Introduction à Maven

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-31

Table des matières

1 Objectifs d'apprentissage 1
2 Qu'est-ce que Maven ? 1
3 Création d'un projet 2
4 Utilisation avancée 6
4.1 Livraison en Fatjar 6
4.2 Livraison modulaire 6
4.3 Dépendances et versions 7
4.4 POM Parent 7

1 Objectifs d'apprentissage

À l'issue de cette leçon, vous serez capable de :

- **OOC-MAVEN-01** — Localiser dans un POM les éléments structurants (coordonnées, dépendances, plugins) qui pilotent le cycle de vie d'un projet Java.
- **OOC-MAVEN-02** — Appliquer les commandes Maven pour créer un projet à partir d'un archetype et en assurer la compilation.
- **OOC-MAVEN-03** — Analyser la déclaration des dépendances et la gestion des versions dans un POM.
- **OOC-MAVEN-04** — Concevoir une organisation multi-modules articulée autour d'un POM parent porteur de la configuration commune.
- **OOC-MAVEN-05** — Évaluer les stratégies de livraison, fatjar ou modulaire, à partir des contraintes d'exécution d'un projet.

2 Qu'est-ce que Maven ?

Maven est un puissant outil de gestion de projet pour Java. Il simplifie diverses tâches essentielles telles que la compilation, la génération de documentation, la gestion des dépendances, l'intégration avec les systèmes de gestion de version du code source, la gestion des releases et la distribution des projets Java.

Les objets gérés par Maven, comme les bibliothèques Java, les modèles de projets, et les plugins Maven, sont appelés des **artefacts**. Chaque artefact est identifié par des **coordonnées** spécifiques, constituées de trois éléments principaux :

- `groupId` : Identifie fonctionnellement un groupe d'artefacts.
- `artefactId` : Identifie un artefact spécifique au sein du groupe.
- `version` : Spécifie la version de l'artefact.

Pour les versions, il est recommandé de suivre une approche « sémantique » conformément aux principes du versionnage sémantique (semver).

- Outil de gestion de projet pour Java
- Simplifie : compilation, documentation, dépendances, versions, releases, distribution
- **Artefacts Maven**:
  - Objets gérés (bibliothèques, modèles de projets, plugins)
  - **Coordonnées** : `groupId` (groupe), `artefactId` (identifiant), `version` (version)
- **Versionnage Sémantique**
  - pour la version suivre l'approche semver

3 Création d'un projet

Un projet Maven peut être créé à partir d'un **archetype** Maven, qui est un modèle de projet préconfiguré facilitant la création de projets conformes à certaines structures et pratiques standardisées.

- **Choix de l'Archetype**
  - `archetypeGroupId` : Identifie le groupe auquel appartient l'archetype.
  - `archetypeArtifactId` : Identifie l'archetype spécifique à utiliser.
  - `archetypeVersion` : Spécifie la version de l'archetype.
- **Options du Projet**
  - `groupId` : Identifie le groupe auquel le projet à créer appartient.
  - `artifactId` : Identifie le nouveau projet.
  - `version` : Spécifie la version initiale du projet.

Dans l'exemple ci-dessous l'archetype utilisé est un projet Java de base. Les options `--no-transfer-progress`, `--batch-mode` et `--color always` ne sont utiles que pour la génération non interactive.

```
%%shell
cd /home/jovyan/work/src/github/ebpro/notebook-java-maven/

mvn archetype:generate --no-transfer-progress --batch-mode --color always \
    -DarchetypeGroupId=org.apache.maven.archetypes \
```

```
    -DarchetypeArtifactId=maven-archetype-quickstart \
    -DarchetypeVersion=1.5 \
    -DgroupId=fr.univtln.bruno.samples \
    -DartifactId=MyMinimalApp \
    -Dversion=0.1.0-SNAPSHOT
```

L'arborescence est celle par défaut de Maven :

```
%%shell
tree -C /home/jovyan/work/src/github/ebpro/notebook-java-maven/MyMinimalApp
```

On peut ensuite compiler le projet depuis l'intérieur du projet (le dossier contenant le fichier `pom.xml`). Les classes sont produites dans le sous-répertoire `target/classes`. Le programme peut être exécuté avec la commande `java` en ajoutant ce répertoire au classpath si aucune dépendance externe n'est nécessaire.

En pratique, il est conseillé de ne pas utiliser maven directement mais via un script standard: Le Maven Wrapper. C'est un outil essentiel pour les développeurs Java souhaitant garantir la cohérence des versions de Maven utilisées dans leurs projets. En ajoutant simplement un script (mvnw ou mvnw.cmd) à votre projet, le Maven Wrapper permet d'exécuter des commandes Maven avec une version spécifique, sans nécessiter une installation préalable de Maven sur chaque machine de développement. La création de ce script se fait via la commande `mvn wrapper:wrapper`.

```
%%shell
cd /home/jovyan/work/src/github/ebpro/notebook-java-maven/MyMinimalApp
mvn -ntp --color always wrapper:wrapper
```

Il est aussi possible d'en faire une archive java (fichier .jar) dans le sous-répertoire `target` avec le cible maven `package`.

```
%%shell
cd /home/jovyan/work/src/github/ebpro/notebook-java-maven/MyMinimalApp
./mvnw -ntp --color always package
```

La plugin exec permet d'exécuter le projet courant en fixant aut`exec:java` d'exécuter une classe Java en prenant en charge automatiquement le classpath y compris les dépendances indiquée dans `pom.xml`.

```
%%shell
cd /home/jovyan/work/src/github/ebpro/notebook-java-maven/MyMinimalApp
```

```
./mvnw -ntp --color always exec:java -Dexec.mainClass="fr.univtln.bruno.samples.App"
```

Le projet peut être paramétré en modifiant le fichier `pom.xml`. Il est possible de modifier le nom du projet, son URL et d'ajouter des métadonnées.

La construction du projet est contrôlée par des plugins paramétrés dans la section `build/plugins`.

Par exemple, il est possible de changer la version du code source acceptée pour ce projet et celle du bytecode java produit. Les évolution du langage et la machine virtuelle correspondent à des versions précises.

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.release>17</maven.compiler.release>
</properties>
```

La commande `javap` permet d'explorer le byecode et d'afficher la version. D'après wikipedia les versions majeures sont :

```
Java SE 22 = 66 (0x42 hex)
Java SE 21 = 65 (0x41 hex)
Java SE 20 = 64 (0x40 hex)
Java SE 19 = 63 (0x3F hex)
Java SE 18 = 62 (0x3E hex)
Java SE 17 = 61 (0x3D hex)
Java SE 16 = 60 (0x3C hex)
Java SE 15 = 59 (0x3B hex)
Java SE 14 = 58 (0x3A hex)
Java SE 13 = 57 (0x39 hex)
Java SE 12 = 56 (0x38 hex)
Java SE 11 = 55 (0x37 hex)
Java SE 10 = 54 (0x36 hex)
Java SE 9 = 53 (0x35 hex)
Java SE 8 = 52 (0x34 hex)
Java SE 7 = 51 (0x33 hex)
Java SE 6.0 = 50 (0x32 hex)
Java SE 5.0 = 49 (0x31 hex)
JDK 1.4 = 48 (0x30 hex)
JDK 1.3 = 47 (0x2F hex)
JDK 1.2 = 46 (0x2E hex)
JDK 1.1 = 45 (0x2D hex)
```

Le modèle de projet utilisé fixe la version de Java à 17 (même si le compilateur est d'une version supérieur).

```
%%shell
cd /home/jovyan/work/src/github/ebpro/notebook-java-maven/MyMinimalApp
echo "Version de JDK installée: $(java --version|head -n 1)"
echo "Version cible indiquée dans le pom.xml: $(grep '<maven.compiler.release>'
pom.xml)"
echo "Version du bytecode de la classe principale compilée :"
javap -v -c target/classes/fr/univtln/bruno/samples/App.class | grep "major
version"
```

Si l'on augmente la version du JDK cela ne change pas la version de bytecode produit.

```
%%shell
cd /home/jovyan/work/src/github/ebpro/notebook-java-maven/MyMinimalApp

sdk use java $(sdk list java|grep installed|cut -d '|' -f 6|tr -d ' '|grep
"^23.*-tem")
./mvnw --quiet package

echo "Version de JDK installée: $(java --version|head -n 1)"
echo "Version cible indiquée dans le pom.xml: $(grep '<maven.compiler.release>'
pom.xml)"
echo "Version du bytecode de la classe principale compilée :"
javap -v -c target/classes/fr/univtln/bruno/samples/App.class | grep "major
version"
```

Il faut modifier le `pom.xml` pour passer de la version 17 à la version 22 de Java (). Vérifier que celle du bytecode augmente.

```
%%shell
cd /home/jovyan/work/src/github/ebpro/notebook-java-maven/MyMinimalApp
sed -i 's/>17</>22</g' pom.xml

sdk use java $(sdk list java|grep installed|cut -d '|' -f 6|tr -d ' '|grep
"^23.*-tem")
./mvnw --quiet package

echo "Version de JDK installée: $(java --version|head -n 1)"
echo "Version cible indiquée dans le pom.xml: $(grep '<maven.compiler.release>'
pom.xml)"
echo "Version du bytecode de la classe principale compilée :"
```

```
javap -v -c target/classes/fr/univtln/bruno/samples/App.class | grep "major
version"
```

4 Utilisation avancée

Un projet java réel est souvent plus complexe, l'archetype non officiel suivant intégre les points les plus importants.

```
%%shell
cd /home/jovyan/work/src/github/ebpro/notebook-java-maven/
mvn archetype:generate \
   --color always --quiet --no-transfer-progress --batch-mode \
   -DarchetypeGroupId=fr.ebruno.maven.archetypes \
   -DarchetypeArtifactId=maven-archetype-simple \
   -DarchetypeVersion=0.1.9 \
   -DgroupId=fr.univtln.bruno.samples \
   -DartifactId=MyAppHelloWorld \
   -Dversion=0.1.0-SNAPSHOT

tree -C MyAppHelloWorld
```

4.1 Livraison en Fatjar

En utilisant le plugin `shade`, il est possible de produire des fichiers `.jar` qui incluent toutes les dépendances nécessaires. Ce type de fichier est appelé un **fatjar** ou **uberjar**. Il permet également de définir une classe principale par défaut pour rendre le jar « exécutable ». C'est l'approche traditionnelle pour livrer un exécutable Java.

Ce modèle inclut également des tests unitaires et des tests d'intégration dans la phase `verify`.

```
%%shell
cd /home/jovyan/work/src/github/ebpro/notebook-java-maven/MyAppHelloWorld
./mvnw --quiet --ntp -P shadedjar verify
ls -lh target/*jar
java -jar target/MyAppHelloWorld-0.1.0-SNAPSHOT-withdependencies.jar
```

4.2 Livraison modulaire

L'approche moderne, bien que plus complexe, pour livrer un programme Java consiste à utiliser les modules introduits dans Java 9 avec l'outil `jlink`. `jlink` génère une version allégée du JRE (Java Runtime Environment) qui inclut seulement les composants requis par votre application. Le profil `jlink` intégre cela, ainsi, il n'est plus nécessaire d'installer de JRE sur la machine du client.

```
%%shell
cd /home/jovyan/work/src/github/ebpro/notebook-java-maven/MyAppHelloWorld
./mvnw --quiet --ntp -P jlink clean verify
du -hs target/image

./target/image/bin/myapp
```

4.3 Dépendances et versions

Les dépendances et plugins sont gérés automatiquement par Maven à partir du `pom.xml`. Ils sont téléchargés à partir d'entrepôt (par défaut `maven central` qui est public). Le cache local se trouve dans `$HOME/.m2/repository`.

Maven dispose de plugin pour gérer efficacement les versions des dependences et plugins: https://www.mojohaus.org/versions/versions-maven-plugin/index.html

Il est possible d'afficher les mises à jour disponibles des dépendances ou des propriétés (ici les transitives sont ignorées) voire de les mettres à jour:

```
%%shell
cd /home/jovyan/work/src/github/ebpro/notebook-java-maven/MyAppHelloWorld
# FIX : le chemin vers rules.xml dans le pom.xml est incorrect
sed -i 's/<rulesUri>file:\/\/\/\/home\/runner\/_work\/maven-archetypes\/maven-
archetypes\/maven-archetype-simple\/rules.xml<\/rulesUri>/<rulesUri>file:.\/
rules.xml<\/rulesUri>/' pom.xml

./mvnw --color always -ntp versions:display-dependency-updates \
    -D processDependencyManagementTransitive=false

./mvnw --color always -ntp versions:display-property-updates \
    -D processDependencyManagementTransitive=false
```

4.4 POM Parent

Avec Maven, un fichier POM peut rapidement devenir volumineux. De plus, une base commune doit souvent être partagée entre plusieurs projets et/ou plusieurs membres d'une équipe. Comme il s'agit d'un modèle orienté objet, il est possible (et fortement conseillé) de factoriser les paramétrages communs dans un ou plusieurs POM parent. Par exemple, chaque POM d'un projet peut hériter de celui d'une équipe, qui hérite lui-même de celui de l'entreprise.

Pour créer un POM parent, il suffit de créer un projet contenant un fichier `pom.xml` dont le packaging est défini à `pom`, puis de le déposer dans un repository commun (par exemple, en utilisant `mvn install` sur chaque machine ou, de façon plus réaliste, en utilisant un entrepôt Maven).

```
<parent>
  <groupId>...</groupId>
  <artifactId>...</artifactId>
  <version>...</version>
</parent>
```

La commande mvn help:effective-pom permet de visualiser le POM après intégration des ancêtres (dont celui par défaut).

Un POM parent est proposé ici https://github.com/ebpro/maven-parentpom.

Un archetype plus complet utilisant le pom parent précédent est proposé ici https://github.com/ebpro/maven-archetypes.

```
%%shell
cd /home/jovyan/work/src/github/ebpro/notebook-java-maven/

mvn --quiet --color always -U -ntp archetype:generate -B \
 -DarchetypeGroupId=fr.ebruno.maven.archetypes \
 -DarchetypeArtifactId=maven-archetype-withparent \
 -DarchetypeVersion=0.1.9 \
 -DgroupId=fr.univtln.bruno.samples \
 -DartifactId=MyAppHelloWorldWithParent \
 -Dversion=0.1.0-SNAPSHOT \
 -DgithubAccount=ebpro
```

```
%%shell
cd                              /home/jovyan/work/src/github/ebpro/notebook-java-
maven/MyAppHelloWorldWithParent
mvn -ntp -quiet -P shadedjar verify
mvn -ntp -quiet exec:java
```

Il est possible d'afficher les mises à jour disponibles des plugins :

```
%%shell
cd                              /home/jovyan/work/src/github/ebpro/notebook-java-
maven/MyAppHelloWorldWithParent

mvn -ntp versions:display-plugin-updates
```

Il est aussi possible de mettre à jour automatiquement :

```
mvn versions:update-parent
mvn versions:use-latest-releases
mvn versions:update-properties
```