Intro à Maven

C'est quoi ?

Ca sert à plusieurs choses :

- Il y a plusieurs projets qui doivent être compilés dans un certain ordre. Disons Projet3, puis Projet2 et enfin Projet1. Si on doit faire ça à la main, c'est rapidement pénible et on a toutes les chances de se tromper. Maven permet de faire ça automatiquement.
- Un projet peut avoir besoin d'un jar utilitaire. Disons jar1. Lequel a besoin de jar2, lequel a besoin de jar3., etc …. Ici aussi, ça devient vite incompréhensible. Maven permet de spécifier la dépendance de premier niveau (jar1) et automatiquement il ira chercher les dépendances de jar1 (jar2, jar3, etc …)
- Maven permet aussi de lancer tous les tests unitaires en une seule passe.

Maven permet de faire plein d'autre choses, mais on laisse tomber pour le moment ☺

Comment ça marche ?

Maven utilise un fichier XML qui décrit les opérations. Il s'appelle pom.xml.
Voyons, un exemple simple.

```
<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>

Ici, on décrit le projet en cours. On verra plus tard
  <groupId>afpa.formation.fr</groupId>
  <artifactId>TestMaven</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>TestMaven</name>

Ici on écrit le type du projet. Ici c'est un jar. Si c'est du web, ce sera un war.
  <packaging>jar</packaging>

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

  Ici diverses propriétés dont la version du compilateur
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>

  Les dépendances (on a JUnit et 2 autres jars)
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
```

```
      <scope>test</scope>
    </dependency>
<!-- https://mvnrepository.com/artifact/log4j/log4j -->
      <dependency>
          <groupId>log4j</groupId>
          <artifactId>log4j</artifactId>
          <version>1.2.9</version>
      </dependency>
      <!-- https://mvnrepository.com/artifact/commons-logging/commons-logging -->
      <dependency>
          <groupId>commons-logging</groupId>
          <artifactId>commons-logging</artifactId>
          <version>1.2</version>
      </dependency>
  </dependencies>
</project>
```

La question qui se pose est : où sont stockés ces fameux jars en dépendance ? En pratique, ils sont téléchargés du réseau la première fois et sont stockés dans un repository local (ne pas confondre avec le repository GIT). En général, ce repository est dans c:\users\nom_user\.m2.

Un jar Maven est défini par trois choses :

- Son groupId qui est en gros identifie l'éditeur. Il est sous la forme d'un package java (xxx.yyy.zzz).
- Son artifactId qui est en gros son nom.
- Sa version

On voit que votre propre projet est de la même forme. Donc quand vous buildez votre projet via Maven, votre projet est copié dans le repository sous la forme d'un jar. Ce qui signifie que votre projet pourra être utilisé par d'autres projets Maven.

Comme Maven fait ou peut faire plein de choses, il faut lui préciser un *goal* qui indique quelle est l'opération à faire.

Pour débuter, on va dire qu'il y a 4 goals principaux :

- clean : efface les classes déjà crées pour partir d'une base saine. A appeler avant les 3 goals suivants.
- build : créer le jar dans votre projet (compiler toutes les classes et construire le jar). Assez peu utilisé (voir install).
- test : lance les tests unitaires. Pas très utilisé (voir install)
- install : en pratique enchaine les 2 goals précédents et copie dans le repository. C'est LE goal le plus utilisé.

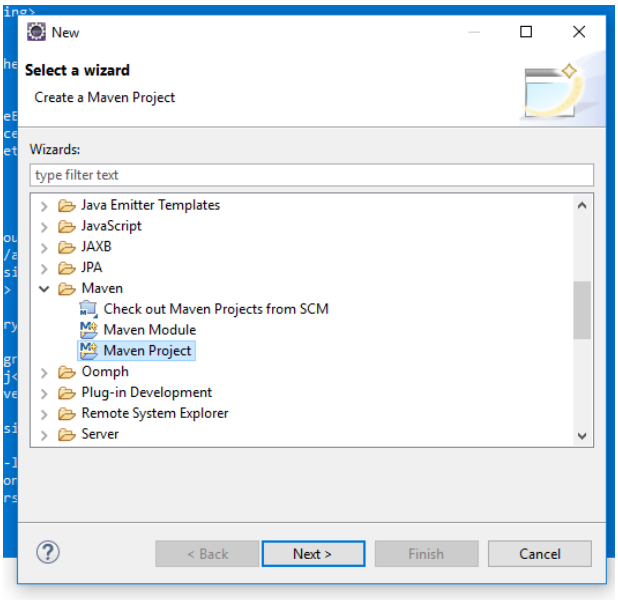
Maven dans Eclipse

Normalement Maven est un exécutable ligne de commande. Mais comme d'habitude, il est intégré dans Eclipse (via M2Eclipse) et c'est celui-ci qu'on utilisera par la suite.

Comment on fait ?

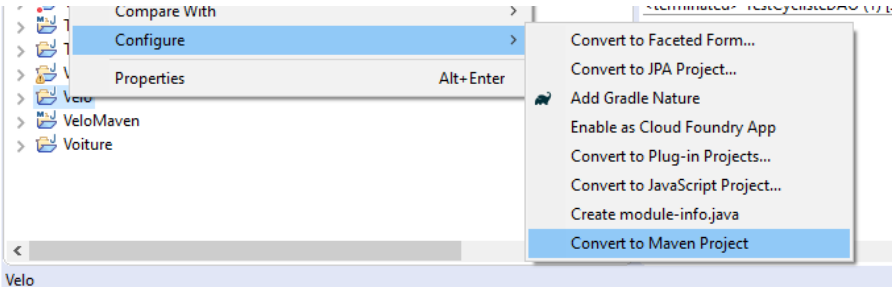
Il y a différents moyens.

On peut faire *New/Other* puis *Maven Project*. Mais il se passe parfois des choses étranges



D'un autre côté, il se passe toujours des trucs un peu étranges avec M2Eclipse.

On peut aussi créer le squelette de votre projet comme d'habitude. Puis on indique que le projet est de nature Maven.

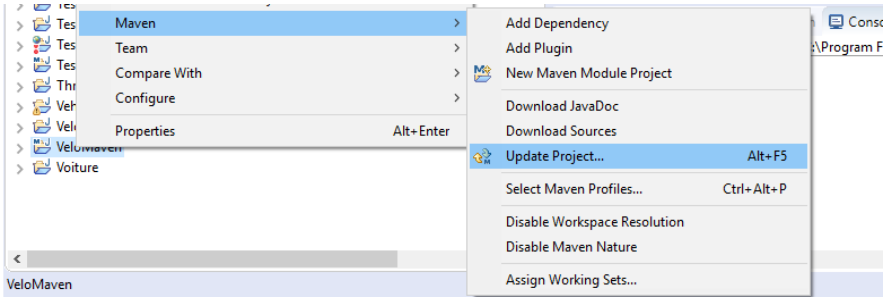


Remarquons qu'un project Maven est signalé par un petit 'M'

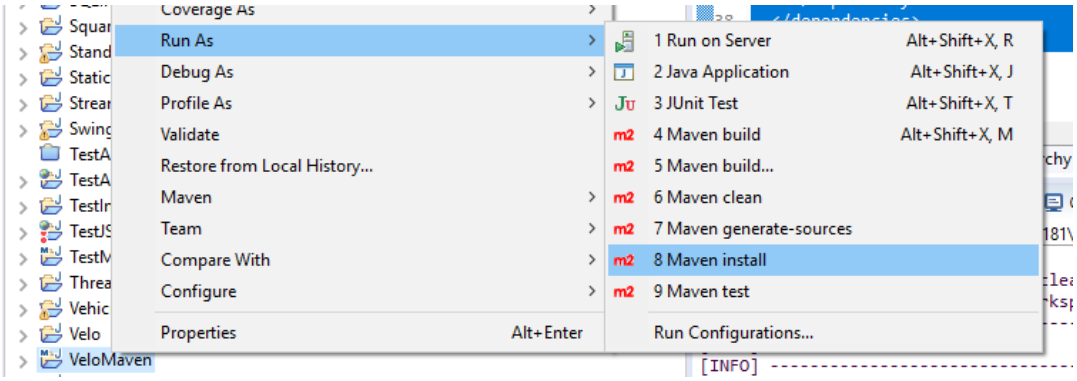
Dans les deux cas, le pom.xml est créé, mais dans le premier cas, on fait intervenir la notion d'archétype et c'est bien assez compliqué comme ça (les archétypes sont en fait des types de squelettes de projets). Donc je préconise plutôt la seconde méthode pour commencer.

Après il suffit d'ajouter les dépendances dont a besoin le projet dans le pom.xml et normalement tout est prêt à fonctionner.

Comme M2Eclipse est un peu capricieux, pour être sûr que le projet est vraiment prêt, il est préférable de faire *Maven/Update Projet*

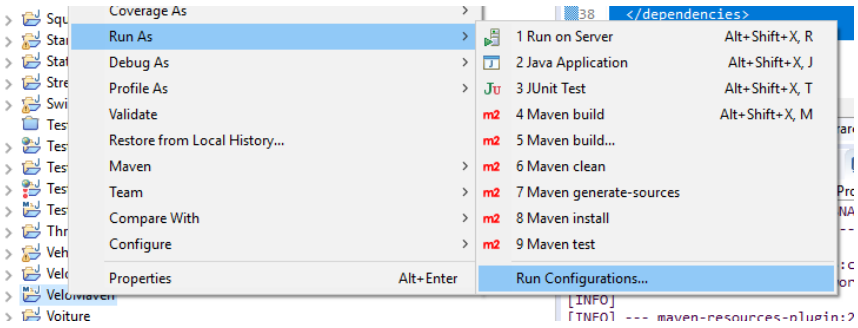


Pour lancer le goal *install* (et les tests unitaires par la même occasion), on fait *Run as/Maven install*.

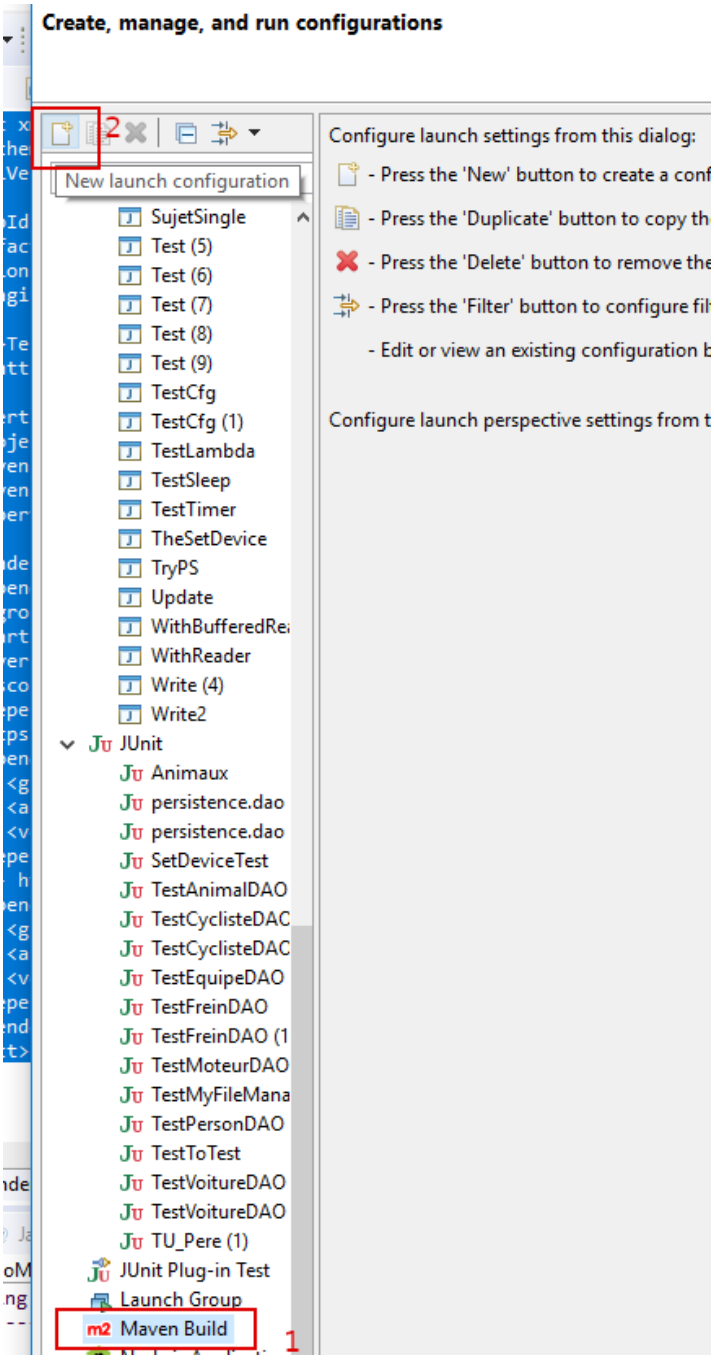


Il est vivement conseillé de faire un clean avant. On peut aussi éviter les TU en cochant « Skip test ».

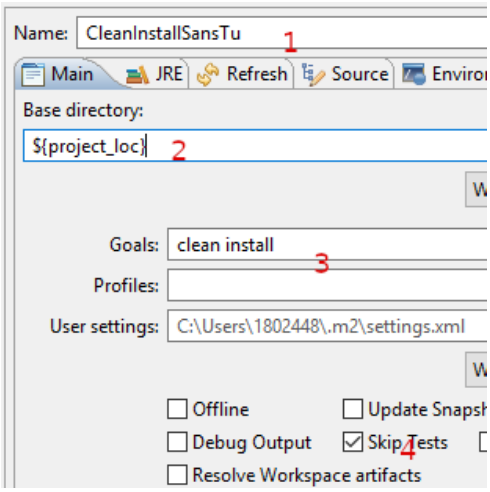
Voyons comment créer un goal personnalisé :



On crée un builder Maven



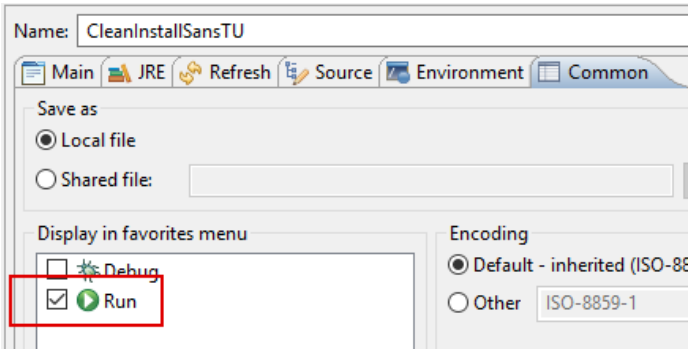
On sélectionne *Maven Build* puis on appuie sur *New*.



Dans l'ordre, on rentre

- Le nom du builder
- L'emplacement où faire le build ({project_loc} indique le projet courant)
- Le ou les goals (ici clean install)
- On sélectionne Skip tests

Enfin, on va dans l'onglet *Common*, et on sélectionne là où doit apparaître le builder (ici dans *Run*)



Puis on appuie sur *Apply*

Le builder apparait sous l'icône *Run*