Dossier de fichiers	
pom.xml	05/10/2011 11:27	Fichier XML	1 Ko

Nous allons maintenant nous intéresser sur le fichier de description de projet appelé *pom* (*Project Object Model*), situé à la racine du projet.

2.2 Le fichier pom.xml

Nous allons maintenant nous intéresser au fichier de description de projet appelé *pom.xml* (*Project Object Model*), situé à la racine du projet.

Un fichier *pom* est un fichier XML qui modélise les propriétés du projet. Il peut être décomposé en cinq grandes parties :

- les propriétés du projet : son nom, son groupe, son numéro de version, les développeurs, la license...
- le *build* : les opérations qui vont être réalisées pour l'obtention de l'artefact.
- le *report* : les opérations qui vont permettre d'établir un rapport sur la production de l'artefact : les métriques sur le code, le rapport des tests unitaires... la création du site Web du projet contenant tous les rapports !
- les *dépendances* : ce sont les artefacts Maven requis pour la production de l'artefact. C'est, par exemple, l'ensemble des bibliothèques Java nécessaires pour un jar qui se retrouveront dans le classpath.
- le *chainage* entre projets : Maven permet de définir des relations père/fils entre les projets, il permet d'une part de faire des configurations partagées entre les projets ou de produire un ensemble d'artefacts cohérents : par exemple packager trois jar qui fonctionnent ensemble.

Dans le cadre de ce document, nous nous intéresserons seulement aux propriétés, au *build* et aux dépendances nécessaires à la production du *bundle HelloWorld*. Pour les autres informations,

veuillez-vous référer au chapitre 3 de [6.] ; il est plus que recommandé de lire les bonnes pratiques : section 3.6.

2.2.1 Propriétés

Exemple Helloworld : pom.xml

Le *pom.xml* généré dans la section précédente est :

```
<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>mon.premier.bundle</groupId>
  <artifactId>HelloWorld</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>HelloWorld</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Les propriétés du projet :
groupe, nom, version...

Dépendances de l'artefact

Dans notre cas nous n'allons pas faire de test unitaire Junit, par conséquent on va retirer la dépendance sur Junit, et on va aussi supprimer le répertoire test.

Maven définit un ensemble de propriétés pour définir un projet. Il y a cinq propriétés obligatoires :

- *modelVersion* : indique la version du pom ; pour Maven 2 et 3 le format est toujours 4.0.0 ;
- *groupId* : cette propriété sert de namespace aux artefacts permettant au passage une classification des artefacts ;
- *artifactId* : cette propriété est l'identifiant de l'artefact ;
- *version* : cette propriété définit la version de l'artefact ;
- *packaging* : cette propriété définit le type de l'artefact. Dans le cas d'un *bundle* OSGi, cette propriété est : *bundle*.

Un artefact est défini de façon unique par le tuple : `<groupId>:<artifactId>:<version>`. Il existe cependant une grande quantité de propriétés pouvant être à caractère informatif ou pour la configuration de *plugin* ; mais dans tous les cas, les cinq propriétés précédentes sont toujours présentes... ou déduites pour le *groupId* et la *version*. En effet, si un parent est déclaré et que la *version* et/ou le *groupId* du projet ne sont pas définis alors les valeurs de ces propriétés sont ceux du parent.

2.2.2 Dépendances

Un des atouts majeurs de Maven est la gestion des dépendances. En effet, dans les outils comme ant ou Makefile, les dépendances (par exemple, bibliothèques .jar) sont placées dans un répertoire

(en général *lib/*) et sont ajoutées au classpath pour la compilation. Le transfert d'un projet nécessite alors de joindre l'ensemble des dépendances qui peuvent être volumineuse. Et, la mise à jour des dépendances devient une tâche fastidieuse. Par opposition, Maven permet de déclarer les dépendances du projet qui sont résolues grâce aux dépôts distribués (cf. section 2.4).

Exemple Helloworld : implémentation et dépendance

Nous allons implémenter la classe `mon.premier.bundle.Activator` (Helloworld). Nous n'utiliserons pas la classe `App` générée par maven : supprimez-la. Puis implémentez la classe `Activator` de la façon suivante :

```
package mon.premier.bundle;

import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

/**
 * Hello world!
 */
public class Activator implements BundleActivator {

    public void start(BundleContext arg0) throws Exception {
        System.out.println("Hello");
    }

    public void stop(BundleContext arg0) throws Exception {
        System.out.println("Bye");
    }

}
```

L'implémentation Helloworld est basée sur la classe d'activation définie par OSGi : *BundleActivator*. Nous avons besoin du package `org.osgi.framework`. Celui-ci est fourni par le bundle `org.apache.felix:org.osgi.core`. Nous ajoutons donc la dépendance au fichier `pom.xml` :

```
<dependency>
  <groupId>org.apache.felix</groupId>
  <artifactId>org.osgi.core</artifactId>
  <version>1.4.0</version>
</dependency>
```

2.2.3 Build

Il nous reste finalement à effectuer les opérations pour construire le bundle : le *build*. Dans Maven, les opérations de *build* sont définies sous la forme de *plugins* qui s'exécutent à des phases données du cycle de vie (Chapitre 4 [6.]). Dans le cas d'un bundle, il faut utiliser un *plugin* de compilation et un autre *plugin* qui va, d'une part, générer le fichier de description (manifest) et, d'autre part, emballer le code et le fichier de description dans un *bundle*.

Exemple Helloworld configuration du build

Nous obtenons la configuration d *build* 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">
  <modelVersion>4.0.0</modelVersion>
  <groupId>mon.premier.bundle</groupId>
  <artifactId>HelloWorld</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>bundle</packaging>
  <name>HelloWorld</name>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.3.2</version>
        <configuration>
          <source>1.6</source>
          <target>1.6</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <version>2.3.5</version>
        <extensions>true</extensions>
        <configuration>
          <instructions>
            <Bundle-Name>${pom.artifactId}.</Bundle-Name>
            <Bundle-Description>Description de l'artefact</Bundle-Description>
            <Bundle-Activator>mon.premier.bundle.Activator</Bundle-Activator>
            <Bundle-SymbolicName>${pom.artifactId}</Bundle-SymbolicName>
            <Bundle-Vendor>Vous</Bundle-Vendor>
            <Export-Package></Export-Package>
            <Private-Package>mon.premier.bundle</Private-Package>
            <Import-Package>org.osgi.framework</Import-Package>
          </instructions>
        </configuration>
      </plugin>
    </plugins>
  </build>

  <dependencies>
    <dependency>
      <groupId>org.apache.felix</groupId>
      <artifactId>org.osgi.core</artifactId>
      <version>1.4.0</version>
    </dependency>
  </dependencies>
</project>
```

Plugin de compilation Java
en 1.6. (cf. [7.]

Plugin pour le bundle. Il
fournit les informations à
mettre dans le manifest du
jar. (cf. [8.]

De plus, si vous utilisez iPOJO dans votre *bundle*, il faut ajouter son *plugin* (cf. [9.]) :

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-ipojo-plugin</artifactId>
  <version>1.6.0</version>
  <executions>
    <execution>
      <goals>
        <goal>ipojo-bundle</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

2.3 Artefacts

Comme nous l'avons dit précédemment, tout projet produit un artefact. Les artefacts peuvent être des *bundles*, des jar, des war, des zip, des nœuds du chaînage structurant les projets... Ces artefacts sont le résultat du *build*. Le *build* suit un cycle de vie précis et l'état de l'artefact dépend de la phase du cycle où l'on s'arrête.

Dans cette section, nous allons voir cinq commandes correspondant à cinq phases du cycle de vie de Maven :

- Clean : cette commande efface du répertoire du projet tout ce qui a été produit par un *build* ;
- Compile : cette commande compile le code source du projet ;
- Package : cette commande empaquette le code compilé dans le format indiqué : dans notre cas, un *bundle* (jar) ;
- Install : cette commande déploie dans le dépôt local le code empaqueté : dans notre cas, le *maven bundle plugin* génère le fichier `repository.xml` utilisable par l'OBR d'OSGi ;
- Deploy : cette commande déploie dans le dépôt distant le code empaqueté : dans notre cas, le *maven bundle plugin* génère le fichier `repository.xml` utilisable par l'OBR d'OSGi.

Il n'est pas nécessaire de faire chaque phase manuellement. Par exemple, si vous avez déjà compilé et que vous faites un « *mvn deploy* » ; maven fera automatiquement toutes les phases entre ces deux phases ; soit : *package* et *install*.

2.4 Dépôts

Comme nous venons de le voir précédemment, Maven utilise des dépôts (cf. : *install* et *deploy*). Maven base tout son raisonnement sur les artefacts dans les différents dépôts. Les dépôts sont hiérarchisés :

- à la base, votre dépôt local ;
- au sommet, un ensemble de dépôts distants.

Par défaut, il n'y a qu'un seul dépôt distant qui est <http://repo2.maven.org/maven2/> (navigation désactivé et accessible via <http://search.maven.org/>). Cependant, il est possible de définir des dépôts intermédiaires. Par exemple, une entreprise va définir un dépôt intermédiaire qui restreint l'accès aux dépôts externes (pour des raisons de licence ou pour ajouter ses propres artefacts pour les redistribuer dans ses différents départements) et où chaque département de cette entreprise va aussi avoir un dépôt intermédiaire pour configurer leurs projets.

Il est important de comprendre que lorsque Maven résout une dépendance ou recherche un parent, **il se base sur les dépôts et non sur la structure de votre projet.**

3 Références

- [1.] E. Simon. « OSGi En Bref ». 2011 - <http://membres-liglab.imag.fr/simon/doc/OSGi-en-bref-v1.0.2.pdf>
- [2.] OSGi Alliance - <http://www.osgi.org>
- [3.] Apache. *Apache Maven Project* - <http://maven.apache.org/>
- [4.] Sonatype. *Maven: The Complete Reference* - <http://www.sonatype.com/books/mvnref-book/reference/>
- [5.] *Maven Guide Fr* - <http://maven-guide-fr.erwan-alliaume.com/maven-guide-fr/maven-guide-fr.pdf>

-
- [6.] *Maven Reference Fr* - <http://maven-guide-fr.erwan-alliaume.com/maven-reference-fr/maven-reference-fr.pdf>
 - [7.] Apache. *Maven Compiler plugin* - <http://maven.apache.org/plugins/maven-compiler-plugin/>
 - [8.] Apache Felix. *Maven Bundle plugin* - <http://felix.apache.org/site/apache-felix-maven-bundle-plugin-bnd.html>
 - [9.] Apache Felix. *Maven iPOJO plugin* - <http://felix.apache.org/site/ipojo-maven-plug-in.html>