

MAVEN & INTÉGRATION CONTINUE

MORISSEAU Mickael

MAVEN

Qu'est ce que Maven ?

Maven

Prérequis à télécharger

Java (minimum JDK8)

Eclipse

BDD (MySQL ou H2 ou PostgreSQL)

Qu'est ce que Maven

Maven est un outil de construction de projets (build) open source développé par la fondation Apache (initialement pour les besoins du projet Jakarta Turbine). Il permet de faciliter et d'automatiser certaines tâches de la gestion d'un projet Java.

Le site web officiel est <http://maven.apache.org>

maven

Mise en place de Maven

Mise en place de Maven

Maven est un outil écrit en **Java** et faisant de la compilation **Java**.

Il faut donc avoir un JDK Java.

Téléchargement et installation

Téléchargez Apache Maven depuis le site officiel :

<https://maven.apache.org/download.cgi>

Télécharger la version en fonction du système d'exploitation :

Linux => Binary tar.gz

Windows => Binary zip

Décompressez l'archive dans votre répertoire, exemple :

D:\Programs\apache-maven-3.6.3

Le répertoire de *Maven* contient plusieurs sous-répertoires :

bin => des scripts dont la commande mvn

conf => contient la configuration par défaut dans le fichier [settings.xml](#)

lib => les bibliothèques contenant le noyau de *Maven*

Mise en place de Maven

Définir les variables d'environnement

Définir le chemin vers le JDK grâce à la variable d'environnement JAVA_HOME et ajouter les binaires du JDK et de Maven au PATH

Sous Windows :

Variable	Valeur
JAVA_HOME	C:\Program Files (x86)\Java\jdk1.6.0_29
Path	C:\Program Files\Intel\WiFi\bin;C:\Pro...
...	..

JAVA_HOME => C:\Program Files\Java\jdk1.8.0_281

PATH => C:\Program Files (x86)\Java\jdk1.8.0_281\bin;D:\Programs\apache-maven-3.6.3\bin

Sous Linux dans :

/etc/profile

```
user@COMPUTER ~
$ export JAVA_HOME="/chemin/vers/repertoire/java/jdk1.8.0_281"
user@COMPUTER ~
$ export MAVEN_HOME="/chemin/vers/repertoire/maven/apache-maven-3.6.3"
user@COMPUTER ~
$ PATH="$PATH:$JAVA_HOME/bin:$MAVEN_HOME/bin"
user@COMPUTER ~
$ source ~/.profile
```

Mise en place de Maven

Tester l'installation

Dans l'invite de commande, taper « mvn -version » :

```
C:\Users\mmorisseau>mvn -v
Apache Maven 3.6.3 (cecedd343002696d0abb50b32b541b8a6ba2883f)
Maven home: C:\Programmation\Programs\apache-maven\apache-maven-3.6.3\bin\...
Java version: 1.8.0_281, vendor: Oracle Corporation, runtime: C:\Program File...
Default locale: fr_FR, platform encoding: Cp1252
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"
```

Attention : si vous avez ce message :

```
C:\Users\mmorisseau>mvn -v
Exception in thread "main" java.lang.UnsupportedClassVersionError: org/apache/maven/cli/MavenCli : Unsupported major.minor version 51.0
        at java.lang.ClassLoader.defineClass1(Native Method)
        at java.lang.ClassLoader.defineClassCond(ClassLoader.java:631)
```

Cela veut dire que vous n'avez pas la version minimum de JDK Java

Configuration Maven

Paramètre

La configuration de *Maven* se fait dans le fichier « `settings.xml` ».

Sous Windows : `.m2/settings.xml` (dans le répertoire utilisateur)

Sous Linux : `~/.m2/settings.xml` (dans le répertoire home)

Astuce : copier/coller le fichier `settings` présent dans le dossier `conf` de `apache-maven-x-x-x`

Ouvrir le fichier `setting.xml` :

Editer le XML pour affecter la ligne `<localRepository>` afin de renseigner le répertoire où seront téléchargé les dépendances sur votre disque.

Exemple :

```
<settings xmlns="http://maven.apache.org/settings/1.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/settings-1.0.0.xsd">
    ...
    <!-- Chemin par défaut du repository -->
    <localRepository>D:\Users\mmorisse\.m2\repository</localRepository>
```

Attention : le répertoire prendra rapidement beaucoup d'espace disque.

Créer un projet Maven

Créer un projet Maven

Maven utilise une approche dite « convention over configuration » (comprendre : utilisation de conventions par défaut pour standardiser les projets).

Cela signifie que *Maven* a établi un certain nombre de conventions, en les respectant beaucoup de choses seront automatiques.

L'avantage de *Maven* est qu'il y a très peu de configuration à faire.

Une des premières conventions concerne l'arborescence d'un projet Maven, elle fixée (voir site *Maven*).

Maven vous permet de générer un squelette de votre projet. C'est ce que je vais vous montrer tout de suite.

Créer un projet Maven

Maven permet de générer un squelette de projet.

Afin de générer le squelette d'un projet, *Maven* s'appuie sur des archétypes (ce sont des sortes de modèles).

Exercice : demander à *Maven* de générer un squelette à partir de l'archétype « quickstart ».

1) En ligne de commande :

```
C:\Users\mmorisse>d:  
D:\>cd Projet_ESEO/Maven
```

2) Lancer une génération à partir de l'archétype, via la ligne de commande suivante :

```
> mvn archetype:generate -DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.1
```

```
D:\Projet_ESEO\Maven>mvn archetype:generate -DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.1
```

Créer un projet Maven

3) Personnalisation de la génération du projet :

groupId : com
artifactId : TP_ESEO_Maven
version (1.0-SNAPSHOT) : laissez vide
package (org.exemple.demo) : laissez vide

4) Pour finir répondre « Y », exemple :

```
Define value for property 'groupId': com
Define value for property 'artifactId': ESEO_TP_Maven
[INFO] Using property: version = 1.0-SNAPSHOT
Define value for property 'package' com: :
Confirm properties configuration:
groupId: com
artifactId: ESEO_TP_Maven
version: 1.0-SNAPSHOT
package: com
Y: :
```

5) Fin avec le résultat suivant : « BUILD SUCCES »

```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 3:50.961s
[INFO] Finished at: Tue Dec 05 21:17:28 CET 2017
[INFO] Final Memory: 8M/28M
[INFO] -----
```

Créer un projet Maven

Autre possibilité de *Maven* pour permettre de générer un squelette de projet.

1) En ligne de commande :

```
C:\Users\mmorisse>d:  
D:\>cd Projet_ESEO/Maven
```

2) Lancer une génération à partir de l'archétype, via la ligne de commande suivante :

```
> mvn archetype:generate \  
  -DarchetypeGroupId=org.apache.maven.archetypes \  
  -DarchetypeArtifactId=maven-archetype-quickstart \  
  -DarchetypeVersion=1.1 \  
  -DgroupId=com \  
  -DartifactId=ESEO_TP_Maven \  
  -Dversion=1.0-SNAPSHOT
```

Attention : sur internet il y a des exemples avec mvn archetype:create, cette méthode est déprécié.

Créer un projet Maven

3) Arborescence générée :

```
D:\Projet_ESEO\Maven>tree /F
Structure du dossier pour le volume Data
Le numéro de série du volume est 0AA1-D3EF
D:.
└── ESEO_TP_Maven
    └── pom.xml

    └── src
        ├── main
        │   └── java
        │       └── com (Package définie)
        │           └── App.java
        └── test
            └── java
                └── com (Package définie)
                    └── AppTest.java
```

Explication :

- un fichier POM ([pom.xml](#)) à la racine du projet.

C'est le fichier de description/configuration du projet Maven.

- un répertoire [src/main/java](#).

C'est le répertoire contenant les sources de du projet, composées seulement pour le moment, d'un fichier [App.java](#) dans le package [com](#).

- un répertoire [src/test/java](#).

C'est le répertoire contenant les sources des tests JUnit du projet, composées seulement, pour l'instant, d'un fichier [AppTest.java](#) dans le package [com](#).

Créer un projet Maven

3) Arborescence générée :

Répertoire	Contenu
/src	les sources du projet (répertoire qui doit être ajouté dans le gestionnaire de sources)
/src/main	les fichiers sources principaux
/src/main/java	le code source (sera compilé dans /target/classes)
/src/main/resources	les fichiers de ressources (fichiers de configuration, images, ...). Le contenu de ce répertoire est copié dans target/classes pour être inclus dans l'artefact généré
/src/test	les fichiers pour les tests
/src/test/java	le code source des tests (sera compilé dans /target/test-classes)
/src/test/resources	les fichiers de ressources pour les tests
/target	les fichiers générés pour les artefacts et les tests (ce répertoire ne doit pas être inclus dans le gestionnaire de sources)
/target/classes	les classes compilées
/target/test-classes	les classes compilées des tests unitaires
/pom.xml	le fichier POM de description du projet

Créer un projet Maven

Contenu du fichier « pom.xml »:

```
<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</groupId>
    <artifactId>ESEO_TP_Maven</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>ESEO_TP_Maven</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>
```

Créer un projet Maven

Contenu du fichier « pom.xml »:

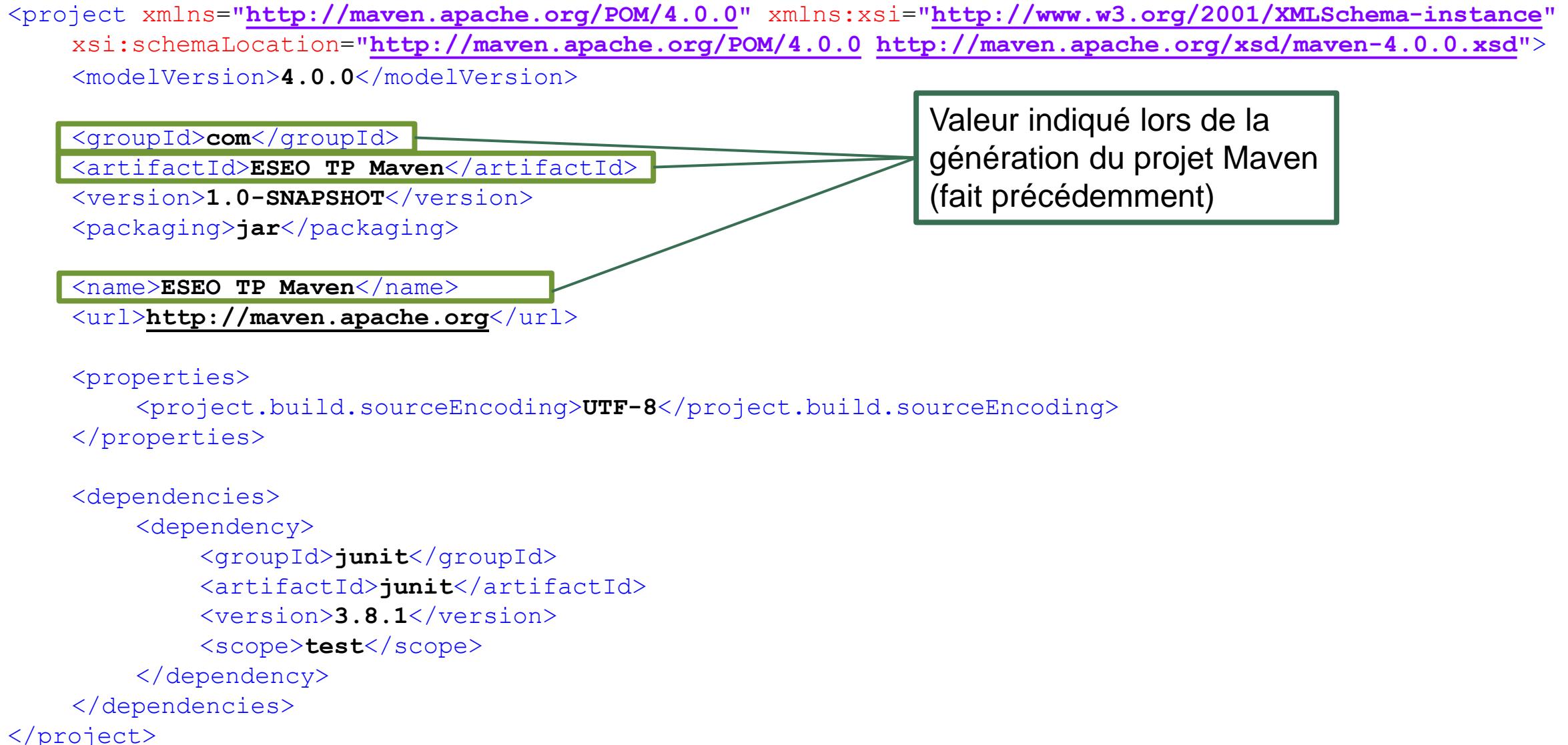
```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com</groupId>
    <artifactId>ESEO TP Maven</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>ESEO TP Maven</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>
```



Valeur indiqué lors de la génération du projet Maven (fait précédemment)

Créer un projet Maven

Contenu du fichier « App.java »:

```
package com;

/**
 * Hello world!
 *
 */

public class App {
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
    }
}
```

Créer un projet Maven

Compilation du projet avec Maven :

En ligne de commande, se placer au niveau du fichier « [pom.xml](#) » :

```
D:\Projet_ESEO\Maven>cd ESEO_TP_Maven  
D:\Projet_ESEO\Maven\ESEO_TP_Maven>mvn package
```

Les lignes suivantes s'affichent dans la console :

```
D:\Projet_ESEO\Maven\ESEO_TP_Maven>mvn package  
[INFO] Scanning for projects...  
[INFO]  
[INFO] ---  
[INFO] Building ESEO_TP_Maven 1.0-SNAPSHOT  
[INFO] ---  
[INFO]  
[INFO] ...  
[INFO]  
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ ESEO_TP_Maven ---  
[INFO] ---  
[INFO] BUILD SUCCESS  
[INFO] ---  
[INFO] Total time: 7.697 s  
[INFO] Finished at: 2017-12-05T23:13:36+01:00  
[INFO] Final Memory: 10M/150M  
[INFO] ---
```

Créer un projet Maven

Plusieurs choses ont été réalisées par Maven :

- copie des ressources de l'application
- compilation des sources de l'application
- compilation des sources des tests
- exécution des tests sur l'application (1 test exécuté, aucune erreur)

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

- génération du JAR de l'application

Dans le répertoire « Target » :

```
D:\Projet_ESEO\Maven\ESEO_TP_Maven>tree /f
D:.

    pom.xml

    src
        main
            java
                com
                    App.java

        test
            java
                com
                    AppTest.java

    target
        ESEO_TP_Maven-1.0-SNAPSHOT.jar
```

Créer un projet Maven

Exécution du jar, avec la commande suivante :

```
> java -cp target/ESEO_TP_Maven-1.0-SNAPSHOT.jar com.App
```

Résultat :

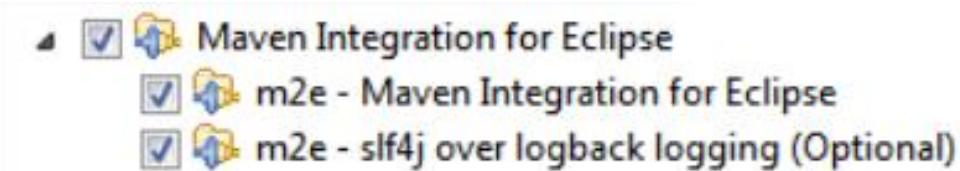
```
D:\Projet_ESEO\Maven\ESEO_TP_Maven>java -cp target/ESEO_TP_Maven-1.0-SNAPSHOT.jar com.App
Hello World!
```

Maven avec Eclipse

Installation, deux solutions :

- 1- Téléchargé Eclipse IDE for Java EE Developers ou Eclipse IDE for Java Developers, le plugin de support de *Maven* est déjà embarqué
- 2- Installer le plugin M2Eclipse

URL : <http://download.eclipse.org/technology/m2e/releases>

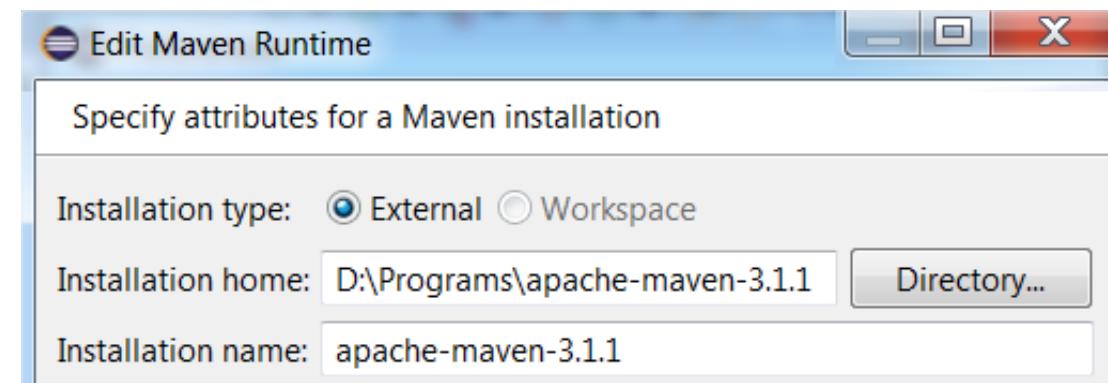


Note : sur les dernières versions d'Eclipse Maven est déjà disponible.

Ouvrir Eclipse

- Aller dans Windows > Preferences
- Aller dans Maven > Installations et Add ...
- Configurer comme l'exemple ==>

Cochez la nouvelle installation que vous venez de créer pour qu'elle devienne celle par défaut.



Maven avec Eclipse

Ouvrir Eclipse

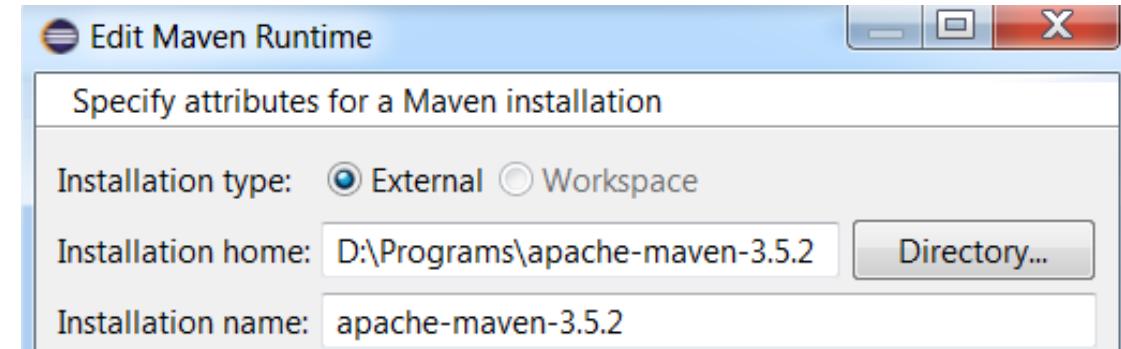
- Aller dans Windows > Preferences
- Aller dans Maven > Installations et Add ...
- Configurer comme l'exemple ==>

Cochez la nouvelle installation que vous venez de créer pour qu'elle devienne celle par défaut.

Installations

Select the installation used to launch Maven:

Name	Details
EMBEDDED	3.2.1/1.5.1.20150109-1819
WORKSPACE	⚠ NOT AVAILABLE [3.0.]
apache-maven-3.5.2	D:\Programs\apache-maven-3.5.2 3.5.2

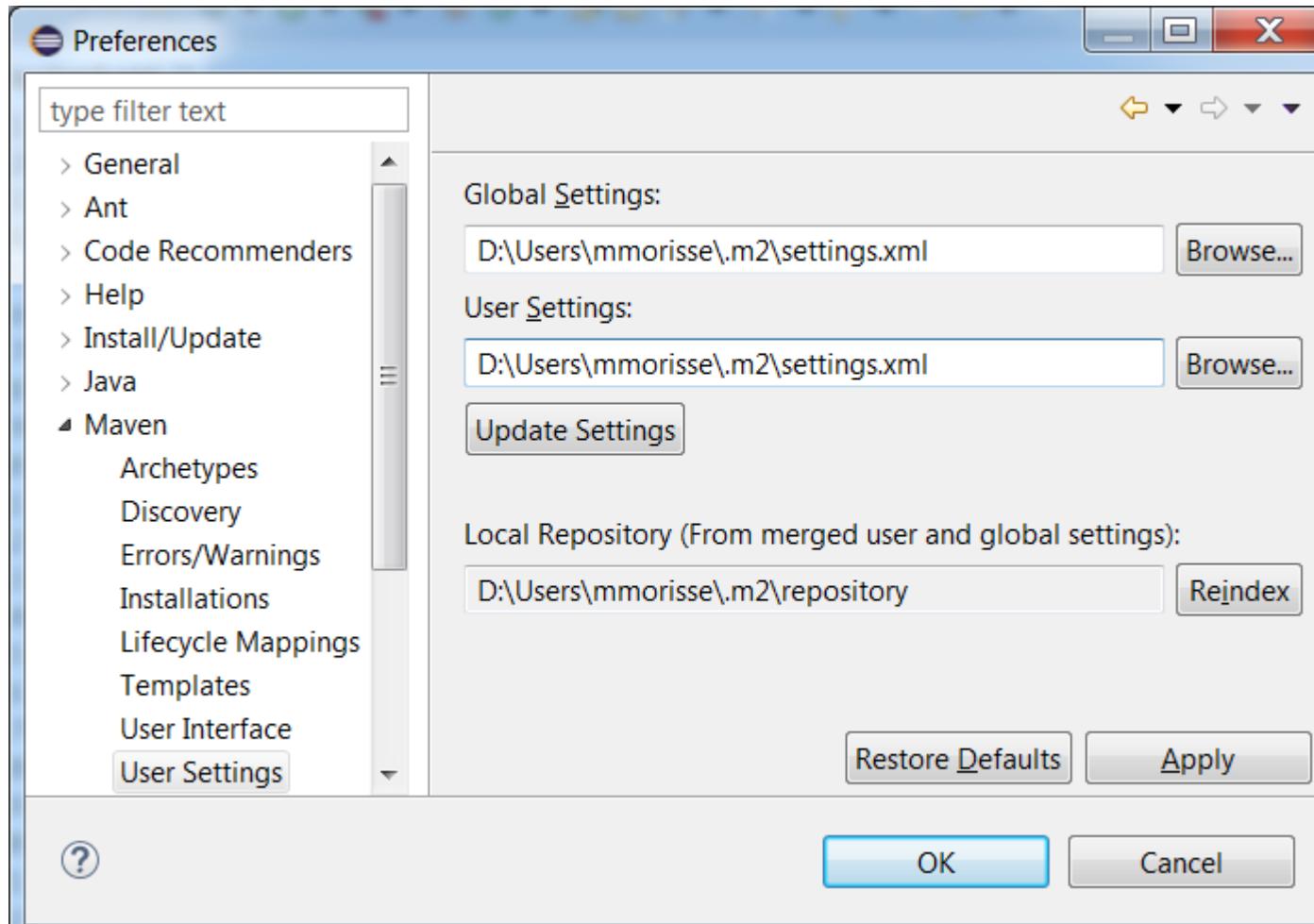


Eclipse embarque une version « EMBEDDED » de base, ce qui explique que l'on peut faire du *Maven* sans avoir obligatoirement à ajouter une installation.

Maven avec Eclipse

Réglage :

Aller dans « User Setting » (le plus important) :



Créer un projet Maven – En résumé

- Maven utilise une approche dite « convention over configuration » (Convention plutôt que configuration).
- L'arborescence d'un projet Maven est fixée par convention, soit :
 - + un fichier `pom.xml` à la racine qui sert à décrire et configurer le projet Maven ;
 - + un répertoire `src/main/java` contenant les sources de l'application ;
 - + un répertoire `src/test/java` contenant les sources des tests JUnit de l'application.
- Pour créer un nouveau projet Maven, ce dernier permet de générer l'arborescence en utilisant un archétype grâce à la commande :
`> mvn archetype:generate`
- Compilez le projet Maven, exéutez les tests et construisez un JAR via la commande :
`> mvn package`
Si la compilation et tous les tests passent sans erreur, un fichier est généré :
`target/<artifactId>-<version>.<packaging>`
dans l'exemple, il s'agit de :
`target/mon-appli-1.0-SNAPSHOT.jar`

Customisation du pom.xml

Customisation du pom.xml

Le fichier pom.xml est le fichier de configuration de la partie *Maven* dans un projet Java.
POM pour **Project Object Model**

La structure et le format du fichier est de type XML

Le lien vers la documentation officielle, et en particulier sur le contenu du fichier pom.xml :
<https://maven.apache.org/pom.html>

Sur le slide suivant, nous allons voir en détail le contenu du fichier.

Customisation du pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
  <!-- ===== -->
  <!-- Informations du projet -->
  <!-- ===== -->
  <!-- ===== Informations Maven ===== -->
  <groupId>com.exemple.demo</groupId>
  <artifactId>mon-appli</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <!-- ===== Informations générales ===== -->
  <name>Mon Application</name>
  <description>L'application sert à faire ...</description>
  <url>http://www.exemple.com/mon-appli</url>

  <!-- ===== Organisation ===== -->
  <organization>
    <name>Mon Entreprise</name>
    <url>http://www.exemple.com</url>
  </organization>

  <!-- ===== Licences ===== -->
  <licenses>
    <license>
      <name>Apache License, Version 2.0</name>
      <url>https://www.apache.org/licenses/LICENSE-2.0.txt</url>
    </license>
  </licenses>

  ...
</project>
```

Customisation du pom.xml – en détail

Définition des informations du projet Maven

```
<project>
  ...
  <!-- ===== Informations Maven ===== -->
  <groupId>com.exemple.demo</groupId>
  <artifactId>mon-appli</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  ...
</project>
```

En détail :

- **<groupId>** : identifiant de l'organisation gérant le projet.
Cet identifiant reprend la notation des packages Java. En général, celui-ci correspond au package de base de l'application, mais ce n'est pas obligatoire.
- **<artifactId>** : identifiant du projet
- **<version>** : version du projet.
- **<packaging>** : type de packaging devant être généré par *Maven* (jar, war, ear...).

Un projet *Maven* est identifié par ses coordonnées : groupId:artifactId:version

Customisation du pom.xml – en détail

Définition des informations du projet Maven

```
<project>
  ...
  <!-- ===== Informations Maven ===== -->
  <groupId>com.exemple.demo</groupId>
  <artifactId>mon-appli</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  ...
</project>
```

Plus d'information sur `<version>` :

Par convention, *Maven* donne un statut particulier aux versions ayant pour suffixe `-SNAPSHOT`. Cela veut dire que le code de cette version de l'application n'est pas figé. C'est une version en cours de développement.

Par opposition, si la version ne se termine pas par le suffixe `-SNAPSHOT`, cela signifie qu'il s'agit d'une release de l'application et que son code est figé.

Customisation du pom.xml – en détail

Informations générales du projet

```
<project>
  ...
  <!-- ===== Informations générales ===== -->
  <name>Mon Application</name>
  <description>L'application sert à faire ...</description>
  <url>http://www.exemple.org/mon-appli</url>
  ...
</project>
```

En détail :

- **<name>** : le nom du projet
- **<description>** : la description du projet
- **<url>** : URL du projet ou de l'application en production.

Customisation du pom.xml – en détail

Informations sur l'organisation du projet

```
<project>
  ...
  <!-- ===== Organisation ===== -->
  <organization>
    <name>Mon Entreprise</name>
    <url>http://www.example.org</url>
  </organization>
  ...
</project>
```

En détail :

- <name> : le nom de l'organisation
- <url> : URL de l'organisation

Note : Ces informations sont optionnelles.

Customisation du pom.xml – en détail

Informations sur l'organisation du projet

```
<project>
  ...
  <!-- ===== Licences ===== -->
  <licenses>
    <license>
      <name>Apache License, Version 2.0</name>
      <url>https://www.apache.org/licenses/LICENSE-2.0.txt</url>
    </license>
  </licenses>
  ...
</project>
```

En détail :

Maven permet d'indiquer la ou les licences du projet grâce à la balise `<licenses>` et une sous-balise `<license>` par licence.

Elément important si le projet est disponible publiquement ou s'il est sous licence libre.

Le pom est accessible dans l'archive du jar.

Name	Size	Packed Size	Modified	Created
pom.properties	113	109	2017-12-05 23:13	
pom.xml	776	344	2017-12-05 21:17	

Customisation du pom.xml – les propriétés

Un autre élément important disponible dans le pom.xml de *Maven* sont les propriétés.

Il s'agit de sorte de constantes. Elles sont remplacées par leur valeur lors de l'exécution de *Maven* en utilisant la notation `${maPropriete}` (qui sera remplacée par la valeur de la propriété `maPropriete`).

Vous pouvez définir les propriétés directement dans le fichier `pom.xml`, grâce à la balise `<properties>`, exemple :

```
<project>
  ...
  <!-- ===== -->
  <!-- Properties -->
  <!-- ===== -->
  <properties>
    <maPropriete1>valeur1</maPropriete1>
    <maPropriete2>valeur2</maPropriete2>
  </properties>
  ...
</project>
```

Customisation du pom.xml – les propriétés

Exemple concret :

```
<project>
    ...
    <properties>
        <slf4j.version>1.5.6</slf4j.version>
        <commonsLogging.version>1.1.1</commonsLogging.version>
    </properties>
    ...
    <dependencies>
        <!-- Dépendances common logging. -->
        <dependency>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
            <version>${commonsLogging.version}</version>
        </dependency>
        <!-- Dépendance slf4j. -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>${slf4j.version}</version>
        </dependency>
    </dependencies>
    ...
</project>
```

Exercice JAVA

Exercice :

Faire un projet Java classique (sans Maven) qui va récupérer le contenu du table d'une BDD de type MySQL ou H2 (ou PostreSQL).

Affichage dans la console

Exercice JAVA et Maven

Exercice :

Faire un projet Java avec Maven qui va récupérer le contenu du table d'une BDD de type MySQL ou H2 (ou PostreSQL).

Affichage dans la console

Pour aller plus loin :

Affichage dans la console depuis le jar généré.

Dépendances Maven

Suite à l'exercice, *Maven* à simplifier la récupération de la librairie, qui dans notre cas fait le lien entre le code Java et la base de donnée MySQL.

Maven utilise des repositories. Un repository est un endroit où sont stockées des dépendances sous un format imposé par *Maven*.

Quand on ajoute une dépendance à un projet, *Maven* la télécharge depuis un repository et la stocke dans votre repository local. Pour rappel, c'est le chemin qui a été configuré dans le fichier [settings.xml](#).

Par défaut, *Maven* utilise le repository central. Celui-ci se trouve à l'adresse :

<https://repo.maven.apache.org/maven2/>

Note : Pour rechercher parmi les dépendances disponibles dans le repository central de *Maven*, vous pouvez utiliser ce site :

<http://search.maven.org>

Customisation du pom.xml – les propriétés

Les propriétés pré-définies :

En plus des propriétés que l'on peut définir grâce à la balise `<properties>`, il existe également des « propriétés pré-définies » :

- `project.basedir` : donne le chemin vers le répertoire de base du projet, c'est-à-dire la racine de votre projet où se trouve le fichier `pom.xml`.
- `project.baseUri` : donne le chemin vers le répertoire de base du projet, mais sous forme d'URI.
- `maven.build.timestamp` : donne l'horodatage du lancement du build Maven.

Complément :

Le properties ci-dessous permet de personnaliser le format d'horodatage (de timestamp) :

```
<properties>
    <maven.build.timestamp.format>yyyy-MM-dd'T'HH:mm:ss'Z'</maven.build.timestamp.format>
</properties>
```

Customisation du pom.xml – les propriétés

Les propriétés particulières :

Vous pouvez aussi accéder à des propriétés particulières grâce aux préfixes suivants :

- env. : permet de renvoyer la valeur d'une variable d'environnement.

Par exemple, \${env.PATH} renvoie la valeur de la variable d'environnement PATH.

- project. : renvoie la valeur d'une balise dans le fichier pom.xml du projet, en utilisant le point (.) comme séparateur de chemin pour les sous-balises.

Exemple :

```
<project>
    <organization>
        <name>1.0</name>
    </organization>
</project>
```

La valeur sera accessible via \${project.organization.name}.

- settings. : renvoie la valeur d'une balise dans le(s) fichier(s) settings.xml utilisé(s) par Maven. Utilise la notation pointée comme pour les propriétés de project.

- java. : renvoie la valeur d'une propriété système de Java. Ces propriétés sont les mêmes que celles accessibles via java.lang.System.getProperties().

Par exemple, \${java.version} renvoie la version de Java.

Customisation du pom.xml – le build

En plus des informations de base et des propriétés dans le pom.xml, *Maven* offre la possibilité d'interagir avec le processus de construction/génération via les sous-balises de la balise principale `<build>`.

Exemple :

Dans le premier exemple de compilation, *Maven* a généré un « jar » dans le répertoire « target » soit :

`${project.basedir}/target`

Avec les lignes ci-dessous, on peut changer l'emplacement par défaut :

```
<project>
  ...
  <!-- ===== -->
  <!-- Build -->
  <!-- ===== -->
  <build>
    <directory>${project.basedir}/output</directory>
  </build>
  ...
</project>
```

Customisation du pom.xml – le build

```
<project>
  ...
  <!-- ===== -->
  <!-- Build -->
  <!-- ===== -->
  <build>
    <!-- Gestion des plugins (version) -->
    <pluginManagement>
      <plugins>
        <plugin> <!-- Plugin responsable de la génération du fichier JAR -->
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-jar-plugin</artifactId>
          <version>3.0.2</version>
        </plugin>
      </plugins>
    </pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <configuration>
          <archive>
            <!-- Création du Manifest pour la définition de la classe Main -->
            <manifest>
              <mainClass>com.App</mainClass>
            </manifest>
          </archive>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

Customisation du pom.xml – le build

Avec la modification du [pom.xml](#) contenant l'indication de la classe Main (principale), il faut relancer la génération du projet avec :

```
> mvn package
```

Ensuite il sera possible d'exécuter le projet Java généré. Pour se faire il faut adapter la commande :

```
> java -jar target/ESEO_TP_Maven-1.0-SNAPSHOT.jar
```

Customisation du pom.xml – le build

Filtrer des fichiers (ressources) :

Les fichiers ressources sont des fichiers qui n'ont pas à être compilés, mais simplement copiés dans le livrable généré par *Maven*.

Par convention, ces fichiers se trouvent dans le répertoire `src/main/resources` (répertoire à créer).

En allant plus loin, on peut utiliser ce répertoire pour des fichiers de type « properties ».

Exemple :

Un fichier « `information.properties` » qui contiendrait la version du projet et sa plateforme.

Customisation du pom.xml – le build

Exemple :

Un fichier « information.properties » qui contiendrait la version du projet et sa plateforme.

Implémentation :

- 1) Ajouter le fichier « information.properties » dans le répertoire **src/main/resources**
- 2) Dans ce fichier on va reprendre la logique du pom.xml pour les propriétés :

```
# Propriété contenant la version du projet  
org.example.demo.version=${project.version}
```

- 3) Ensuite il faut indiquer à *Maven*, donc dans le pom.xml, de filtrer les fichiers du répertoire **src/main/resources** avec la balise **<filtering>** :

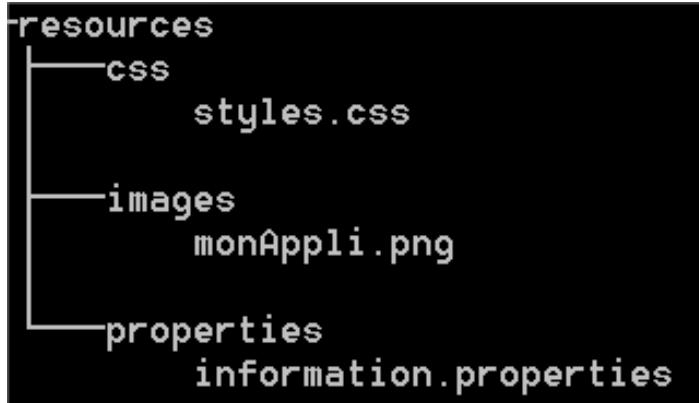
```
<project>  
  <build>  
    <resources>  
      <resource>  
        <directory>src/main/resources</directory>  
        <filtering>true</filtering>  
      </resource>  
    </resources>  
  </build>  
</project>
```

Customisation du pom.xml – le build

Complément :

Dans les gros projets, il peut y avoir beaucoup d'éléments dans le dossier resources.
Il est possible de faire des sous-répertoires pour une meilleure organisation.

Exemple :



Dans le pom.xml :

```
<project>
  ...
  <build>
    <resources>
      <resource>
        <directory>src/main/resources/properties</directory>
        <filtering>true</filtering>
      </resource>
      <resource>
        <directory>src/main/resources/images</directory>
        <filtering>false</filtering>
      </resource>
      <resource>
        <directory>src/main/resources/css</directory>
        <filtering>false</filtering>
      </resource>
    </resources>
  </build>
  ...
</project>
```

The code snippet shows the configuration for the 'build/resources' section of a Maven pom.xml file. It defines three resources: one from the 'properties' directory with filtering enabled ('true'), one from the 'images' directory with filtering disabled ('false'), and one from the 'css' directory with filtering disabled ('false'). The XML uses standard Maven conventions like 'src/main/resources' for source paths and 'true/false' for filtering flags.

Customisation du pom.xml – les profils

Dernier point important dans *Maven*, il s'agit des profils. Via l'élément profiles, ce dernier permet de créer des options dans le build Maven.

A minima dans les projets on retrouve 2 environnements, développement et production.

L'avantage est de pouvoir embarquer dans le projet, des fichiers de configurations différents en fonction des plateformes. Ainsi pas besoin de faire de modification manuelle qui sont sources d'erreurs en fonction de la version de livraison.

Toujours dans le répertoire [src/main/resources](#), on peut donc définir des sous-répertoires pour chacun des environnements.

Par exemple :

dev

prod

Customisation du pom.xml – les profils

```
<project>
    <!-- ===== -->
    <!-- Profils -->
    <!-- ===== -->
    <profiles>
        <profile> <!-- Profil pour l'environnement de dev -->
            <id>dev</id>
            <build>
                <resources>
                    <resource>
                        <directory>src/main/resources/conf-test</directory>
                    </resource>
                </resources>
            </build>
        </profile>
        <profile> <!-- Profil pour l'environnement de production -->
            <id>prod</id>
            <build>
                <resources>
                    <resource>
                        <directory>src/main/resources/conf-prod</directory>
                    </resource>
                </resources>
            </build>
        </profile>
    </profiles>
</project>
```

Customisation du pom.xml – les profils

Il ne reste qu'à mettre les fichiers de configuration de dev et de production dans leur répertoire ressource respectif et d'activer le bon profil lors du lancement du build Maven grâce à l'option :

-P

Exemple pour le dev :

```
> mvn package -P dev
```

Exemple pour la prod :

```
> mvn package -P prod
```

Customisation du pom.xml – les profils

Activation automatique des profils :

Il est aussi possible d'activer automatiquement en fonction de certains critères définis dans la sous-balise `<activation>`.

Exemple :

```
...
<!-- Profil activé automatiquement si la version du JDK est 1.8 et sous-versions mineures (ex 1.8.0_1) -->
<profile>
    <activation>
        <jdk>1.8</jdk>
    </activation>
...
</profile>
<!-- Profil activé automatiquement si la propriété système "environnement" vaut "test" -->
<profile>
    <activation>
        <property>
            <name>environnement</name>
            <value>test</value>
        </property>
    </activation>
...
</profile>
...
```

Dans cet exemple, pour activer le deuxième profil, il faut lancer *Maven* avec la commande suivante :
 > mvn package -D**environnement**=test

Customisation du pom.xml – Assembly

Descripteur Maven Assembly :

Le plugin Maven Assembly est un plugin qui vous permet de créer toutes les distributions que vous voulez pour vos applications.

Exemple :

```
...
<build>
  <plugins>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <version>2.6</version>
      <configuration>
        <descriptors>
          <descriptor>src/main/assembly/assembly.xml</descriptor>
        </descriptors>
        <finalName>MonLivrable</finalName>
      </configuration>
    </plugin>
  </plugins>
</build>
...
```

Dans cet exemple, il faut aussi créer un fichier [assembly.xml](#)

Customisation du pom.xml – Assembly

```
<assembly xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.2" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.2 http://maven.apache.org/xsd/assembly-1.1.2.xsd">
    <id>${version}</id>
    <formats> <format>tar.gz</format> </formats>
    <includeBaseDirectory>false</includeBaseDirectory>

    <fileSets>
        <fileSet>
            <directory>target/</directory>
            <outputDirectory>/batch/</outputDirectory>
            <includes> <include>Batch_Java*.jar</include> </includes>
            <fileMode>755</fileMode>
        </fileSet>
        <fileSet>
            <directory>src/main/shell</directory>
            <outputDirectory>/batch/</outputDirectory>
            <includes> <include>*.sh</include> </includes>
            <fileMode>755</fileMode>
            <lineEnding>unix</lineEnding>
        </fileSet>
        <fileSet>
            <directory>src/main/resources</directory>
            <outputDirectory>/batch/config/</outputDirectory>
            <includes> <include>*.properties</include> </includes>
            <fileMode>755</fileMode>
            <lineEnding>unix</lineEnding>
        </fileSet>
    </fileSets>
</assembly>
```

Ici on génère un tar.gz avec :

- un *.jar et droit 755
- des *.sh, droit 755 et conversion unix
- des *.properties dans un dossier config, droit 755 et conversion unix

Les goals

Un **goal** est une tâche précise que Maven est en mesure de réaliser à partir des informations qu'il pourra trouver dans le fichier [pom.xml](#)

Un **goal** est lié à un cycle de vie par défaut définit les étapes suivantes :

- validate => Valide que le projet est correctement défini
- compile => Compile les sources
- test => Lance les tests unitaires
- package => Prépare la distribution du projet. (archives Jar, War, Ear...)
- integration-test => Lance les tests d'intégration
- verify => Lance des tests de validation du package créé.
- install => Installe le package en local sur la machine pour pouvoir être réutilisé comme dépendance.
- deploy => Déploie le package sur un serveur pour qu'il puisse être réutilisé par tout le monde.

Note : Il est possible d'enchaîner deux ou plusieurs goals

Projet multi-modules Maven

Il est possible de créer un projet *Maven* (dit principale) qui englobe des sous projet *Maven*. Pour se faire il faut créer le projet princial :

```
> mvn archetype:generate \
  -DarchetypeGroupId=org.apache.maven.archetypes \
  -DarchetypeArtifactId=maven-archetype-quickstart \
  -DarchetypeVersion=1.1 \
  -DgroupId=com.exemple \
  -DartifactId=multiModules \
  -Dversion=1.0-SNAPSHOT
```

Supprimer la partie **src**, qui n'aura pas d'utilité :

```
> cd multiModules
> rm -r ./src
```

Projet multi-modules Maven

Editer le **pom.xml**, pour utiliser le packaging POM au lieu de JAR :

```
<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>org.example</groupId>
    <artifactId>multiModules</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>pom</packaging>
    ...
</project>
```

Et supprimer les dépendances inutiles :

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

Projet multi-modules Maven

Une fois le projet principale créé, vient le tour des sous-projets.

Exemple avec un sous-projet batch et web :

```
> mvn -B archetype:generate \
      -DarchetypeGroupId=org.apache.maven.archetypes \
      -DarchetypeArtifactId=maven-archetype-quickstart \
      -DarchetypeVersion=1.1 \
      -DgroupId=com.exemple \
      -DartifactId=multiModules-batch \
      -Dpackage=org.exemple.demo.batch

> mvn -B archetype:generate \
      -DarchetypeGroupId=org.apache.maven.archetypes \
      -DarchetypeArtifactId=maven-archetype-webapp \
      -DgroupId=com.exemple \
      -DartifactId=multiModules-webapp \
      -Dpackage=org.exemple.demo.webapp
```

Note : Pour le 2^{ème} on utilisation de l'archetype "webapp"

Projet multi-modules Maven

Le fichier **pom.xml** du fichier principale devrait être modifié :

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
    <!-- Informations du projet -->
    <!-- ===== Informations Maven ===== -->
    <groupId>com.example</groupId>
    <artifactId>multiModules</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>pom</packaging>

    <!-- ===== Informations générales ===== -->
    <name>ticket</name>
    <url>http://maven.apache.org</url>

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

    <!-- Modules -->
    <modules>
        <module>multiModules-batch</module>
        <module>multiModules-webapp</module>
    </modules>
</project>
```

Projet multi-modules Maven

Le gros point fort est de n'avoir à exécuter seulement le POM principale, qui va lui-même déclenché la génération des POM des sous-projets.

Portée des dépendances Maven

En fonction de l'environnement, on ne souhaite pas toujours inclure toutes les dépendances.

Par exemple :

En développement : connexion sur BDD MySQL

En production : connexion sur BDD Oracle

Pour indiquer cela à Maven, les dépendances sont attachées à un scope.

Par défaut ce scope est **compile**.

En détail :

compile => Ce scope indique que la dépendance est utilisée lors de la compilation et sera accessible dans tous les contextes

test => Cela indique que les dépendances ne seront accessible que lors de la compilation des tests et leur exécution.

provided => indique que la dépendance est disponible à la compilation, mais elle devra être fournie par le contexte d'exécution (le serveur d'application par exemple)

runtime => en revanche, la dépendance n'est pas accessible lors de la compilation, mais elle est disponible à l'exécution.

Gestion des versions (Snapshot et Release)

Dans le domaine du développement informatique, un livrable est défini par un numéro de version.

Maven ajoute un concept de version à savoir **snapshot** et **release**.

Pour la **release** :

Pour garantir au maximum qu'une version ne changera plus, il y a un principe de base : une version d'une dépendance est conventionnellement immuable. Cela veut dire qu'une fois la version publiée, elle ne sera plus modifiée. Si un changement doit intervenir dans le code, une nouvelle version sera publiée.

Pour le **snapshot** :

Dans le cas d'un élément en cours de développement, la version de cet élément sera une version snapshot. Cela est précisé dans *Maven* avec le suffixe **-SNAPSHOT** à la fin de la version.

Une version snapshot peut être modifiée à tout moment et elle fait donc l'objet d'un traitement particulier dans *Maven*.

Contrôle du code

Contrôle du code

Génération de rapport :

- Générer la javadoc :

Ajoutez le code suivant dans le [pom.xml](#) :

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-javadoc-plugin</artifactId>
      <version>2.9.1</version>
    </plugin>
  </plugins>
</reporting>
```

Puis lancez :

```
> mvn site
```

Cela va créer un site Web lié au projet. Par défaut, les goals Maven générant des fichiers travaillent dans le dossier [target](#) se trouvant au même niveau que le fichier [pom.xml](#).

Allez dans le dossier [target/site](#) et ouvrez le fichier [index.html](#). Vous pouvez regarder la Javadoc générée en cliquant sur Project reports.

Contrôle du code

- Valider la qualité du code avec le plugin checkstyle :

Ajoutez à la section `<plugins>` dans `<reporting>` le plugin checkstyle :

```
<reporting>
  <plugins>
    ...
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-checkstyle-plugin</artifactId>
      <version>2.11</version>
    </plugin>
  </plugins>
</reporting>
```

Lancez :

 > mvn clean site (le goal clean vide le dossier target).

Une nouvelle section Checkstyle a été ajouté dans Project reports.

Contrôle du code

Exercice :

- Ajouter Checkstyle à Eclipse
- Ajouter PMD à Eclipse

Contrôle du code

Test :

Ajouter la classe Java suivante « `Addition.java` » dans `src/main/java` (et package `com`) :

```
package com;

class Addition {
    public Long calculer(Long a, Long b) {
        return a+b;
    }

    public Character lireSymbole() {
        return '-';
    }
}
```

Créer une page test (nommé « `AdditionTest` ») affin de tester les deux méthodes.