Intro à Maven

Rémi Forax

Outils de build en Java

Il existe plusieurs outils qui permettent de gérer la fabrication de librairies/applications

Les 3 principaux sont

- Ant (correspondant à Make en C)
  - Plus vraiment utilisé
- Maven (spécification déclarative)
  - Utilisé pour les projets simples ou que l’on veut simple
- Gradle (spécification en code)
  - Utilisé pour les projets plus complexes (ex Android)

Maven

De façon simplifier

- Gère et va chercher les dépendances versionnées
- Compile le code source puis les tests
- Exécute les tests (unitaires et intégrations)
- Génère un jar (librarie ou exécutable)

Deux modes

- un seul “module” (1 seul jar)
- Multi-modules (1 projet parent, 1 sous-module par sous-répertoire)

POM - Project Object Model

Le fichier pom.xml décrit un projet sous forme d’objet eux même décrit en XML

Anatomie

- Information sur le module
  <groupId>, <artifactId>, <version>
- Le type de pom (simple module ou multi-module)
  <packaging> (jar ou pom)
- Metadata
  <name>, <description>, <url>, <licenses>, <organization>, <developers>, <contributors>
- Propriété
  - <properties>
- Dépendences
  <dependencies>
- Build (configuration des plugins)
  <build>

Coordonées Maven

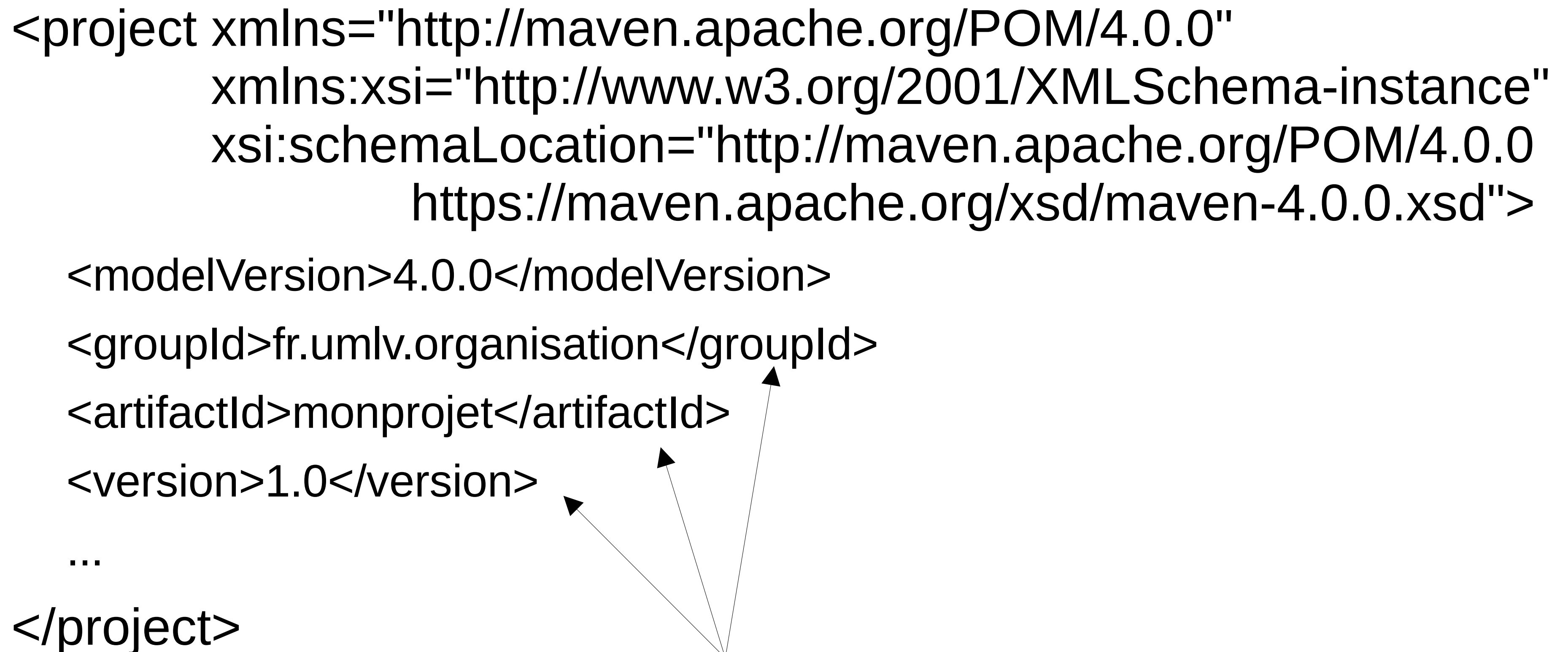
Maven utilise un système de 3 valeurs pour définir un module de façon universelle

<groupId>:<artifactId>:<version>

- <groupId>
  - Nom de l'organisation souvent com.google.bar (reverse DNS)
- <artifactId>
  - Nom du projet souvent un nom simple (pas de '.')
- <version>
  - Valeur de la version souvent x.y.z (semver.org)
    - x version majeur qui casse la compatibilité
    - y version mineur qui ajoute des APIs
    - z version patch qui corrige les bugs sans changer l'API

POM: Coordonée du module

La version du POM est la version 4.0



```
<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
                  https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.umlv.organisation</groupId>
  <artifactId>monprojet</artifactId>
  <version>1.0</version>
  ...
</project>
```

Repository / Maven Central

Un repository est un endroit contenant les modules accessibles à partir de leur coordonée

- Maven Central (ou Central)

  https://repo1.maven.org/

  par exemple, le module org.junit.jupiter:junit-jupiter:5.8.1

  - https://repo1.maven.org/maven2/org/junit/jupiter/junit-jupiter/5.8.1/
- Un repository Maven dans une entreprise
- Un répertoire Maven local (pour une équipe de devs)

Maven garde aussi une copie en local dans $HOME/.m2

$HOME/.m2/org/junit/jupiter/junit-jupiter/5.8.1/

Projet simple module

Par défault, le but d’un build Maven est de produire des jars

- Un jar qui contient les .class (librarie ou application)
- Un jar qui contient le code source (pour le debug)
- Un jar qui contient la javadoc (pour la documentation)

On indique le packaging (jar par défaut)

<packaging>jar</packaging>

Projet multi-modules

Dans le pom.xml, on indique pom comme packaging et on liste les modules

```
<packaging>pom</packaging>

<modules>
  <module>my-module</module>
</modules>
```

Sur le disque, un sous-répertoire par module avec le nom du module (ici, my-module)

Chaque module a un pom.xml qui indique les coordonées du pom parent

```
<parent>
  <artifactId>foo</artifactId>
  <groupId>fr.umlv.group</groupId>
  <version>1.0</version>
</parent>
```

Dépendences

L’ensembles des modules qui sont nécessaire pour le module à la compilation et à l’exécution

```
<dependencies>
  <dependency>
    <groupId>org.ow2.asm</groupId>
    <artifactId>asm</artifactId>
    <version>9.2</version>
    <scope>test</scope>
    <optional>true</optional>
  </dependency>
  …
</dependencies>
```

Scopes
- **compile** (défaut)
  - compilation et exécution
- **test**
  - compilation/executions des tests
- **runtime**
  - uniquement à l’exécution
- **provided**
  - uniquement pour les plugins

Optionel à l’exécution

Le build

Pour builder le projet, on utilise des plugins, les plugins sont aussi des modules Maven

- Par exemple, le plugin maven-compiler-plugin

```
<build>
 <plugins>
  <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-compiler-plugin</artifactId>
   <version>3.8.1</version>
  </plugin>
 </plugins>
</build>
```

Chaque plugin s’enregistre sur sa phase automatiquement

Le cycle de vie

Conventions

Un projet Maven sur disque à toujours la même structure

foo (répertoire du projet)

- pom.xml (fichier de configuration de Maven)
- src/main/java
  - Fichiers sources
- src/main/resources
  - Fichiers textes, images, associés
- src/test/java
  - Fichier des tests JUnit ou TestNG
- src/test/resources
  - Fichiers textes, etc pour les tests

Le cycle de vie (simplifié)

- generate-sources, generate resources
  - Generation par plugin: ANTLR ou processeur d'annotations (cf cours reflection
- **compile**
  - Compilation du code principale (src/main/java + fichiers générés)
- generate test-sources, generate test-resources
- **test-compile**
  - Compilation du code des tests (src/test/java + fichiers générés) avec le code principale (main) en dépendance avec le plugin `compiler`
- **test**
  - Execution des tests Unitaires Junit avec le plugin `surefire`
- **package**
  - On crée le jar (ou le war, pour ancien JavaEE) + les sources et la javadoc
- integration-test
  - Test d'intégration avec le plugin `failsafe`
- Install
  - Copies les jars dans $HOME/.m2
- deploy
  - Upload sur Maven Central

Le cycle de vie complet:
https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

Ligne de commande

- mvn permet d’exécuter Maven
  - `mvn --version`
- mvn *goal*
  - Execute toutes les phases jusqu’au *goal* indiqué
  - `mvn test` (compile et test)
  - `mvn package` (compile, test et package)

  - si on veut sauter les tests
    - `mvn -Dmaven.test.skip` (compile pas les tests)
    - `mvn -DskipTests` (n’exécute pas les tests)

Exécuter avec un JDK particulier

La commande mvn utilise

- La variable d’environement JAVA_HOME

  ```
  export JAVA_HOME=/path/to/my/jdk
  ```

- Le fichier $HOME/.mvn/jvm.config

  Par exemple,

  ```
  --enable-preview
  ```

  ajoute les options à chaque commande qui exécute java

Les plugins usuels de Maven

maven-compiler-plugin

Configuration du compilateur

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.10.1</version>
  <configuration>
    <release>19</release>
    <compilerArgs>
      <compilerArg>--enable-preview</compilerArg>

      <compilerArg>--add-modules</compilerArg>
      <compilerArg>jdk.incubator.vector</compilerArg>
    </compilerArgs>
  </configuration>
</plugin>
```

Version de Java

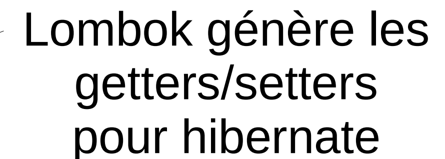
Active les preview features

Ajoute un module en incubation
(module avec API pas stable de Java)

Avec un processeur d’annotations

Un processeur d’annotations est un plugin du compilateur Java qui parcours les annotations dans le code Java et génère du code Java à partir des valeurs de annotations

```xml
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.10.1</version>
    <configuration>
    <release>19</release>
    <annotationProcessorPaths>
      <path>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.16.12</version>
      </path>
    </annotationProcessorPaths>
  </configuration>
</plugin>
```



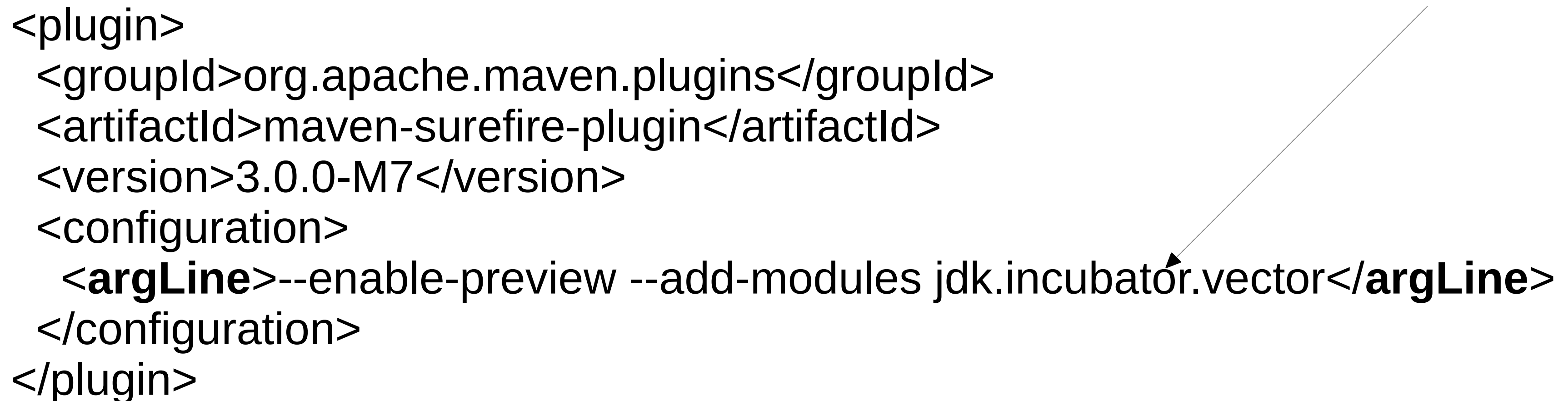
Il faut aussi déclarer les coordonnées du processeur d’annotation dans les dépendences avec le scope provided.

maven-surefire-plugin

Exécute les tests unitaires (avant de créer le jar)

Le nom des fichiers de test doit finir par Test

Chaque plugin a sa propre façon de se configurer, :(



```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>3.0.0-M7</version>
  <configuration>
    <argLine>--enable-preview --add-modules jdk.incubator.vector</argLine>
  </configuration>
</plugin>
```

Il faut aussi ajouter l’API JUnit (junit-jupiter.api) dans les dépendences avec le scope test.

maven-failsafe-plugin

Exécute les tests d’intégrations

Le nom des fichiers de test doit finir par IT

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>3.0.0-M7</version>
  <configuration>
    <argLine>--enable-preview</argLine>
  </configuration>
</plugin>
```

maven-shade-plugin

Créé un seul jar (uber-jar) avec le code et toutes les dépendances sous forme d’un jar exécutable

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <version>3.4.0</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>shade</goal>
      </goals>
      <configuration>
        <finalName>my-application</finalName>
        <transformers>
          <transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
            <mainClass>org.openjdk.jmh.Main</mainClass>
          </transformer>
        </transformers>
        <filters>
          <filter>
            <artifact>*:*</artifact>
            <excludes>
              <exclude>**/module-info.class</exclude>
              <exclude>META-INF/MANIFEST.MF</exclude>
            </excludes>
          </filter>
        </filters>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Pour chaque plugin, on peut indiquer sa phase d’exécution

Le goal pour la ligne de commande

Le nom du uber-jar

Le nom de la classe qui contient le main()

Les trucs que l’on doit exclure car il y en a dans chaque bibliothèque