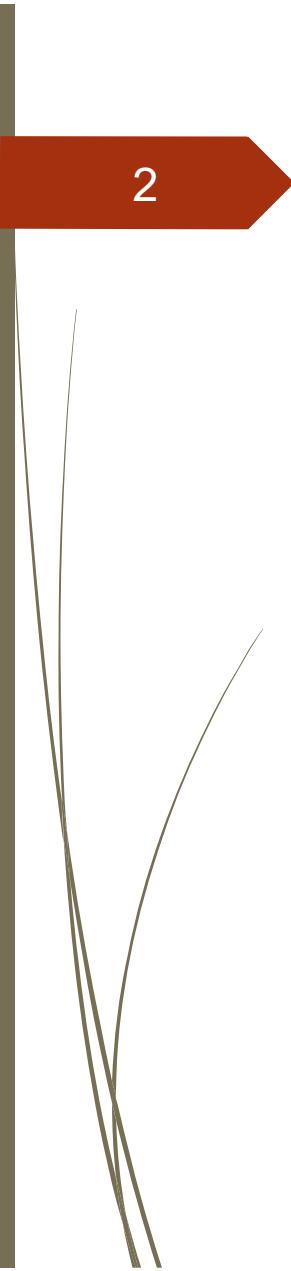




MAVEN

1



Sujets abordés

Maven

- 1. Définition**
- 2. Installation**
- 3. Le POM (ProjectObjectModel)**
- 4. Les Archétypes**
- 5. Cycle de vie du build**
- 6. Options d'exécution**
- 7. Maven dans son outil de dev**
- 8. Rôle d'un Nexus**

Introduction

3

**Maven est un
outil de BUILD**



Maven - Notion de Build

- « Build » : ensemble d'opérations destinées à produire un artefact logiciel
- Builder un projet ⇔ fournir un livrable pour le test, la prex, la production
 - ▶ Potentiellement re-nommer / supprimer / ajuster les fichiers en fonction de la cible
 - ▶ Compiler le code Java ⇔ générer des fichiers class
 - ▶ Fabriquer le livrable (JAR, WAR, EAR, RAR, ...)
- Maven remplace les make file C / C++
- '**Avant**' Maven il y avait ANT
 - ▶ <https://ant.apache.org/>
- '**Après**' Maven il y aura peut être Gradle
 - ▶ <https://gradle.org/>



Rappel sur les différents livrables Java

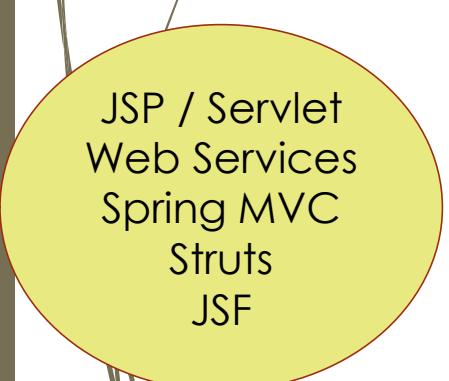
➤ Différents livrables en Java

► **JAR** : librairie de code Java, il contient

- ❑ Assurément des fichiers compilés Java = .class
- ❑ Potentiellement des fichiers de configurations = .properties, .xml
- ❑ Potentiellement des fichiers binaires utiles à l'applications

► **WAR** : application web Java, il contient

- ❑ Assurément des fichiers compilés Java = .class
- ❑ Assurément des fichiers web : HTML, CSS, JavaScript, JSP
- ❑ Assurément des librairies Java (JAR)
- ❑ Potentiellement des fichiers de configurations = .properties, .xml
- ❑ Potentiellement des fichiers binaires utiles à l'applications



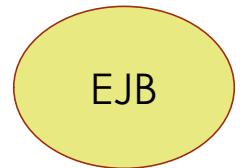
JSP / Servlet
Web Services
Spring MVC
Struts
JSF

Rappel sur les différents livrables Java

➤ Différents livrables en Java (suite)

► **EAR** : application d'entreprise Java, il contient

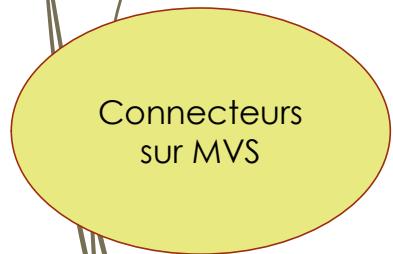
- ❑ Assurément des fichiers compilés Java = .class
- ❑ Assurément des applications web Java (WAR)
- ❑ Potentiellement des fichiers de configurations = .properties, .xml
- ❑ Potentiellement des fichiers binaires utiles à l'applications



EJB

► **RAR** : connecteur Java, il contient

- ❑ Assurément des fichiers compilés Java = .class
- ❑ Assurément des fichiers binaires compilés pour chaque plateforme cible (DLL, SO, ..)
- ❑ Potentiellement des fichiers de configurations = .properties, .xml
- ❑ Potentiellement des fichiers binaires utiles à l'applications



Connecteurs
sur MVS

Rappel sur les différents livrables Java

7

➤ Différents livrables en Java (suite)

- ➡ **PAR** : application portail (plus vraiment à la mode)
 - ❑ Assurément des fichiers compilés Java = .class
 - ❑ Assurément des fichiers web : HTML, CSS, JavaScript, JSP
 - ❑ Assurément des librairies Java (JAR)
 - ❑ Potentiellement des fichiers de configurations = .properties, .xml
 - ❑ Potentiellement des applications web Java (WAR)
 - ❑ Potentiellement des fichiers binaires utiles à l'applications

Maven - Plus qu'un outil de Build

- Mais Maven va faire plus que de fabriquer votre livrable
- Il va faire passer les tests automatisés et vérifier le résultat
 - ➡ Si tous les tests sont ok = on produit un livrable, sinon pas de livrable
- Il va générer des rapports divers et variés
 - ➡ Qui à fait quoi et quand sur CVS, SVN, GIT ... (votre SCM)
 - ➡ Les évolutions apportées à l'application, les corrections de bug
 - ➡ La qualité du logiciel
 - ➡

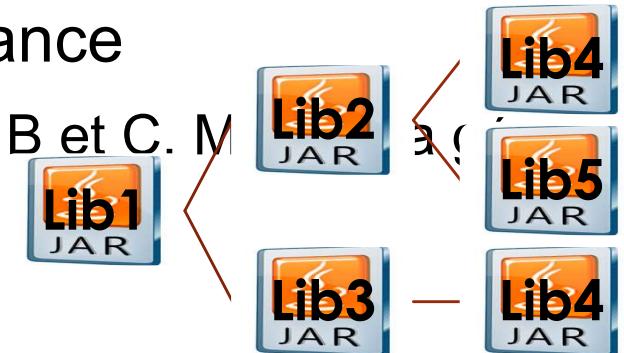
Maven - Plus qu'un outil de Build

- Il va aussi gérer les dépendances entre
 - ➡ Votre projet et les librairies externes dont il a besoin (les framework Java standard)
 - ➡ Vos projets

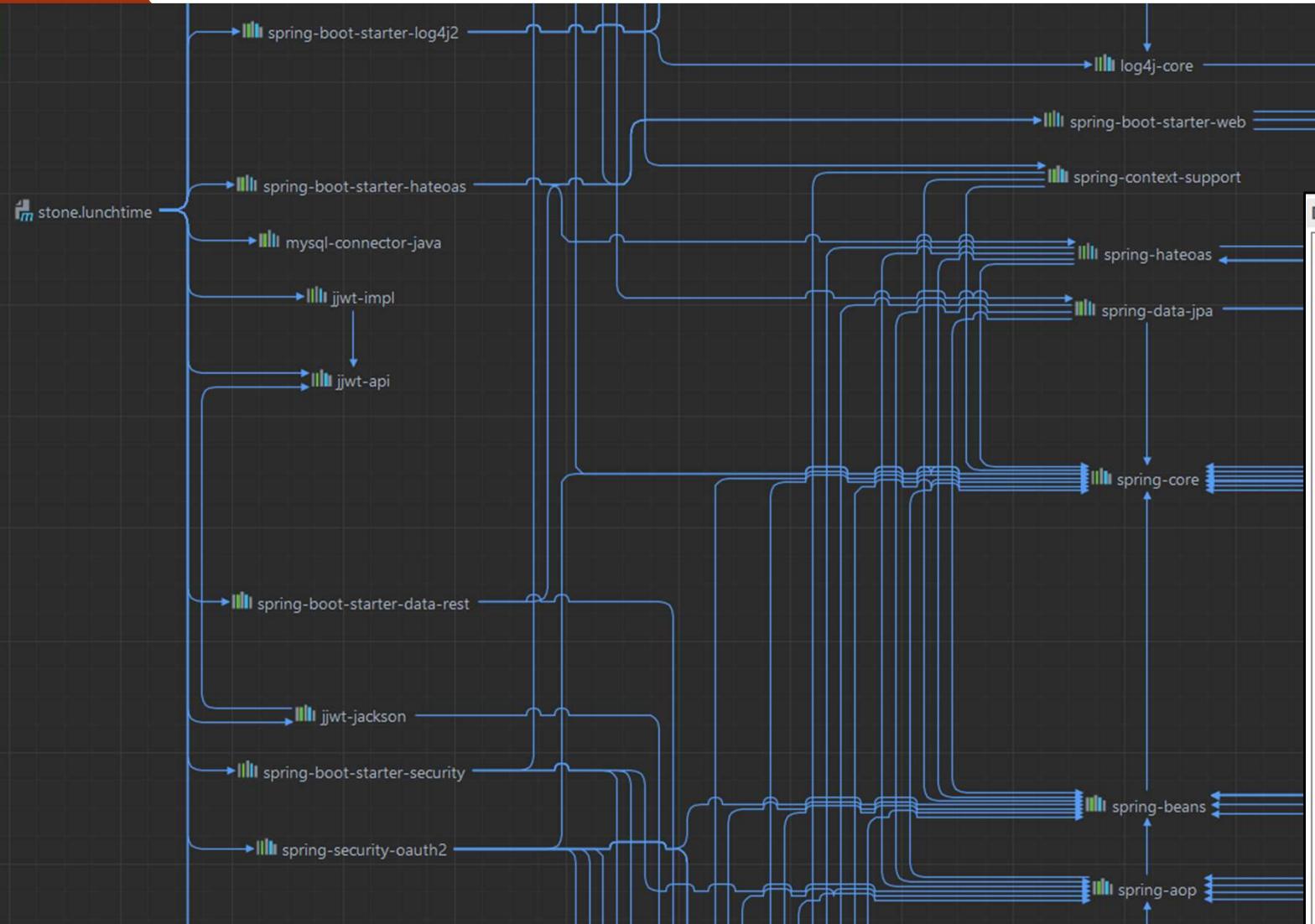
Notion de DEPENDANCE

➤ Maven permet donc

- ▶ A chaque développeur de builder et de travailler sur son poste en garantissant une unicité des build
 - Avant : le résultat était dépendant de l'environnement (la configuration de sa machine) du développeur
- ▶ D'automatiser les builds en y ajoutant des étapes qualités / reporting
- ▶ De simplifier les contraintes de dépendance
 - Si un framework A a besoin des framework B et C. Maven gère les versions et la dépendance elle-même
 - Il gère la transitivité des dépendances



Notion de DEPENDANCE



| Dependency Hierarchy | |
|----------------------|---|
| \ | spring-boot-starter-log4j2 : 2.6.4 [compile] |
| \ | \ log4j-slf4j-impl : 2.17.1 [compile] |
| \ | \ slf4j-api : 1.7.36 (managed from 1.7.25) [compile] |
| \ | \ log4j-api : 2.17.1 [compile] |
| \ | \ log4j-core : 2.17.1 [runtime] |
| \ | \ log4j-core : 2.17.1 [compile] |
| \ | \ log4j-api : 2.17.1 [compile] |
| \ | \ log4j-jul : 2.17.1 [compile] |
| \ | \ log4j-api : 2.17.1 [compile] |
| \ | \ jul-to-slf4j : 1.7.36 [compile] |
| \ | \ slf4j-api : 1.7.36 [compile] |
| \ | \ spring-boot-starter-mail : 2.6.4 [compile] |
| \ | \ spring-boot-starter : 2.6.4 [compile] |
| \ | \ spring-boot : 2.6.4 [compile] |
| \ | \ spring-boot-autoconfigure : 2.6.4 [compile] |
| \ | \ spring-boot : 2.6.4 [compile] |
| \ | \ jakarta.annotation-api : 1.3.5 [compile] |
| \ | \ spring-core : 5.3.16 [compile] |
| \ | \ snakeyaml : 1.29 [compile] |
| \ | \ spring-context-support : 5.3.16 [compile] |
| \ | \ spring-beans : 5.3.16 [compile] |
| \ | \ spring-context : 5.3.16 [compile] |
| \ | \ spring-core : 5.3.16 [compile] |
| \ | \ jakarta.mail : 1.6.7 [compile] |
| \ | \ jakarta.activation : 1.2.2 (managed from 1.2.1) [compile] |
| \ | \ spring-boot-starter-data-jpa : 2.6.4 [compile] |

Notion de DEPENDANCE

12

- La gestion des packages / dépendance en Maven se réalise grâce à un **repository** accessible via Internet et dans lequel il va chercher les paquets **requis**.

- Vous pouvez naviguer dessus manuellement via, entre autre, le site : <https://search.maven.org/>



Notion de DEPENDANCE

13

- TOUS LES PROJETS OPEN SOURCE IMPORTANTS ont adopté Maven et déposent leurs jars et la description de leurs propres dépendances sur ce repository
- C'est un standard de l'industrie
- Il existe d'autres repository gérés par différentes organisations
- Vous pouvez aussi définir votre propre repository en interne

Notion de PLUGIN

- Maven est aussi un micro kernel extensible via des plugins
 - ➡ Le cœur de Maven a très peu de fonctions.
 - ➡ Il doit être étendu par des composants spécialisés, spécifiquement conçus pour être invocables par son noyau.
- Ces composants sont appelés **PLUGINS**
- Ces plugins sont des **artefacts** particuliers.
- Comme tous les **artefacts** ils sont dans le repository central
- Comme toutes les dépendances, le noyau les télécharge dès qu'il en a besoin

Notion de PLUGIN

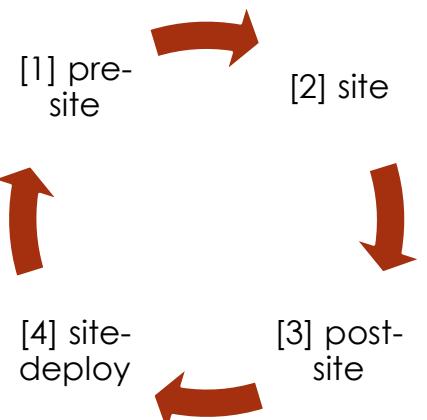
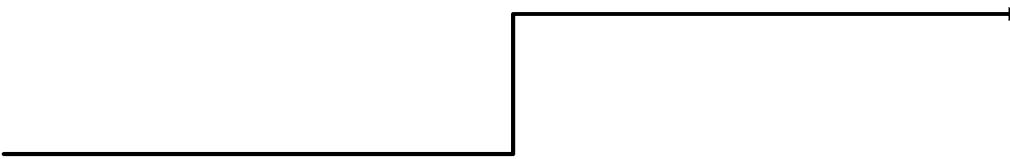
15

- Chaque plugin expose un nombre de services invocables
- Chacun de ces services est appelé **GOAL**
- Ces GOALS peuvent être invoqués via la ligne de commande
`>mvn nomDuPlugin:nomDuGoal`
- Chaque Goal est implémenté par une classe java appelée familièrement MOJO.
 - ➡ Maven plain Old Java Object
- Un plugin est un packaging d'un ensemble de MOJOS

Cycle de vie

- Au cœur de Maven on trouve la notion de ProjectLifeCycle
- Un ProjectLifeCycle est une succession ordonnée de phases
- Le noyau Maven pilote des ProjectLifeCycle et comprend la définition de 3 ProjectLifeCycle, dont un par défaut.
 - ▶ clean
 - ▶ default
 - ▶ site
- Le cycle de vie projet par défaut est le cycle de vie d'un build ; il décrit les phases successives d'un processus de build

```
>mvn clean
```



Cycle de vie de DEFAULT

- Les phases standards de **default**
 - ▶ **validate** : valide le projet et vérifie que tout est présent
 - ▶ **compile** : compile le projet
 - ▶ **test** : test le code compile avec un framework de test disponible
 - ▶ **package** : prend le code compile et fabrique un livrable (jar, war, ...).
 - ▶ **verify** : vérifie que les critères de qualité sont vérifié
 - ▶ **install** : installe le livrable dans un repository maven local.
 - ▶ **deploy** : déploie le livrable dans un repository maven global

```
>mvn clean package
```

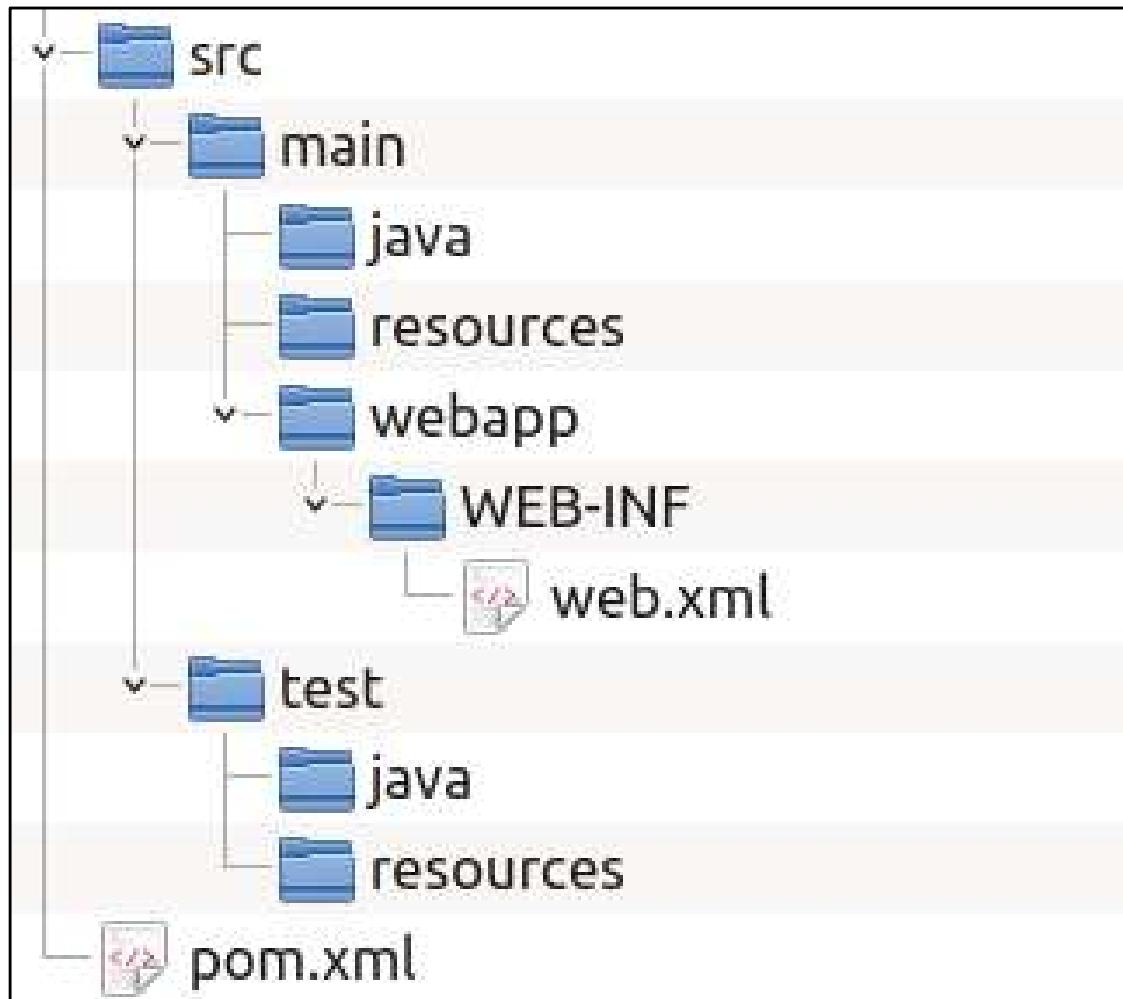
- Toutes les phases :
 - ▶ http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference

Organisation des projets

18

- Les différents goals invoqués durant le build vont avoir besoin de certaines informations essentielles :
 - ▶ Où se trouve le code source ?
 - ▶ Où ranger les classes compilées ?
 - ▶ Où déposer l'artefact généré ?
- Pour éviter à chaque projet de devoir fournir ces informations, Maven définit un **layout standard** ⇔ un modèle de répertoires.
- En suivant ces **conventions**, pas besoin de passer par une fastidieuse phase de **configuration**

Organisation classique des projets

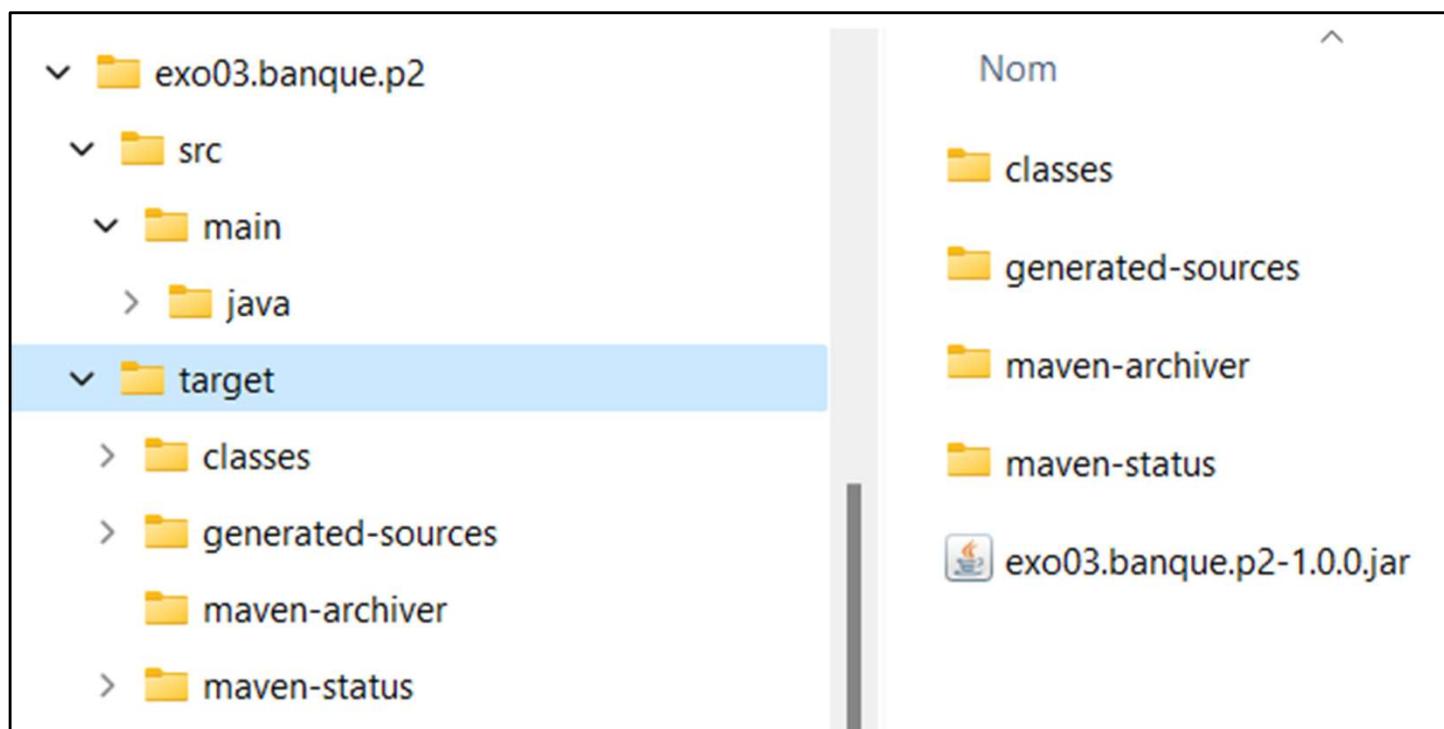


Organisation des projets : dossier target

- En plus des dossiers par défaut, Maven va en créer d'autres
- Le dossier **target** qui sera à la racine du projet sera le dossier 'bac à sable' de Maven
- Ce dossier NE DOIT JAMAIS être placé sur votre SCM (GIT/SVN/CVS ...)
 - ➡ il doit donc faire partie de .gitignore

Organisation des projets : dossier target

➤ Exemple



Organisation des projets : dossier target

- Le dossier target peut être supprimé sans crainte
- Il sera recréé automatiquement lors de vos appels à Maven
- C'est dans ce dossier que vous retrouverez le résultat de vos commandes Maven
 - ➡ Le livrable
 - ➡ Les rapports
 - ➡ Les sites web
 - ➡ ...

Première Synthèse

- On a donc :
 - ▶ Un **repository** central sur Internet
 - ▶ Une série de **plugins**, téléchargés au premier usage, qui sont des composants spécifiques et capables de certaines activités (compilation, lancement de tests, packaging ...) exposées sous forme de **goals**
 - Une **association** (binding) entre des **goals** et les **phases** du cycle de vie
 - ▶ Un **cycle de vie** par défaut qui définit le cycle de build de tout projet
 - ▶ Un **layout standard**
 - ▶ Une **interface en ligne de commande** qui permet d'exécuter un goal d'un plugin (mvn plugin:goal) ou tous les goals associés aux phases successives d'un cycle de vie projet (mvn phase)

Installation

- Pour installer Maven, dans le cas d'une utilisation en ligne de commande
 1. Installez un JDK sur votre machine
 2. Téléchargez Maven
 3. Dézippez Maven
 4. Ajoutez MAVEN_HOME
 5. Ajustez PATH
 6. Optionnel : configurez proxy/repository en fonction de vos contraintes internes

Installation

25

1. Installez un JDK sur votre machine

- ▶ Attention : Maven ne fonctionnera peut être pas sur la toute dernière version du Java ou une très vieille version du Java
 - Au minimum il vous faudra la version 1.7
- ▶ Lisez les préconisations en fonction de la version de Maven que vous désirez
- ▶ Rappels :
 - Il faut souvent les droits d'administrateurs pour installer un JDK
 - Pensez à spécifier votre variable JAVA_HOME
- ▶ <https://www.oracle.com/java/technologies/downloads/>

Installation

2. Téléchargez Maven sur le site de l'éditeur

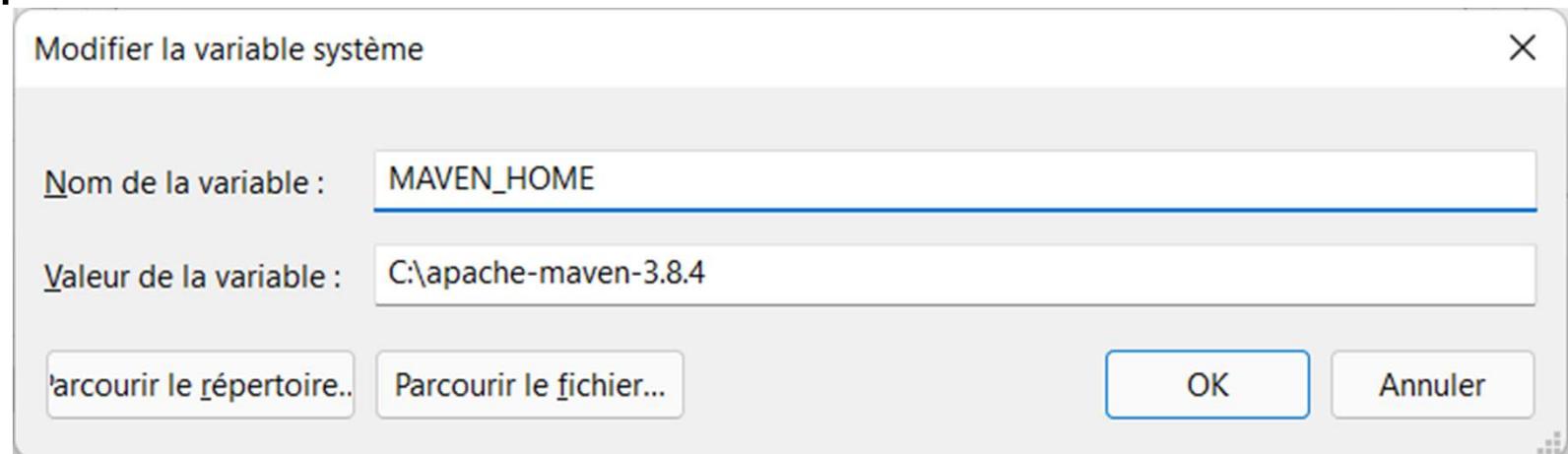
- ▶ <https://maven.apache.org/download.cgi#>
- ▶ Il n'y a pas d'archive spécifique en fonction de l'OS qui va faire tourner Maven car il est fait en Java
 - ZIP ou tar.gz en fonction de votre préférence
- ▶ Vous n'avez pas besoin des sources

Installation

27

3. Dézippez l'archive de Maven où vous voulez
4. Ajoutez la variable d'environnement système MAVEN_HOME (respectez les MAJUSCULES)

- ▶ Elle pointera sur le dossier d'installation de Maven
- ▶ Ex :

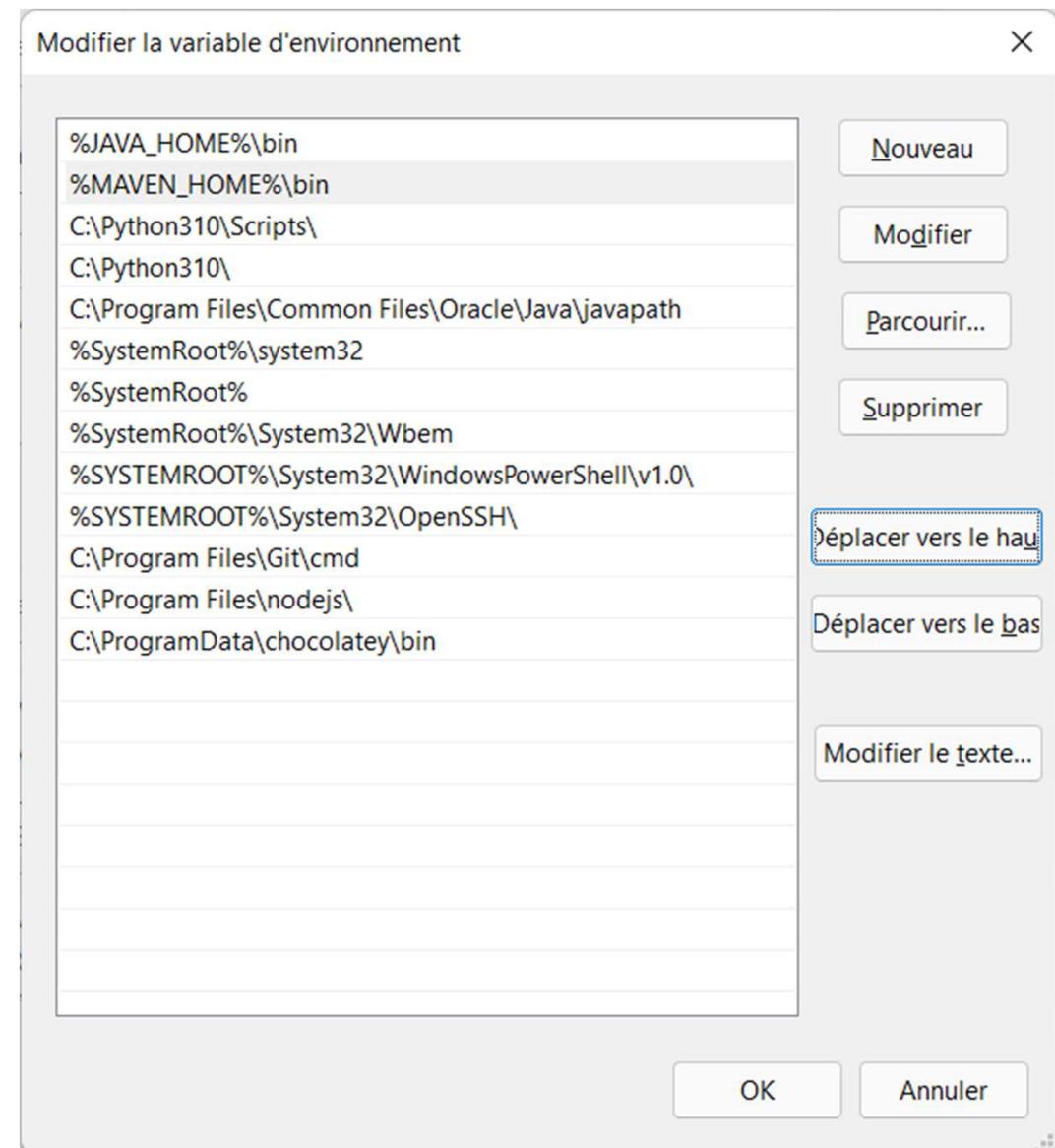


Installation

28

5. Ajustez la variable système PATH

- ▶ afin qu'elle contiennent le chemin vers l'exécutable Maven
- ▶ Notez que l'on a ajouté '\bin'



Installation

29

6. Optionnellement, ajustez la configuration

- ▶ Si vous faites usage de proxy de repository spécifiques
- ▶ Il vous faudra créer un fichier **settings.xml**
 - Il se trouvera dans le dossier [UTILISATEUR]/.m2/
- ▶ <https://maven.apache.org/settings.html>

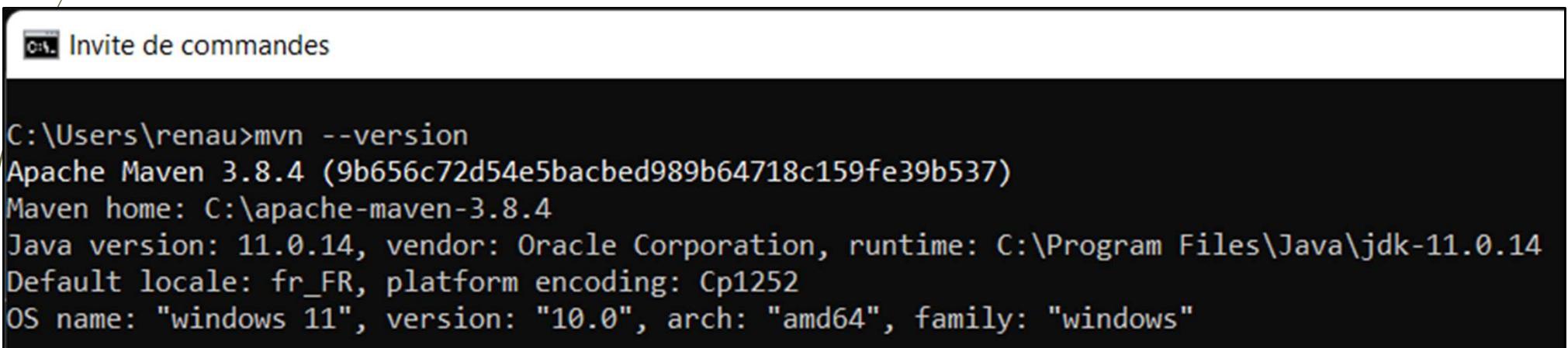
Installation

30

➤ Testez

- ▶ Ouvrez un invité de commande
- ▶ Tapez directement

```
>mvn --version
```



```
C:\Users\renau>mvn --version
Apache Maven 3.8.4 (9b656c72d54e5bacbed989b64718c159fe39b537)
Maven home: C:\apache-maven-3.8.4
Java version: 11.0.14, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk-11.0.14
Default locale: fr_FR, platform encoding: Cp1252
OS name: "windows 11", version: "10.0", arch: "amd64", family: "windows"
```

Le POM (Modèle Objet de Projet)

- Tout projet informatique est en général constitué de :
 - ▶ Code source
 - ▶ Documentation
 - ▶ Ressources (images, vidéos, fichiers de configuration...)
 - ▶ Dépendances externes (librairies de programmation tierce-partie)
- Pour être exploités ils doivent être :
 - ▶ compilés (code source java, .Net...)
 - ▶ vérifiés
 - ▶ packagés (jar, zip, ...)
 - Parfois de façons différentes en fonction de l'usage (clients différents, environnements techniques différents...)
 - ▶ Livrés / déployés aux utilisateurs finaux

Le POM (Modèle Objet de Projet)

32

- Chaque nouvel incrément du projet livré possède un numéro de « version »
 - ▶ Ce numéro augmente en général avec le temps, selon des règles spécifiques à chaque projet
 - ▶ Il permet d'identifier avec précision de quelle « version » du projet on parle
- Culturellement, un projet est souvent « rattaché » à une organisation
 - ▶ « Windows » -> « Microsoft »
 - ▶ « Tomcat » -> « Apache »
 - ▶ « Java » -> « Oracle »

Le POM (Modèle Objet de Projet)

33

- Maven propose de formaliser et recenser un certain nombre de ces informations dans un « modèle objet de projet »
 - ▶ Où sont les sources ?
 - ▶ Quelle est le numéro de version du projet ?
 - ▶ Quelles sont les dépendances du projet ?
 - ▶ ...
- L'objectif étant de centraliser toutes ces informations utiles pour pouvoir les exploiter facilement et effectuer des tâches du cycle de vie projet
 - ▶ Compilation, packaging, exécution de tests automatisés, copie sur une machine distante...

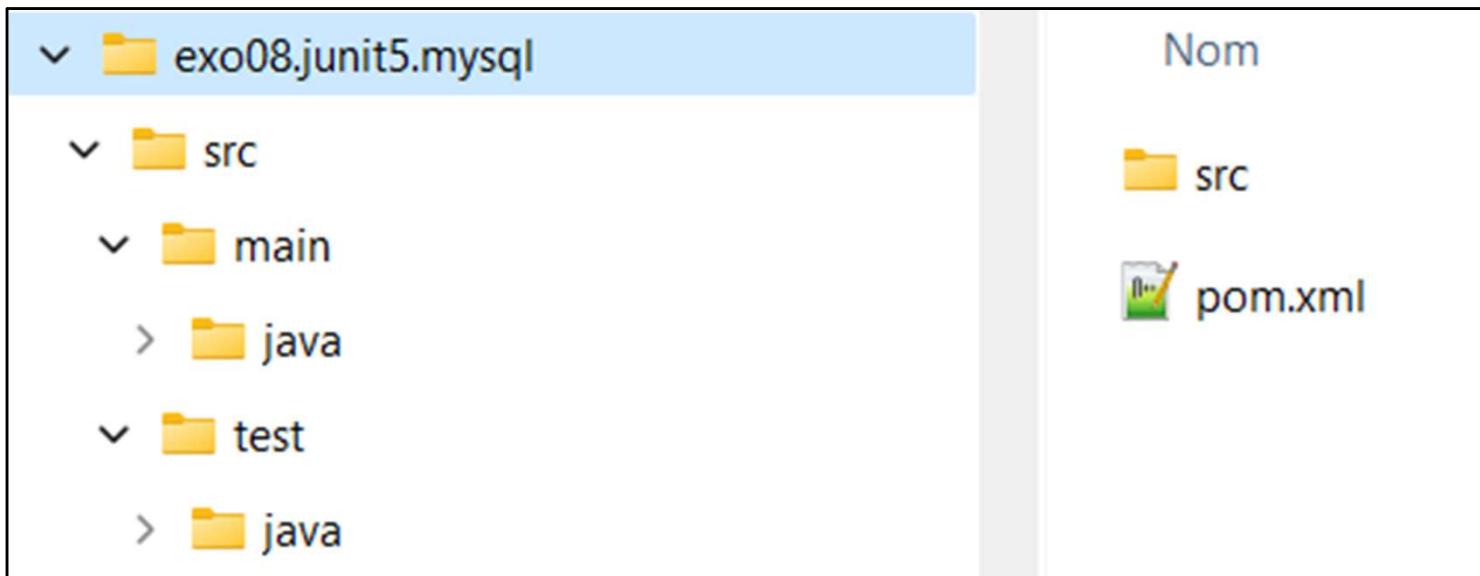
Le POM (Modèle Objet de Projet)

- Ce modèle objet de projet (POM – Project Object Model) est décrit dans un fichier xml de configuration associé au projet
 - Ce fichier fait partie du code source de l'application
 - il se trouve à la racine de celui-ci
 - C'est ce fichier que va lire Maven pour prendre connaissance de tous les aspects utiles à son fonctionnement dans le cadre du projet
 - Ce fichier permet aussi d'activer et configurer certaines tâches (effectuées par les plugins) – **mais ce n'est pas un fichier de scripting, comme ant !**

Le POM (Modèle Objet de Projet)

35

- Le POM est donc décrit dans un fichier xml
 - ➡ Nommé **pom.xml** depuis la version 2 de Maven
 - ➡ Chaque projet, ou sous-module de projet (nous y reviendrons), possède son propre pom à la racine du projet



Le POM (Modèle Objet de Projet)

36

- Exemple de fichier pom minimaliste
 - ▶ C'est un fichier xml encodé en UTF-8
 - ▶ Normalisé par un schéma xsd

```
<?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/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>com.training</groupId>
    <artifactId>maven-example</artifactId>
    <version>1</version>

</project>
```

Le POM (Modèle Objet de Projet)

37

- On y trouve :
 - ▶ **groupId** : permet d'identifier le projet comme appartenant à un groupement plus vaste (nom d'une organisation, nom d'un service, d'un projet plus global dans le cas d'un sous projet...)
 - ▶ **artifactId** : détermine le nom précis du projet au sein de ce groupId
 - ▶ **version** : quelle est la version courante du projet
- Ce sont les **seules** informations à faire figurer dans un pom pour que Maven puisse fonctionner :
 - ▶ Les autres informations ou actions à effectuer lors d'une build, comme la localisation des fichiers sources à compiler, ou le type de packaging à produire, sont pré-configurées par défaut

Le POM (Modèle Objet de Projet)

38

- Le triplet **groupId:artifactId:version** forme ce que l'on appelle les **coordonnées** du projet
 - ▶ Dans l'exemple précédent, notre projet est donc identifié par le triplet
 - ❑ com.training:maven-example:1
- Lorsque nous aborderons la problématique des dépendances, nous verrons que celles-ci sont elles aussi identifiées par leur coordonnées
 - ▶ Ex :
 - ❑ org.springframework:spring-beans:4.3.16-RELEASE
 - ❑ org.hibernate:hibernate-entitymanager:5.2.16.Final
 - ▶ Ce système de coordonnées est présent dans de nombreux systèmes de gestion de dépendances (linux, ivy, sbt, gradle...)

Le POM (Modèle Objet de Projet)

39

➤ Le groupId

- ▶ Est en général constitué du nom de l'entreprise précédé de com, org, fr...
 - com.sqli, fr.sogem, fr.probtp

➤ L'artifactId

- ▶ Est en général le nom du projet
 - datavance, hermes ...
- ▶ Peut-être composé du nom du projet et du sous projet, séparés par un '-'
 - datavance-front, datavance-back, hermes-web-services...

Le POM (Modèle Objet de Projet)

➤ La **version**

- ▶ Est en général composé d'un numéro à plusieurs chiffres
 - 1, 3.0, 2.5.5
- ▶ Ce numéro peut être étendu pour encoder des informations plus précises
 - 2.5-BETA
- ▶ A chaque projet / organisation de poser les règles de construction du numéro
 - Ex : Quand incrémenter le numéro majeur, le numéro mineur ?
- ▶ Maven n'impose pas une nomenclature fixée ; cependant, il est conseillé de respecter certaines règles ...

Le POM (Modèle Objet de Projet)

41

- Maven propose et comprend une nomenclature précise pour le numéro de version
 - <major version>.<minor version>.<incremental version>-<qualifier>
- Elle n'est pas obligatoire
- Mais il ne faut pas oublier que certains plugins se basent sur cette nomenclature pour effectuer certains choix, ou actions
 - Par exemple, le plugin delivery doit choisir un nouveau numéro de version automatiquement pour le projet, lors d'une release
 - Passer de 1.0.1 à 1.0.2 automatiquement est facile, mais qu'y a-t-il après VER12b, par exemple ?
 - Ou déterminer si une version d'un projet est plus récente qu'une autre (tri des numéros de version)

Le POM (Modèle Objet de Projet)

42

- La chaîne **-SNAPSHOT**, dans le numéro de version a une signification spéciale pour maven
 - ▶ **SNAPSHOT** est un **qualifier**
 - ▶ Il indique que le projet est **en cours de développement** et donc potentiellement **instable**
 - ❑ Aussi, avant toute release / livraison officielle, il convient de passer le numéro de version en numéro « stable » , en retirant le **-SNAPSHOT**
 - 1.2-SNAPSHOT -> 1.2, (ou même 1.2-RELEASE, ...)
 - ▶ Si l'on souhaite quand même déployer (plugin ‘release’) une version **SNAPSHOT**, maven va substituer cette chaîne par un timestamp de l'heure de release
 - 1.2-SNAPSHOT -> 1.2-20080207-230803-1
 - ❑ En cas de dépendance **-SNAPSHOT**, c'est la version la plus récente qui sera alors utilisée (tri grâce au timestamp)

Super POM

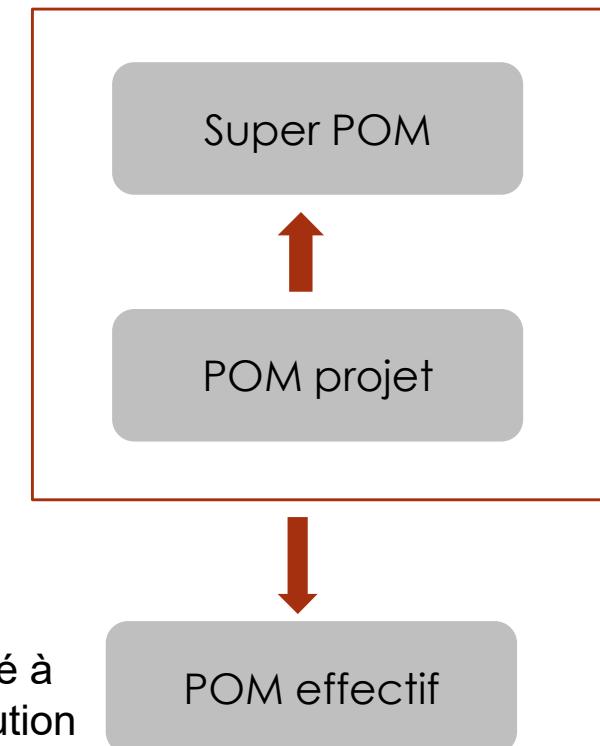
- Nous avons vu que maven adoptait un certain nombre de conventions par défaut pour la majorité des informations projet qui ne sont pas précisées dans notre POM
- Un certain nombre d'informations sont issues d'un Super POM
 - ▶ Ce POM est un POM ancêtre des POM projets
 - ▶ Tout POM projet hérite de celui-ci
 - ▶ Des valeurs par défauts y sont renseignées
- Le Super POM est contenu dans les librairies de maven
 - ▶ Dans le jar maven-model-builder-xxx
 - ▶ Sous le nom pom-4.0.0.xml

Super POM

- Le Super POM sert de base à maven
- Il est fusionné avec le pom projet pour produire le « pom effectif »
- Le pom effectif est le résultat de la fusion entre le super POM et le POM projet
- La commande :

```
>mvn help:effective-pom
```

permet d'afficher le pom issu de ce merge.



Super POM

- L'affichage du POM effectif nous a montré que tout POM définit en réalité de nombreuses informations
 - ▶ Ces informations par défaut sont toutes surchargeables dans notre propre pom projet (il suffit de les ré-écrire), mais...
 - ▶ Ces informations par défaut forment les conventions maven : les adopter permet à toute personne familière de maven de pouvoir prendre en main le projet très facilement
 - ❑ Les choses sont « toujours à la même place » (layout standard)
 - ❑ Builder , packager, compiler revient à toujours exécuter la même commande, quel que soit le projet
 - ❑ Customiser un aspect du build projet revient toujours à la même chose
 - Configurer si besoin le bon plugin
 - Ajouter un nouveau plugin (toujours de la même façon, dans le pom)

Structure générale du POM

➤ Observons les différentes parties du POM effectif de notre POM minimaliste :

```
<?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/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.training</groupId>
    <artifactId>maven-example</artifactId>
    <version>1</version>

    <repositories> ... </repositories>

    <pluginRepositories> ... </pluginRepositories>

    <build> ... </build>

    <dependencyManagement> ... </dependencyManagement>

    <dependencies> ... </dependencies>

    <reporting> ... </reporting>
</project>
```

Structure générale du POM

➤ **Repositories et pluginRepositories**

- ▶ Indique à maven où chercher les dépendances projet ainsi que les plugins maven à exécuter lors du build
- ▶ Détailé au chapitre de gestion des dépendances

➤ **Reporting**

- ▶ Indique où doivent-être générés les rapports produit par maven lors de l'activation d'un build particulier
- ▶ Détailé au chapitre Génération de site

Structure générale du POM

➤ build

- ▶ Cette partie contient le cœur de la configuration du processus de transformation (build) du projet en artefact livrable

```
<build>
  <sourceDirectory>(...)solutions\pom-minimaliste\src\main\java</sourceDirectory>
  <scriptSourceDirectory>(...)solutions\pom-minimaliste\src\main\scripts</scriptSourceDirectory>
  <testSourceDirectory>(...)solutions\pom-minimaliste\src\test\java</testSourceDirectory>
  <outputDirectory>(...)solutions\pom-minimaliste\target\classes</outputDirectory>
  <testOutputDirectory>(...)solutions\pom-minimaliste\target\test-classes</testOutputDirectory>
  <resources> (...) </resources>
  <testResources> (...) </testResources>
  <directory>(...)solutions\pom-minimaliste\target</directory>
  <finalName>maven-example-1</finalName>
  <pluginManagement> (...)</pluginManagement>
  <plugins> (...)</plugins>
</build>
```

Structure générale du POM

- **sourceDirectory**
 - ▶ Où se trouve le code source à compiler
- **scriptSourceDirectory**
 - ▶ Si le projet contient des scripts (sh, bat...), indique où se trouvent ceux-ci
- **testSourceDirectory**
 - ▶ Où se trouve le code source des tests unitaires à compiler
- **directory**
 - ▶ Dans quel répertoire de travail temporaire doivent être déposés les fichiers résultants du build
- **outputDirectory**
 - ▶ Où est déposé le résultat de la compilation des sources
- **testOutputDirectory**
 - ▶ Où est déposé le résultat de la compilation des sources des tests

Structure générale du POM

➤ **resources**

- ▶ Où se trouvent les ressources non compilables du projet (fichier de configuration, images, ...)

➤ **testResources**

- ▶ Où se trouvent les ressources non compilables spécifiques aux tests.

➤ **finalName**

- ▶ Le résultat d'un build maven est un fichier (dépendant du type de packaging choisi, vu dans un prochain chapitre) dont le nom sera celui précisé par cette balise.

□ Ex dans notre cas : maven-example-1

➤ **pluginManagement et plugin**

- ▶ Ces sections servent à lister et configurer les plugins maven participants au build du projet. Cette section sera abordée dans le chapitre sur le cycle de vie.

Propriétés - Variables

51

- Si l'on observe le POM effectif, on remarque que certaines valeurs (répertoire courant du projet par exemple) sont bien « insérées » dans des éléments prédéfinis par le Super POM
- Afin de permettre cette « injection » de valeur dans un POM (Super POM ou non !), maven dispose d'une mécanisme de propriétés dynamiques (similaire aux variables d'un langage de templating comme JSP/EL)
- Ces propriétés sont référencées dans les POM par \${ nom_de_la_propriété } et sont remplacées dynamiquement lors de la phase de « merge » des POM pour obtenir le POM effectif.

Propriétés - Variables

52

- Maven pré définit un certain nombre de propriétés par défaut. Par exemple
 - ▶ **project.basedir** : répertoire du projet où se trouve le fichier pom.xml
 - ▶ **project.build.directory** : répertoire où sont déposés les fichiers de travail du build
 - ▶ **project.version** : version du projet
 - ▶ ...
- Exemple d'utilisation : la valeur par défaut de <finalName> est définie dans le Super POM de la façon suivante
 - ▶ <finalName>\${project.artifactId}-\${project.version}</finalName>
- Pour la liste plus complète des propriétés accessibles :
 - ▶ <https://books.sonatype.com/mvnref-book/reference/resource-filtering-section-properties.html>

Propriétés - Variables

53

- Vous pouvez ajouter vos propres propriétés dans votre POM, à vous de choisir son nom sous la forme d'une balise XML

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <version.struts>2.5.2</version.struts>
    <version.xwork>2.3.30</version.xwork>
    <version.hibernate>5.2.16.Final</version.hibernate>
    <version.spring>4.3.16.RELEASE</version.spring>
    <b><version.spring.security>4.2.5.RELEASE</version.spring.security></b>
</properties>
```

- Et les utiliser où vous voulez via \${xxx} :

```
<dependencies>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-config</artifactId>
        <version>${version.spring.security}</version>
    </dependency>
    ....
</dependencies>
```

Structure générale du POM

- Le POM peut contenir d'autres éléments informatifs :
 - ▶ Liste des développeurs
 - ▶ Liste des contributeurs
 - ▶ Référence au gestionnaire de sources
 - ▶ Référence au gestionnaire d'anomalies
 - ▶ License
 - ▶ MailingList
 - ▶ ...
- https://maven.apache.org/pom.html#More_Project_Information

Dépendances d'un projet

55

- Il est extrêmement rare pour les projets de développement de ne pas dépendre de librairies externes
 - ▶ Librairie de logging, spring, hibernate ...
- Historiquement, les dépendances étaient :
 - ▶ téléchargées à la main
 - ▶ déposées dans un répertoire de la machine de chaque développeur
 - ▶ Ajoutées au classpath du projet dans l'éditeur de code
 - ▶ Poussées dans le SCM
- A cette gestion s'ajoutait la difficulté des dépendances des dépendances... (<=> dépendances transitives)
 - ▶ J'ai besoin de spring, mais spring a besoin de XXX qui a besoin de YYY ...

Dépendances d'un projet

- Maven simplifie radicalement le problème
- On ajoute dans le POM de notre projet la liste des dépendances directes de celui-ci, sous la forme :

```
<dependencies>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-config</artifactId>
        <version>4.2.5.RELEASE</version>
    </dependency>
    ....
</dependencies>
```

- On retrouve la nomenclature d'identification projet : groupId, artifactId et version

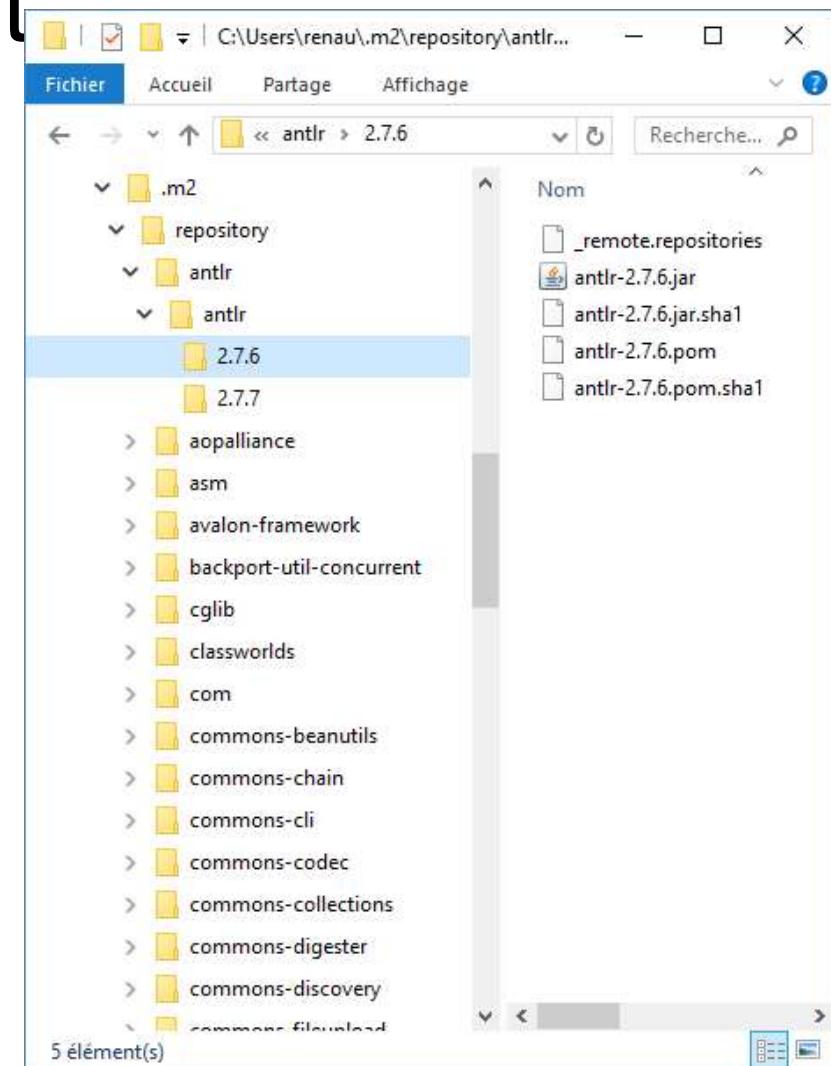
Dépendances d'un projet

57

- Lors d'une build, Maven va automatiquement récupérer les dépendances listées manquantes
 - ▶ Si elle n'ont pas déjà été récupérées précédemment
- Les dépendances des dépendances sont elles-aussi récupérées
 - ▶ On parle de dépendances transitives
 - ▶ Les dépendances étant souvent des projet « *mavenisés* » qui ont pris soin de définir dans leur propre pom la liste de leurs dépendances
- Maven interroge un ou plusieurs repository pour tenter de retrouver / télécharger les dépendances
 - ▶ Il vous indiquera si il rencontre un problème
 - ▶ Vous pouvez lui indiquer un repository local propre à votre organisation

Dépendances d'un projet

- Le repository local est d'abord examiné pour déterminer si la dépendance est présente sur la machine ou s'effectue le build
 - ➡ Un repository local n'est autre qu'un répertoire sur le disque dur
 - ➡ Par défaut, ce répertoire est situé dans \$HOME/.m2/repository
 - ➡ A l'intérieur de ce répertoire, on trouve, organisés dans une arborescence basée sur les groupId/artifactID, les fichiers binaires des dépendances (jar, war, ...)



Dépendances d'un projet

59

- Maven distingue dans la configuration les repositories distants dédiés aux dépendances de ceux dédiés aux plugins
- Une section du POM est réservée à la déclaration des repositories des plugins

```
<pluginRepository>
  <id>central</id>
  <name>Central Repository</name>
  <url>http://repo.maven.apache.org/maven2</url>
  <layout>default</layout>
  <snapshots>
    <enabled>false</enabled>
  </snapshots>
  <releases>
    <updatePolicy>never</updatePolicy>
  </releases>
</pluginRepository>
```

SCOPE

- Nos dépendances peuvent être variées et concerner différents aspects de notre produit
 - ▶ Dépendances pour les tests unitaires
 - ▶ Dépendances pour la compilation
 - ▶ Dépendances pour le déploiement
- Il est possible de préciser des scopes pour affiner la nature de la dépendance
 - ▶ Balise <scope></scope> dans une <dependency>
- Maven distingue **6 scopes** prédéfinis :
 - ▶ compile (*par défaut*)
 - ▶ provided
 - ▶ runtime
 - ▶ test
 - ▶ system
 - ▶ import

SCOPE

- Le scope **compile** est le scope par défaut
 - ➡ Les dépendances de ce scope sont accessible lors de tous les goals
 - ❑ De la compilation de toutes les sources (tests inclus),
 - ❑ De l'exécution des programmes (test inclus),
 - ❑ Dans tous les goals,
 - ❑ Elles sont transitives pour le projet
- Le scope **provided**
 - ➡ Indique que la dépendance est nécessaire à la compilation mais que l'environnement d'exécution du futur programme contient déjà la dépendance.
 - ❑ Par exemple, l'api Servlet de JEE ; nécessaire à la compilation des servlets, mais déjà présente sur le serveur ne nécessite pas d'être livrée avec l'application
 - ❑ Elles ne sont pas transitives pour le projet

SCOPE

➤ Le scope **runtime**

- ▶ La dépendance est nécessaire à l'exécution, mais pas à la compilation (driver jdbc par exemple).

➤ Le scope **test**

- ▶ La dépendance de ce scope n'est disponible qu'à la compilation et l'exécution des tests automatisés.
- ▶ Les librairie **n'est pas** packagée avec l'application.

SCOPE

63

➤ Le scope **system**

- ▶ Equivalent au scope **provided**, sauf qu'il désigne une librairie native spécifique à un environnement (dll windows par exemple), qui doit être présente sur la machine de build. Maven ne cherche pas la dépendance dans les repositories. Il faut indiquer le chemin local vers la librairie (un JAR contenant du Java et du code natif).
- ▶ Déprécié : utiliser provided

➤ Le scope **import**

- ▶ N'est utilisé que pour les dépendances entre POM dans une section `<dependencyManagement>`.
- ▶ Elle indique une liste de dépendances qui sera prioritaire à celle du POM parent
 - ❑ Utilisé par Spring, Log4J2, ... les framework qui ont beaucoup de JAR liés entre eux par un numéro de version

SCOPE

64

- Le choix du scope influe sur les phases de test/compilation/lancement

| | compile-classpath | test-classpath | runtime-classpath |
|----------|-------------------|----------------|-------------------|
| compile | ✓ | ✓ | ✓ |
| test | ✗ | ✓ | ✗ |
| runtime | ✗ | ✓ | ✓ |
| provided | ✓ | ✓ | ✗ |
| system | ✓ | ✓ | ✗ |
| import | No change | No change | No change |

<https://www.thecodejournal.tech/2020/08/maven-dependency-scopes/>

SCOPE

65

- Le choix du scope influe sur la transitivité.
 - ➡ Hormis le scope import
- Rappel : Une dépendance transitive est une dépendance d'une dépendance
 - ➡ Si le projet-a possède une dépendance dans le scope test vers le projet-b qui a une dépendance dans le scope compile vers le projet-c. Le projet-c sera donc une dépendance transitive du projet-a dans le scope test.

SCOPE

- Le tableau suivant indique l'impact en matière de transitivité
 - Au croisement des scopes est indiqué le scope pris en compte
 - Une case vide indique que la dépendance transitive est ignorée

| Le scope choisi par mon projet pour la dépendance A | Scope final transitif | | | |
|---|-----------------------|----------|-----------------|------|
| | compile | provided | runtime | test |
| compile | compile | - | runtime | - |
| provided | provided | - | provided | - |
| runtime | runtime | - | runtime | - |
| test | test | - | test | - |

Archétype

- C'est une organisation de projet déjà réalisée à l'avance pour vous
- Il n'est pas obligatoire d'en faire usage
 - ➡ cependant quand vous démarrez un projet sur une thématique donnée cela peut grandement vous aider
- Maven propose plusieurs archétypes, de bases (11)
 - ➡ Attention, ils sont tous dépassés (très ancien ...)

Archétype - Les 11

68

- maven-archetype-archetype : Projet simple
- maven-archetype-j2ee-simple : Projet JEE simple
- maven-archetype-mojo : Projet simple pour réaliser un plugin Maven.
- maven-archetype-plugin : Projet simple pour réaliser un plugin Maven.
- maven-archetype-plugin-site : Projet simple pour la génération d'un site de plugin
- maven-archetype-portlet : Projet simple pour la JSR-268 Portlet.
- maven-archetype-quickstart : projet simple.
- maven-archetype-simple : Projet simple
- maven-archetype-site : Projet simple pour la génération d'un site avec les standards APT, XDoc, et FML et internationnalisation.
- maven-archetype-site-simple : Projet simple pour un site.
- maven-archetype-webapp : projet simple pour une webapp

Archétype - Utilisation

69

- Pour lister TOUS les archétypes (3063) :

```
>mvn archetype:generate
```

- ▶ Vous pourrez indiquer le numéro que vous souhaitez utiliser
- ▶ Vous pourrez filtrer par mots clefs

- Une fois votre Archétype ciblé, il vous suffira de sélectionner son id
 - ▶ Un numéro qui dépend de votre filtre

Archétype - Utilisation

70

- Pour générer un projet à partir d'un Archétype directement avec son nom et sa version

```
>mvn archetype:generate  
      -DgroupId=votreGId  
      -DartifactId=votreAID  
      -DarchetypeGroupId=groupDeArchetype  
      -DarchetypeArtifactId=idDeArchetype  
      -DarchetypeVersion=versionDeArchetype  
      -DinteractiveMode=trueOuFalse
```

- ▶ **archetypeGroupId** : si absent considère que vous êtes dans les groupId de Maven (les 11)
- ▶ **archetypeVersion** : si absent, prend la dernière version disponible

Archétype - Résultat

71

- Après la commande, vous aurez un dossier qui porte le nom de l'ArtifactID
 - ➡ Dedans vous trouverez votre projet Maven
- Il faudra parfois chercher sur internet un archétype 'utilisable'
 - ➡ Les archétypes Maven par défaut sont en Java 5 sur JEE 5
 - ❑ Nous sommes en Java 18 et JEE 9 (jakarta EE 9)

Archétype - Exemple

72

- Nous allons réaliser un projet JEE 8 pour Java 11

```
>mvn archetype:generate  
      -DgroupId=com.entreprise  
      -DartifactId=banque.ee8  
      -DarchetypeGroupId=de.rieckpil.archetypes  
      -DarchetypeArtifactId=javaee8  
      -DarchetypeVersion=2.1.2  
      -DinteractiveMode=true
```

- Vous pouvez builder le projet et le déployer directement sous JBoss si vous le souhaitez

Conflits de versions

73

- Des conflits de versions peuvent survenir
 - ▶ Si l'on spécifie une dépendance directe vers A en version X mais qu'une dépendance transitive vers A nécessite une version Y
- Ces conflits de versions vont générer un ajout multiple d'une même dépendance **en plusieurs versions** différentes
 - ▶ Cela peut ne pas avoir d'effet (immédiat...) sur le projet
 - ▶ Mais tôt ou tard, des problèmes surviendront
- A vérifier avec `>mvn dependency:tree -Dverbose`

Conflits de versions

- Première solution :
 - ▶ Utiliser la balise `<exclusions>` dans la dépendance à l'origine du problème
 - ▶ Vous pouvez en indiquer plusieurs
 - ▶ On se servira des coordonnées de l'artefact
- Préférer la seconde

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>${version.spring.security}</version>
    <exclusions>
        <exclusion>
            <artifactId>spring-aop</artifactId>
            <groupId>org.springframework</groupId>
        </exclusion>
        <exclusion>
            <artifactId>spring-core</artifactId>
            <groupId>org.springframework</groupId>
        </exclusion>
        <exclusion>
            <artifactId>spring-context</artifactId>
            <groupId>org.springframework</groupId>
        </exclusion>
        <exclusion>
            <artifactId>spring-beans</artifactId>
            <groupId>org.springframework</groupId>
        </exclusion>
    </exclusions>
</dependency>
```

Conflits de versions

75

- Une seconde solution bien plus puissant est d'utiliser la balise `<dependencyManagement>`
 - ➡ Cette balise permet de préciser **la version qui doit être prise en compte pour une dépendance**
 - ❑ Cela fixe le numéro de version pour les dépendances transitives ET directes !
 - ❑ Vous pouvez aussi y indiquer un scope

```
<groupId>fr.tryade</groupId>
<artifactId>maven-example</artifactId>
<version>1</version>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aop</artifactId>
            <version>4.3.3</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Conflits de versions

- Par la suite, pour les dépendances directes, il devient inutile de préciser le numéro de version car il est fixé dans le *dependencyManagement*
- Et les dépendances transitives sont automatiquement récupérées dans la version managée !
- Conclusion : utilisez le **dependencyManagement** sur vos projets !

Dépendances entre vos projets

77

- Nous n'avons pour l'instant parlé que de projet simple « monolithique »
 - Le projet est autonome et n'a qu'un usage unique ; il n'est pas utile de le découper en plusieurs parties distinctes pouvant être réutilisées séparément
- Maven propose un mécanisme pour découper un projet en plusieurs modules potentiellement réutilisables
 - On peut réutiliser uniquement un module sans devoir « tirer » la totalité du projet

Dépendances entre vos projets

- Pour Maven, 1 POM = 1 Artéfact généré
- Si l'on souhaite packager un projet en plusieurs artefacts, il faut découper le projet en plusieurs modules
- Un module est essentiellement un sous répertoire
 - ➡ Par exemple, pour le projet ProjetXYZ
 - ❑ \ProjetXYZ
 - ❑ \Module1
 - ❑ \Module2
- Chaque module doit contenir un fichier pom.xml

Dépendances entre vos projets

79

- Chaque module est donc considéré comme un projet à part entière
 - ▶ On peut alors exécuter des commandes maven dans chacun des modules indépendamment
 - ▶ Chaque module doit avoir un artifactId différent : le groupId est en général le même puisqu'il s'agit du même projet
 - ▶ On doit déclarer des dépendances entre modules de la façon standard (<dependency>)
 - ❑ C'est obligatoire, les modules ne pouvant se 'voir' automatiquement entre eux, même au sein d'un même projet
- Il est cependant fastidieux de devoir construire chaque module indépendamment, s'il s'agit du même projet

Dépendances entre vos projets

80

- Pour résoudre ce problème, le pom du dossier racine du projet multi-module peut être configuré de façon à jouer le rôle de pom agrégateur
 - ▶ Son packaging sera ‘pom’
 - <packaging>pom</packaging>
 - ▶ Il faut ajouter la balises <modules> et lister les modules que l'on souhaite construire en même temps que le pom agrégateur avec la sous balise <module>

```
<modules>
    <module>MonProjetUn</module>
    <module>MonProjetDeux</module>
</modules>
```
 - ▶ Le nom du module correspond **au nom du répertoire** du sous-module, et non son artifactId
 - Une bonne convention est cependant d'adopter artifactId = nom du répertoire du module
 - Module non listé = module non construit

Dépendances entre vos projets

81

➤ Relations entre projets – projet multi modules

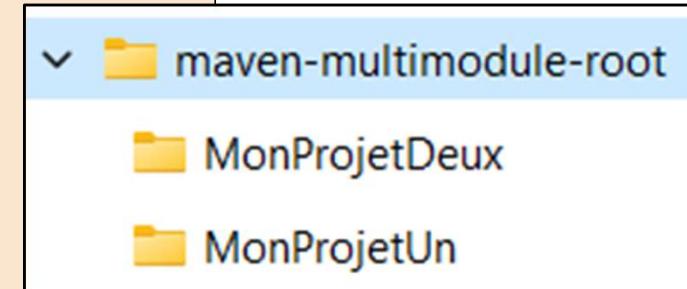
```
<?xml version="1.0" encoding="UTF-8"?>
<project>

    <modelVersion>4.0.0</modelVersion>

    <groupId>fr.training</groupId>
    <artifactId>maven-multimodule-root</artifactId>
    <version>2.2.0</version>
    <packaging>pom</packaging>

    <modules>
        <module>MonProjetUn</module>
        <module>MonProjetDeux</module>
    </modules>

</project>
```



Dépendances entre vos projets

82

- Il est tout à fait possible de constituer des modules / sous-modules de façon récursive
 - ▶ Certains projets complexes possèdent plusieurs niveaux de modules / sous-module avec des pom agrégateurs à différents étages
- La création de plusieurs pom pour les différents modules d'un projet génère en général beaucoup de copier-coller
 - ▶ Des dépendances communes à tous les modules...
 - ▶ Des configurations de plugins identiques...
- Maven offre un mécanisme simple pour régler ce problème : **l'héritage** de pom
- Souvenez-vous du pom effectif : celui-ci est la résultante de la fusion du Super POM et du POM projet
 - ▶ Il s'agit en réalité d'un héritage implicite !

Dépendances entre vos projets

- Pour réaliser un héritage entre POM, il faut utiliser la balise `<parent>` dans le POM enfant

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>fr.training</groupId>
        <artifactId>maven-multimodule-root</artifactId>
        <version>2.2.0</version>
    </parent>

    <artifactId>MonProjetDeux</artifactId>
    <!-- Tout ce qui concerne projet deux ... -->
</project>
```

Dépendances entre vos projets

- Le POM parent doit par défaut se trouver dans le répertoire parent du POM enfant
 - ➡ Il est possible de changer cet emplacement avec la balise `<relativePath>`
 - ➡ Il faut préciser le numéro de version du parent
- Il est alors possible de ne plus préciser le numéro de version ou le groupId dans le POM enfant
 - ➡ Ils sont hérités du parent.

Dépendances entre vos projets

- Le POM effectif du POM enfant est alors le résultat de la fusion du Super POM, du POM parent et du POM enfant
- L'utilisation de l'héritage est très utile pour factoriser les éléments communs
 - ▶ Dépendances communes
 - ▶ DependencyManagement commun dans le parent permettant de contrôler l'ensemble des versions des modules enfants et de résoudre les conflits
 - ▶ Plugins et configurations des plugins

Dépendances entre vos projets

- Certaines organisations / DSI ont élaboré un POM d'entreprise, duquel tous les projets héritent
 - ▶ votre propre POM parent doit hériter de ce POM d'entreprise
- Cette pratique vise à standardiser un certain nombre d'éléments au sein d'une organisation
 - ▶ Librairies, frameworks autorisés, versions, scope ...
 - ▶ Profils de build standardisés (vu dans un prochain chapitre)
 - ▶ Simplification de l'intégration des projets dans l'usine logicielle
 - Plugins et leurs configuration
 - ▶ ...

Bonnes pratiques

- Indiquez les versions dans des propriétés
 - ▶ Balise <properties></properties>
- Gérer les versions avec le mécanisme de dependency management
 - ▶ Pas de numéro de version directement dans une dependency !
 - ▶ Pensez à vérifier les dépendances transitives et à vous assurer régulièrement au fur et à mesure de l'ajout de librairies sur les projets qu'aucun conflit de version n'apparaît
- Grouper les dépendances par responsabilités et les identifier avec des commentaires
 - ▶ Librairies de log
 - ▶ Librairies de persistance
 - ▶ Librairies destinées aux tests...

Bonnes pratiques

88

- Le POM est un fichier XML, vous avez le droit d'y mettre des commentaires
 - ▶ <!-- Un commentaire XML -->
- Conservez une arborescence de répertoire respectant la hiérarchie de l'héritage
 - ▶ Il est possible de ne pas le faire mais cela amène de la confusion
- ArtifactId d'un module = nom du répertoire
- ArtifactId du POM parent = « nom du projet »-parent
- Ne changez pas les valeurs par défaut des emplacements de répertoire srcDirectory, testDirectory...
 - ▶ Convention over configuration ! Même si l'on peut changer la configuration....

Cycle de vie du build

89

- Nous avons jusqu'à présent parlé des projets, de leurs dépendances, de leurs organisations en modules et de leur relations inter-modules
 - ▶ C'est le côté « statique » du modèle
- Intéressons nous maintenant aux aspects dynamiques
 - ▶ Que se passe t'il lors d'un build ?
 - ▶ Que signifie « build » d'ailleurs?
 - ▶ Quelle partie de Maven effectue les tâches du « build » ?
 - ▶ Comment adapter une build ?

Cycle de vie du build

- Une build est appellée lifecycle
 - ▶ Maven définit 3 lifecycles standards
 - clean
 - default (appelé aussi par abus de langage « build »)
 - site
- Un lifecycle est un ensemble de phases
- A une phase sont rattachés un ensemble de goals
 - ▶ Le rattachement des goals aux phases des lifecycles s'effectue dans les pom, via les balises <plugin> et <execution>
- Les goals sont portés par les plugins maven
- lifecycle <=> Des Phases <=> des Goals portés par des plugins

Cycle de vie du build

- Dans la plupart des cas, déclencher « une build » équivaut à invoquer un lifecycle complet ou faire appel à une phase précise d'un lifecycle
 - ➡ Invocation de la phase install du cycle de vie par défaut

```
>mvn install
```

- ➡ Invocation de la phase clean du cycle de vie clean

```
>mvn clean
```

- ➡ Invocation du cycle de vie clean puis de la phase install du cycle de vie par défaut

```
>mvn clean install
```

Clean lifecycle

- Le cycle de vie clean est le plus simple
 - ▶ mvn clean
 - ▶ Il vide le répertoire de travail généré par maven lors de l'invocation des autres lifecycles
 - Le dossier target
- Il est constitué de 3 phases, et invoque les goals suivants
 - ▶ Pre-clean
 - *N'est liée à aucun goal par défaut*
 - ▶ Clean
 - Plugin ‘maven-clean-plugin’ (alias : clean), goal clean
 - ▶ Post-clean
 - *N'est liée à aucun goal par défaut*

Clean lifecycle

93

- Pre-clean et Post-clean sont des phases du cycle de vie qui existent pour vous permettre de rattacher des goals spécifiques à votre build
- La phase Clean est liée au plugin maven-clean-plugin, et plus précisément à son goal clean
- Ainsi, les commandes suivantes vont, par défaut, toutes aboutir au même résultat concret
 - ▶ mvn clean (lifecycle clean) => invoquera le plugin clean lors de la phase liée
 - ▶ mvn maven-clean-plugin:clean (appel direct au plugin sans passer par un lifecycle / une phase)
 - ▶ mvn clean:clean (appel direct au plugin en utilisant son alias)

Default lifecycle - En résumé

- **validate** : valide que le projet est correct et que toutes les informations nécessaires sont disponibles.
- **initialize** : initialiser l'état de construction, par exemple définir des propriétés ou créer des répertoires.
 - ▶ generate & process sources & resources
- **compile** : compile le code source du projet.
 - ▶ process-classes : post-traite les fichiers générés à partir de la compilation, par exemple pour faire une amélioration du bytecode sur les classes Java.
 - ▶ generate & process tests sources & resources
 - ▶ test-compile : compile le code source du test dans le répertoire de destination du test
 - ▶ process-test-classes : post-traite les fichiers générés à partir de la compilation de test, par exemple pour faire l'amélioration du bytecode sur les classes Java.
- **test** : exécuter des tests à l'aide d'un cadre de test unitaire approprié. Ces tests ne doivent pas nécessiter que le code soit empaqueté ou déployé.
 - ▶ prepare-package : effectue toutes les opérations nécessaires à la préparation d'un livrable avant le conditionnement proprement dit. Cela se traduit souvent par une version non compressée et traitée du package.
- **package** : prend le code compilé et le 'zip' dans son format distribuable, tel qu'un JAR.
 - ▶ pre-integration-test : effectue les actions requises avant l'exécution des tests d'intégration. Cela peut impliquer des choses telles que la configuration de l'environnement requis.
- **integration-test** : traite et déploie le package si nécessaire dans un environnement où les tests d'intégration peuvent être exécutés.
 - ▶ post-integration-test : effectue les actions requises après l'exécution des tests d'intégration. Cela peut inclure le nettoyage de l'environnement.
- **verify** : effectue des vérifications pour vérifier que le livrable est valide et répond aux critères de qualité (par exemple avec un SonarQube).
- **install** : installe le livrable dans le référentiel local, pour une utilisation en tant que dépendance avec d'autres projets localement.
- **deploy** : effectué dans un environnement d'intégration ou de publication, copie le livrable final dans le référentiel distant pour le partager avec d'autres développeurs et projets.

Default lifecycle - En détail

- Il est constitué des 23 phases suivantes :
 - ▶ **validate** : valide que le projet est correct et que toutes les informations nécessaires sont disponibles.
 - ▶ **initialize** : initialiser l'état de construction, par exemple définir des propriétés ou créer des répertoires.
 - ▶ **generate-sources** : génère n'importe quel code source à inclure dans la compilation.
 - ▶ **process-sources** : traite le code source, par exemple pour filtrer toutes les valeurs.
 - ▶ **generate-resources** : génère des ressources à inclure dans le package.
 - ▶ **process-resources** : copie et traite les ressources dans le répertoire de destination, prêtes pour le conditionnement.

Default lifecycle

- ▶ **compile** : compile le code source du projet.
- ▶ **process-classes** : post-traite les fichiers générés à partir de la compilation, par exemple pour faire une amélioration du bytecode sur les classes Java.
- ▶ **generate-test-sources** : génère n'importe quel code source de test à inclure dans la compilation.
- ▶ **process-test-sources** : traite le code source du test, par exemple pour filtrer toutes les valeurs.
- ▶ **generate-test-resources** : crée des ressources pour les tests.
- ▶ **process-test-resources** : copie et traite les ressources dans le répertoire de destination du test.
- ▶ **test-compile** : compile le code source du test dans le répertoire de destination du test
- ▶ **process-test-classes** : post-traite les fichiers générés à partir de la compilation de test, par exemple pour faire l'amélioration du bytecode sur les classes Java.
- ▶ **test** : exécuter des tests à l'aide d'un cadre de test unitaire approprié. Ces tests ne doivent pas nécessiter que le code soit empaqueté ou déployé.

Default lifecycle

97

- ▶ **prepare-package** : effectue toutes les opérations nécessaires à la préparation d'un livrable avant le conditionnement proprement dit. Cela se traduit souvent par une version non compressée et traitée du package.
- ▶ **package** : prend le code compilé et le 'zip' dans son format distribuable, tel qu'un JAR.
- ▶ **pre-integration-test** : effectue les actions requises avant l'exécution des tests d'intégration. Cela peut impliquer des choses telles que la configuration de l'environnement requis.
- ▶ **integration-test** : traite et déploie le package si nécessaire dans un environnement où les tests d'intégration peuvent être exécutés.
- ▶ **post-integration-test** : effectue les actions requises après l'exécution des tests d'intégration. Cela peut inclure le nettoyage de l'environnement.
- ▶ **verify** : effectue des vérifications pour vérifier que le livrable est valide et répond aux critères de qualité (par exemple avec un SonarQube).
- ▶ **install** : installe le livrable dans le **référentiel local**, pour une utilisation en tant que dépendance avec d'autres projets localement.
- ▶ **deploy** : effectué dans un environnement d'intégration ou de publication, copie le livrable final dans le référentiel **distant** pour le partager avec d'autres développeurs et projets.

Default lifecycle

- Contrairement au cycle de vie clean, aucun binding n'est pré-défini pour les phases du default lifecycle
- C'est le choix d'un type de packaging pour notre projet qui liera certains plugins aux phases présentées précédemment
 - ➡ Le type de packaging est abordé au chapitre suivant
 - ➡ Il permet d'indiquer si l'on veut un JAR ou WAR

Default lifecycle

- Nous ne sommes pas obligé de faire un
 - >mvn install
- Si vous voulez juste compiler, faites simplement
 - >mvn compile
- Notez tout de même que chaque phase dépend la précédente
 - ➡ Avant **test**, il faut **compiler**

Site lifecycle

- Enfin, le lifecycle site est dédié à la génération de documentation
 - ➡ Site internet « résumé » du projet, javadoc, métriques et rapport d'analyses statiques et dynamiques du code source...
- Il est constitué de 4 phases et des goals suivants
 - ➡ pre-site
 - ➡ site
 - plugin:goal = site:site
 - ➡ post-site
 - ➡ site-deploy
 - plugin:goal = site:deploy

Packaging

- Le choix du type de packaging va déterminer les plugins qui seront rattachés aux différentes phases du default lifecycle

- Nous avons rapidement abordé le principe de « packaging » d'un projet Maven avec le pom parent
 - ➡ Packaging : pom
 - ➡ Balise <packaging>

Packaging

- On distingue les types de packaging suivants
 - ▶ Jar : Java ARchive (application console ou librairie) (**valeur par défaut**)
 - ▶ Pom
 - ▶ War : Web ARchive (application web)
 - ▶ Ear : Enterprise ARchive (application avec EJB)
- On a aussi de manière plus anecdotique
 - ▶ Rar
 - ▶ Par
 - ▶ Maven-plugin

Packaging JAR

- La packaging Jar est le packaging par défaut lorsqu'il n'est pas précisé
- Ce packaging affecte les goals des plugins suivants aux phases :

| Phase | Plugin alias : goal |
|------------------------|-------------------------|
| process-resources | resources:resources |
| compile | compiler:compile |
| process-test-resources | resources:testResources |
| test-compile | compiler:testCompile |
| test | surefire:test |
| package | jar:jar |
| install | install:install |
| deploy | deploy:deploy |

Packaging JAR

- Il faut se référer à la documentation des plugins pour avoir une description précise de leur utilité
 - ▶ <https://maven.apache.org/plugins/maven-jar-plugin/>
- En quelques mots :
 - ▶ resources:resources
 - Copie des ressources du projet
 - ▶ compiler:compile
 - Compilation des .java en .class
 - ▶ resources:testResources
 - Copie des ressources de tests
 - ▶ compiler:testCompile
 - Compilation des .java de test en .class
 - ▶ surefire:test
 - Exécution des tests unitaires
 - ▶ jar:jar
 - ▶ Création du jar de l'application : .class et ressources
 - ▶ install:install
 - ▶ Installation du jar de l'application dans le repository maven local
 - ▶ deploy:deploy
 - ▶ Installation du jar de l'application dans un repository maven distant

Packaging POM

- Le packaging le plus simple : se contente de publier le pom.xml comme artefact produit !
- Ce packaging affecte les goals des plugins suivants aux phases :

| Phase | Plugin alias : goal |
|---------|------------------------|
| package | site:attach-descriptor |
| install | install:install |
| deploy | deploy:deploy |

Packaging WAR

- Le packaging war est le packaging dédié aux applications web
- Ce packaging affecte les goals des plugins suivants aux phases :

| Phase | Plugin alias : goal |
|------------------------|-------------------------|
| process-resources | resources:resources |
| compile | compiler:compile |
| process-test-resources | resources:testResources |
| test-compile | compiler:testCompile |
| test | surefire:test |
| package | war:war |
| install | install:install |
| deploy | deploy:deploy |

Packaging WAR

107

- On remarque de nombreuses similitudes avec le packaging jar
- Différence notable : la façon dont va être packagée l'application
 - ➡ Au lieu de produire un jar, le build va produire un war
 - ➡ Le war contiendra le code compilé, les sources des pages html, jsp, css, js... ainsi que les dépendances en scope compile et system (et les transitives du même scope, cf. tableau sur la gestion des scopes et des transitives)
 - ➡ Se référer à la documentation du maven-war-plugin pour une description détaillée
 - ❑ <https://maven.apache.org/plugins/maven-war-plugin/>

Gestion des ressources

- Le traitement des ressources projet correspond à la phase process-resources
- Le traitement des ressources implique
 - ➡ Copie des fichiers « /src/main/resources » dans « target/classes »
 - ➡ Lors de la copie, filtrage du contenu des fichiers et remplacement des tokens délimités par \${...} si le filtering est activé
 - Nous reviendrons sur ce mécanisme dans un chapitre dédié
- Attention à l'encoding des ressources !
 - ➡ UTF-8, ISO... le plugin resources doit être configuré pour utiliser le bon encoding (là aussi, nous y reviendrons)

Compilation

- La compilation va compiler le code source java en .class
 - ➡ <https://maven.apache.org/plugins/maven-compiler-plugin/>
- Vous pouvez, par exemple y préciser le niveau de compatibilité Java de votre code

```
<plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
        <source>1.7</source>
        <target>1.7</target>
    </configuration>
</plugin>
```

Compilation

- Maven comprend les propriétés suivantes automatiquement :

```
<properties>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
</properties>
```

- Pour les versions Java > 1.8, vous pouvez indiquer 9, 10, 11 ...
(il n'y a plus de 1.x)
- Vous n'avez pas besoin de préciser <source> ou <target>
quand vous faites usage de ses propriétés

Ressources de tests et compilation

- Le traitement des ressources de tests et la compilation des sources de tests sont effectuées à l'identique
 - ▶ /src/main/resources devient src/test/resources
 - ▶ /src/main/java devient src/test/java
 - ▶ target/classes devient target/test-classes

Tester

112

- Maven a intégré très tôt le principe de tests automatisés en continu
- A chaque build, les tests compilés dans src/test/java sont joués
 - ▶ Bien sûr, il faut avoir écrit des tests...
- En cas d'échec d'un seul test, la build est interrompue
 - ▶ Build Failed
- Sémantiquement, on distingue plusieurs catégories de test. On peut en retenir deux :
 - ▶ Test unitaires (boîte blanche) : TU
 - ▶ Tests d'intégration (boîte noire) : TI
- Maven nous permet de distinguer les deux catégories en liant les TU et TI à des phases différentes et à des plugins différents

Tester

113

- Tests unitaires
 - ▶ Plugin : org.apache.maven.plugins:maven-surefire-plugin
 - ▶ Lié à la phase **test**
 - ▶ <https://maven.apache.org/surefire/maven-surefire-plugin/>
- Tests d'intégration
 - ▶ Plugin : org.apache.maven.plugins:maven-failsafe-plugin
 - ▶ Lié à la phase **integration-test**
 - ▶ <https://maven.apache.org/surefire/maven-failsafe-plugin/>
- Possibilité d'utiliser différents noyaux de tests automatisés (JUnit, testng,...)
- On doit distinguer au développement les TU des TI via une règle de nommage qui sera utilisée par les deux plugins (configurés en conséquence) pour savoir quels tests exécuter

Tester

114

- Pour dissocier vos tests d'intégration, vous pouvez les nommer xxxxIT.java
- Puis configurer le plugin surefire pour qu'il ignore ces classes

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <configuration>
        <excludes>
            <exclude>**/*IT.java</exclude>
        </excludes>
    </configuration>
</plugin>
```

Tester

115

➤ Par défaut, si vos tests unitaires échouent, la build sera un échec.

➤ Pour les tests d'intégration, il faudra ajouter au plugin failsafe le fait d'appeler la phase verify

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-failsafe-plugin</artifactId>
  <executions>
    <execution>
      <id>integration-tests</id>
      <goals>
        <!-- Lancement des tests d'integration -->
        <goal>integration-test</goal>
        <!-- Build en failure si erreur lors des tests d'integrations -->
        <goal>verify</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Installer

- La phase d'installation suit la phase de packaging
 - ▶ Rappel : la phase de packaging sert à produire un livrable unique à partir des sources du projet et dépend du type de packaging choisi (war, jar ...)
- Cette phase invoque le plugin maven-install-plugin qui va copier dans le **repository local** de la machine l'artefact produit
 - ▶ En suivant le layout de répertoire groupId / artifactId / version
 - ▶ <https://maven.apache.org/plugins/maven-install-plugin/>
- Installer l'artefact dans le repository local est indispensable pour « partager » ce dernier avec d'autres projets locaux

Déployer

117

- La phase de déploiement est équivalente à la phase d'installation,
mais sur le **repository distant**
 - ▶ Copie de l'artifact produit par la phase package
- Le plugin chargé du déploiement distant doit bien sûr être configuré
pour pouvoir effectuer le déploiement
 - ▶ Quelle est l'adresse du repository distant ?
- Le serveur doit bien sûr accepter le transfert de fichier
 - ▶ Webdav
 - ▶ Scp...

```
<distributionManagement>
    <repository>
        <id>MonMavenRepository</id>
        <url>dav:http://10.0.0.1/repository/</url>
    </repository>
</distributionManagement>
```

Déployer

- En fonction du mode de transfert choisi, la configuration dans le pom doit-être adaptée :
 - ▶ Ajouter les librairies nécessaires (webdav, scp...)
 - ▶ Configurer les credentials d'accès
- Se référer aux ressources disponibles sur internet, en fonction du choix fait par votre organisation
 - ▶ C'est une partie très variable et fluctuante d'une organisation à l'autre
 - ▶ <https://maven.apache.org/plugins/maven-deploy-plugin/>

Profils de build

- Il est souvent nécessaire d'adapter une build en fonction de l'environnement cible pour lequel on construit son application
 - Dev, recette, production, client x, client y...
- Maven propose une mécanique standardisée pour « adapter » le build projet et le décliner en plusieurs variantes : les profils
- Un profil porte un nom (id) et est une sorte de « mini pom » dans le pom
 - On peut y définir / redéfinir de nombreux éléments du pom

Profils de build

➤ Ex de profil dans le 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/maven-v4_0_0.xsd">

    <build> ... </build>
    <profiles>
        <profile>
            <id>production</id>
            <build>
                <plugins>
                    <plugin>
                        <groupId>org.apache.maven.plugins</groupId>
                        <artifactId>maven-compiler-plugin</artifactId>
                        <configuration>
                            <debug>false</debug>
                            <optimize>true</optimize>
                        </configuration>
                    </plugin>
                </plugins>
            </build>
        </profile>
    </profiles>
```

Profils de build

121

- Une définition de profil peut contenir les éléments suivants :

```
<profile>
    <id></id>
    <activation></activation>
    <build>
        <defaultGoal>...</defaultGoal>
        <finalName>...</finalName>
        <resources>...</resources>
        <testResources>...</testResources>
        <plugins>...</plugins>
    </build>
    <reporting>...</reporting>
    <modules>...</modules>
    <dependencies>...</dependencies>
    <dependencyManagement>...</dependencyManagement>
    <distributionManagement>...</distributionManagement>
    <repositories>...</repositories>
    <pluginRepositories>...</pluginRepositories>
    <properties>...</properties>
</profile>
```

Profils de build

122

- Si l'on observe les éléments que l'on peut faire figurer dans le profil, on voit que
 - ▶ On peut modifier les plugins invoqués, en invoquer d'autres supplémentaire
 - ▶ Ajouter des dépendances, changer les versions des dépendances
 - ▶ Filtrer des ressources différentes
 - ▶ Activer des sous-modules spécifiques
 - ▶ ... etc
- On pourrait presque définir l'ensemble du pom dans un profil...
- Mais il faut tout de même se limiter au maximum pour ne représenter QUE les éléments spécifiques pour chaque profil
 - ▶ L'absence de profil revenant à un profil par défaut qui est celui du pom lui-même.
 - ▶ Les profils ne devraient contenir que des deltas mineurs

Profils de build

- On peut créer autant de profils que l'on souhaite
 - ➡ Chaque profil devant avoir un id unique
- Utilisation classique
 - ➡ Profil « dev », « recette », « production », adaptant le build à la plateforme ciblée lors du build
- Pour être utilisé, un profil doit être activé
 - ➡ Par défaut
 - ➡ ou via ligne de commande
 - ➡ ou via une condition

Profils de build

- Les profils sont hérités
- Il est possible d'identifier un/des profils « actif par défaut » dans la configuration d'activation
 - ➡ <activation> <activeByDefault>true</activeByDefault> </activation>
 - ➡ Il(s) sera/seront alors actif(s) sans avoir à le spécifier en ligne de commande
- Il peut être utile de connaître la liste des profils actifs

```
>mvn help:active-profiles
```

Profils Activation Manuelle

125

- Pour activer un profil, on peut le faire explicitement lors de l'invocation de Maven en utilisant le nom du profil avec le flag -P

```
>mvn clean install -P acceptanceTesting
```

- Le profil dont l'id correspond sera activé et le pom effectif prendra en compte ses spécificités
- On peut préciser plusieurs profils à la fois en séparant les ids par des virgules

```
>mvn clean install -P dev,postgresDb,noLdap
```

Profils Activation automatique

- On peut aussi définir dans la déclaration du profil des conditions d'activation automatique
 - ➡ Balise <activation>
- Plusieurs conditions d'activation automatique sont gérées
 - ➡ <os> L'activation se base sur l'os de la machine de build
 - ➡ <jdk> L'activation se base sur la version du jdk utilisée pour le build
 - ➡ <file> Présence / absence d'un certain fichier
 - ➡ <property> Présence (ou absence via le '!') d'une propriété ou d'une valeur de propriété d'environnement au lancement du build, spécifiée avec -D

Profils Activation automatique

127

➤ Activation par propriété, exemples

```
<profile>
  <id>dev</id>
  <activation>
    <jdk>1.5</jdk>
    <os>
      <name>Windows XP</name>
      <family>Windows</family>
      <arch>x86</arch>
      <version>5.1.2600</version>
    </os>
    <property>
      <name>customProperty</name>
      <value>BLUE</value>
    </property>
    <file>
      <exists>file2.properties</exists>
      <missing>file1.properties</missing>
    </file>
  </activation>
</profile>
```

Profils

128

- Il est aussi possible de définir des profiles au niveau
 - ▶ du fichier settings.xml
 - ❑ celui dans votre dossier .m2
 - ❑ celui dans le dossier conf de l'installation de Maven
 - ❑ <https://maven.apache.org/ref/current/maven-settings/settings.html>
 - ▶ dans un fichier dédié : profiles.xml
 - ❑ N'est plus supporté en Maven 3+

Profils

129

- Si vous demander un profil qui n'existe pas, Maven ne se lancera pas
- Vous pouvez désactiver un profil en le préfixant par un '!' (ou un '-')

```
>mvn clean install -P dev,postgresDb,!noLdap
```

Filtre

130

- Lors de la phase process-resources les fichiers situés dans src/main/resources étaient copiés et éventuellement filtrés des tokens délimités par \${nomDeLaPropriete}
- Afin que le filtrage s'effectue, il faut l'activer dans le pom du projet
 - ➡ Il n'est pas activé par défaut pour éviter les effets de bords non voulus

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

Filtre

131

- Il est possible d'ajouter des répertoires de ressources au build en plus de celui par défaut

```
<build>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
    </resource>
    <resource>
      <directory>src/main/xml</directory>
    </resource>
    <resource>
      <directory>src/main/images</directory>
    </resource>
  </resources>
</build>
```

Filtre

132

- Il est possible d'inclure / exclure certains types de fichiers par leur extension

```
<build>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <includes>
        <include>*.properties</include>
      </includes>
      <excludes>
        <exclude>*.bat</exclude>
        <exclude>*.sh</exclude>
      </excludes>
    </resource>
  </resources>
</build>
```

Propriétés

133

- Toutes les propriétés suivantes sont utilisables / filtrables
- Les Propriétés Projet
 - ▶ Commencent par project.*
 - ▶ Permettent de référencer n'importe quelle valeur du POM
 - project.groupId, project.build.finalName etc... (respecter la hiérarchie des éléments du pom)
 - L'ensemble des propriétés accessibles sont disponibles dans la documentation de maven
 - <https://maven.apache.org/ref/current/maven-model/maven.html>
- Les Propriétés Settings
 - ▶ Commencent par settings.*
 - ▶ Permettent de référencer n'importe quelle valeur du fichier de settings maven
 - Sur le même principe que le pom, les éléments de ce fichier sont accessibles en suivant l'imbrication des balises
 - L'ensemble des propriétés accessibles sont disponibles dans la documentation de maven
 - <https://maven.apache.org/ref/current/maven-settings/settings.html>

Propriétés

134

- Toutes les propriétés suivantes sont utilisables / filtrables (suite)
- Les Propriétés d'environnement
 - ▶ Commencent par env.*
 - ▶ Permettent de référencer n'importe quelle variable d'environnement définie par la machine exécutant le build
 - env.HOME
 - env.MVN_HOME
 - ...
- Les Propriétés système
 - ▶ Sont les propriétés exposées par la classe java.lang.System
 - ▶ Toute variable retournée par System.getProperty() est accessible
 - ▶ Quelques exemples
 - java.version
 - user.name
 - ...

Propriétés

135

- Toutes les propriétés suivantes sont utilisables / filtrables (suite)
- Les Propriétés utilisateur
 - ▶ Sont des propriétés définies dans le pom via la balise <properties>
 - Rappel : Les profils peuvent aussi déclarer des properties
 - ▶ Chaque sous-balise de <properties> devient une propriété accessible
 - Balise <foo> => à référencer avec \${foo}

```
<build>
  <filters>
    <filter>src/main/filters/test-config.properties</filter>
  </filters>
</build>
```

- Les Propriétés définies dans les filters
 - ▶ Sont des propriétés définies dans un fichier properties, lui-même déclaré dans une balise filter

Options de lancement

- Quelques options activable en ligne de commande peuvent être très utiles
 - ▶ ‘mvndebug’ au lieu de ‘mvn’ : permet de lancer maven en mode ‘remote debugging’ java. L’exécution se bloque et attend qu’un debugger se connecte. Il suffit ensuite de connecter son ide sur la remote jvm pour débogguer l’application lancée via maven.
 - ▶ -h : liste l’ensemble des propriétés maven activable en ligne de commande
 - ▶ -D : permet de définir une propriété système à la volée
 - Exemple : mvn –DmyProp=titi

Options de lancement

137

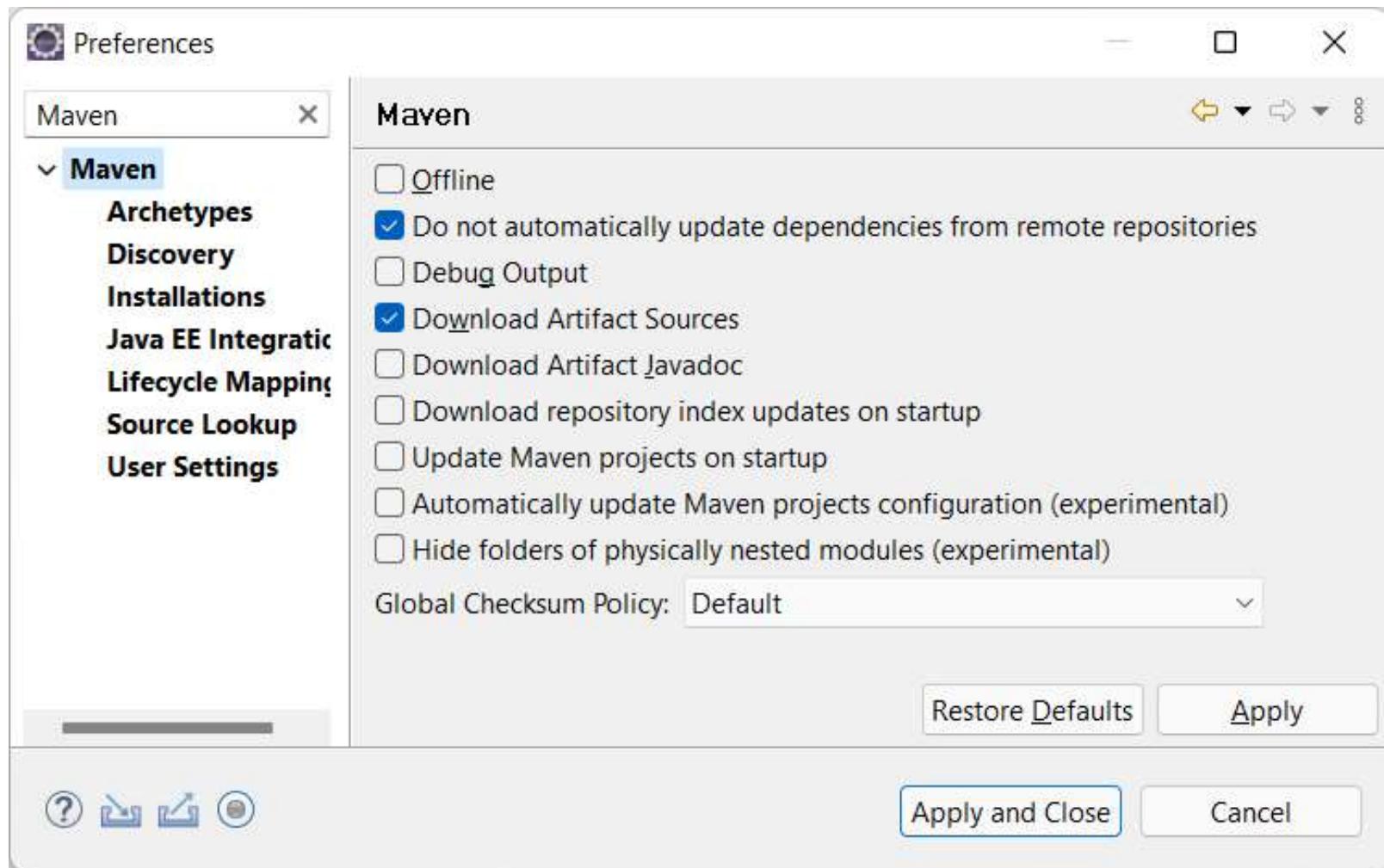
- ▶ -l : spécifie un fichier de log ou écrire la sortie console du build
- ▶ -o : (utile en cas de coupure réseau) lance maven en mode déconnecté ; maven n'interrogera aucun repository distant.
- ▶ -q : sortie console minimalist
- ▶ -X : sortie console en mode debug ; très utile pour diagnostiquer divers problèmes liés au build
- ▶ -gs : permet de pointer sur un fichier de settings.xml spécifique
- ▶ -f : permet de lancer un build en pointant sur un fichier pom.xml particulier

Eclipse et Maven

- Maven est intégré à Eclipse
- Il n'y a rien à installer en plus
 - ➡ Vous n'avez même pas à installer Maven sur votre machine, il est 'dans' Eclipse
- Vous pouvez le configurer dans Eclipse
 - ➡ Changer les propriétés par défaut
 - ➡ Cibler un Maven externe à Eclipse si la version embarquée ne vous convient pas

Eclipse et Maven

139



Eclipse et Maven

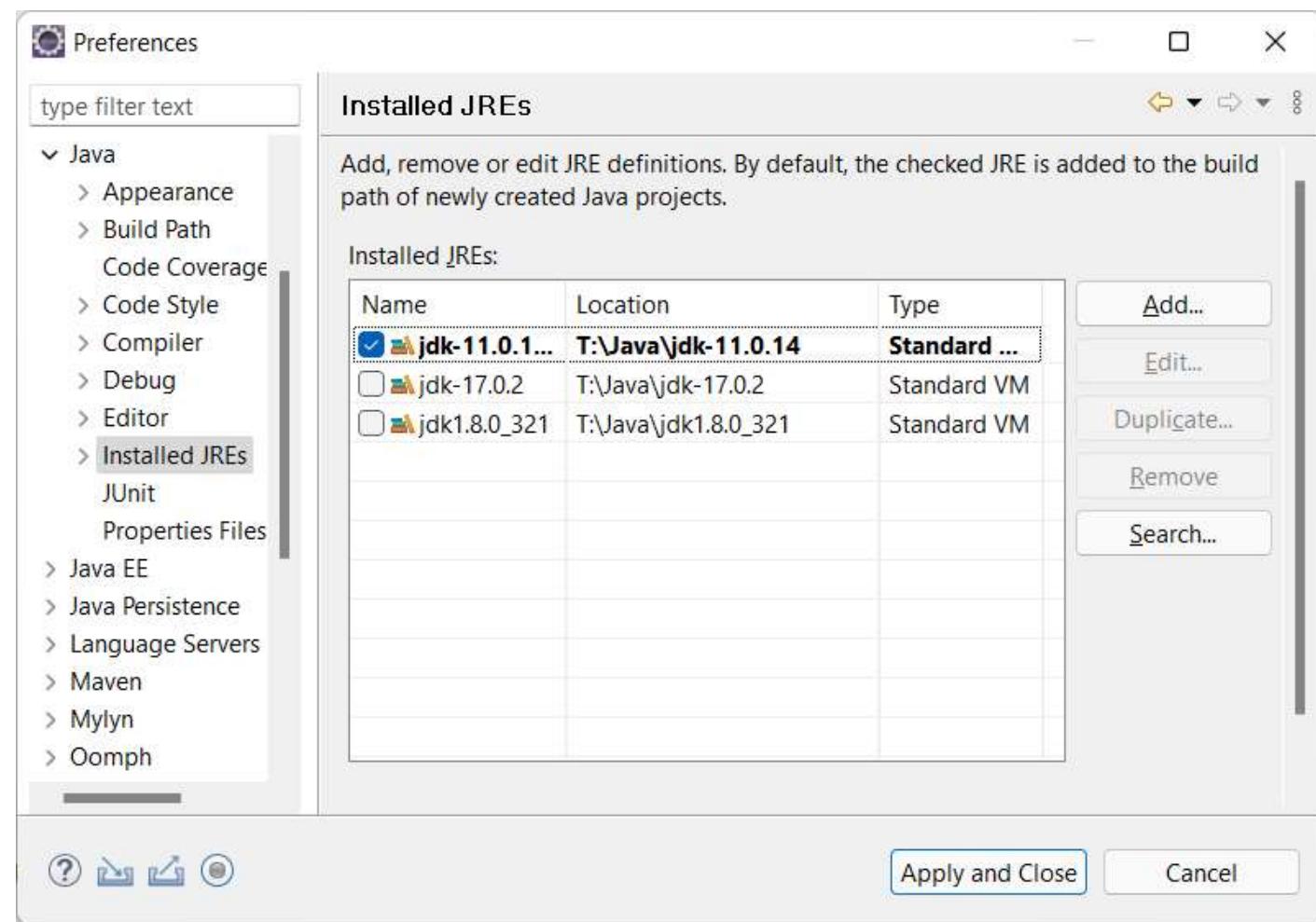
- Attention : par défaut, sous Eclipse, il n'y a pas de checksum
 - ➡ Potentiellement vous allez télécharger des dépendances incorrectes sans le savoir <=> votre projet marchera 'mal'
- Il est conseillé d'activer le checksum afin d'éviter des erreurs de téléchargement (sur des réseaux lents ou limités)



Eclipse et Maven - JDK for Ever

141

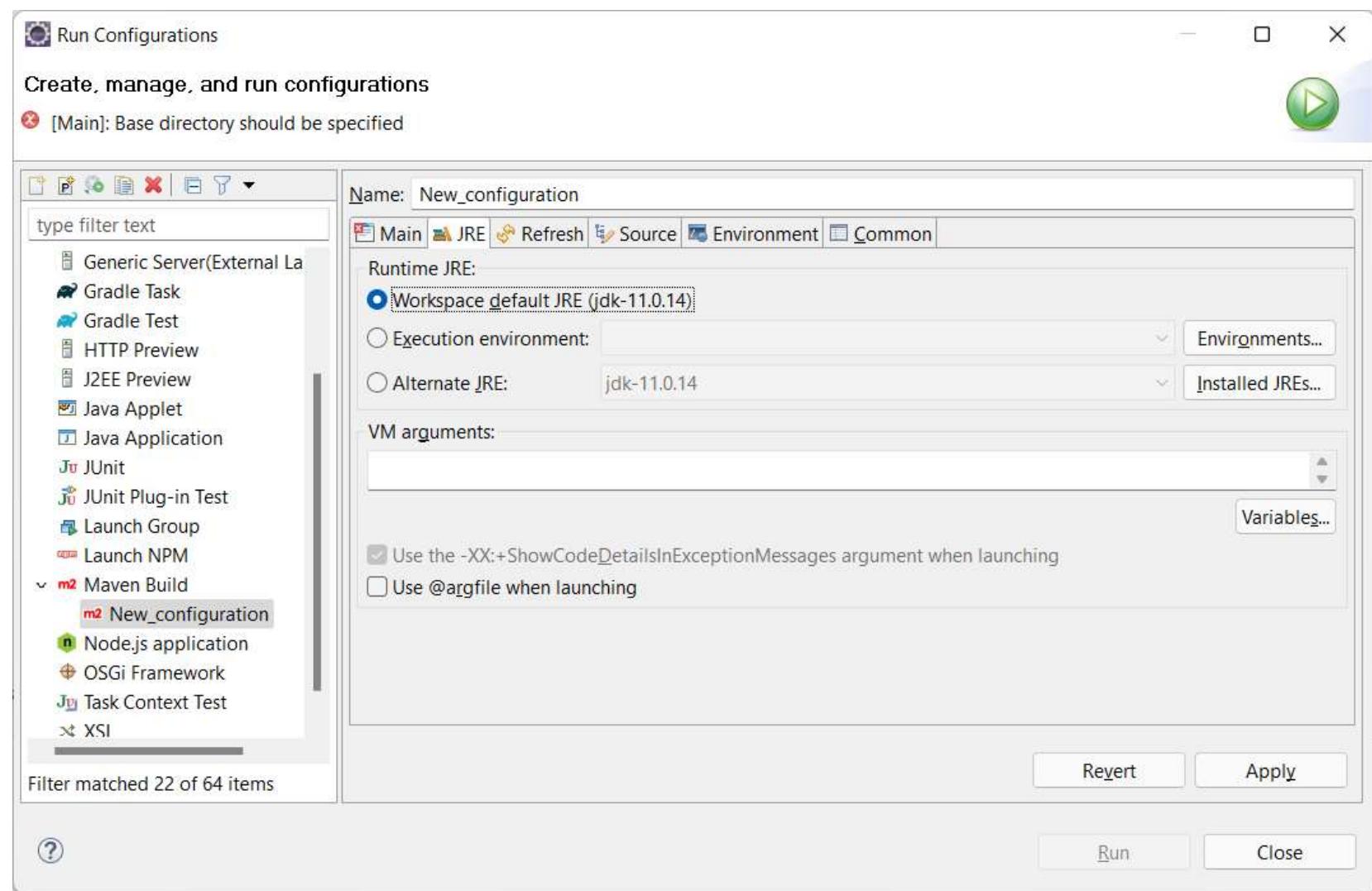
- Même si cela peut vous sembler surprenant, sous Eclipse vous devrez toujours faire usage d'un JDK pour lancer Maven
- Vérifiez bien que vous pointer vers un JDK (et surtout PAS vers un JRE)



Eclipse et Maven - JDK for Ever

142

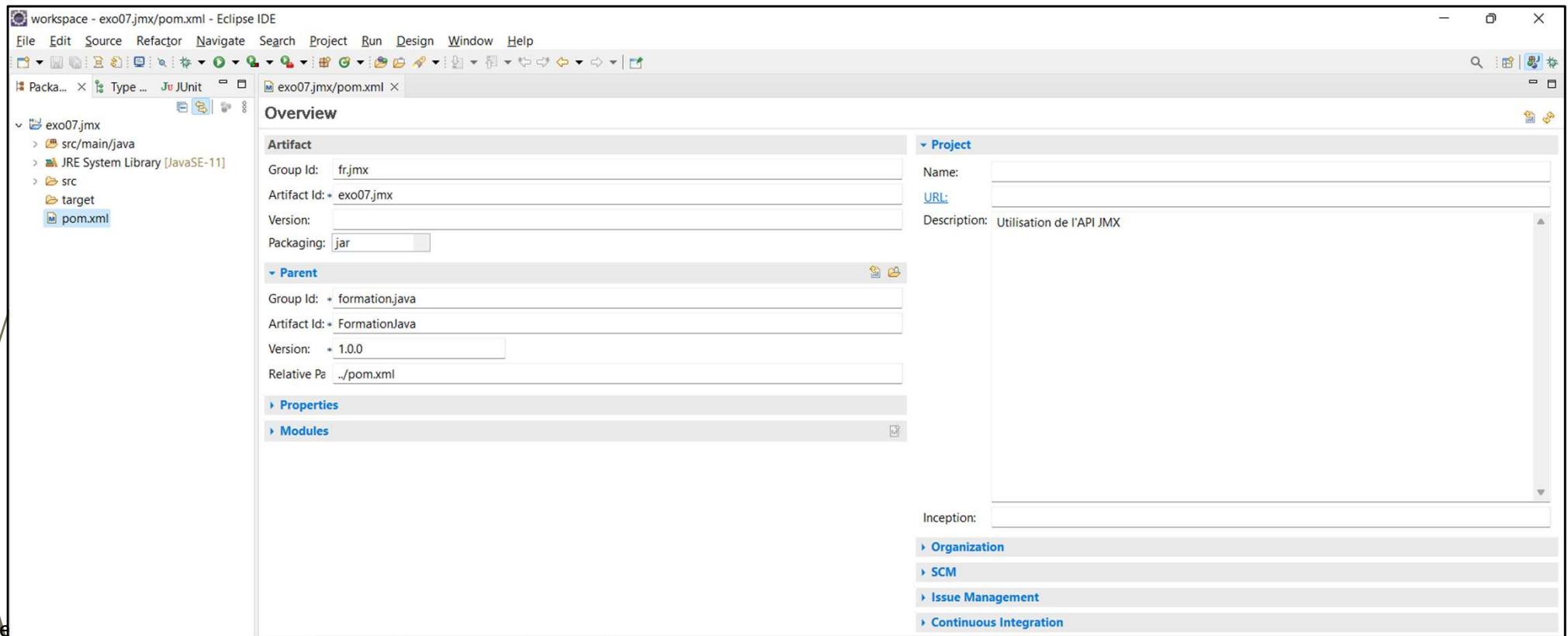
- Vous pouvez aussi gérer cette contrainte de manière plus localisée via le gestionnaire de configuration de lancement Eclipse



Eclipse

143

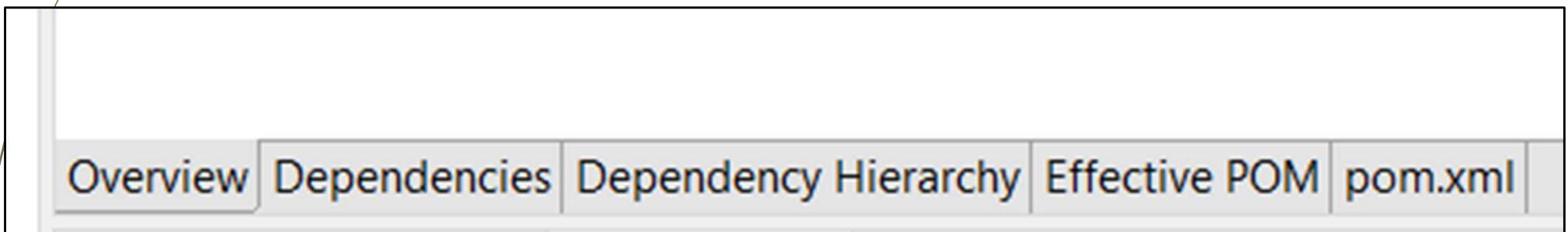
- Vous avez un éditeur graphique pour manipuler votre fichier pom.xml



Eclipse

144

- Notez aussi les onglets, en bas, de cet éditeur qui donnent un accès très visuelle à
 - ▶ Vos dépendances (transitives ou pas)
 - ▶ Vos hiérarchies de POM



Eclipse

➤ Exemple de Dependency Hierarchy

stone.lunchtime/pom.xml ×

Dependency Hierarchy [test]

Dependency Hierarchy

- org.jacoco.agent : 0.8.7 - runtime [test]
- spring-boot-starter-log4j2 : 2.6.4 [compile]
 - log4j-slf4j-impl : 2.17.1 [compile]
 - slf4j-api : 1.7.36 (managed from 1.7.25) [compile]
 - log4j-api : 2.17.1 [compile]
 - log4j-core : 2.17.1 [runtime]
 - log4j-core : 2.17.1 [compile]
 - log4j-api : 2.17.1 [compile]
 - log4j-jul : 2.17.1 [compile]
 - log4j-api : 2.17.1 [compile]
 - jul-to-slf4j : 1.7.36 [compile]
 - slf4j-api : 1.7.36 [compile]
- spring-boot-starter-mail : 2.6.4 [compile]
 - spring-boot-starter : 2.6.4 [compile]
 - spring-boot : 2.6.4 [compile]
 - spring-boot-autoconfigure : 2.6.4 [compile]
 - spring-boot : 2.6.4 [compile]
 - jakarta.annotation-api : 1.3.5 [compile]
 - spring-core : 5.3.16 [compile]
 - snakeyaml : 1.29 [compile]
 - spring-context-support : 5.3.16 [compile]
 - spring-beans : 5.3.16 [compile]
 - spring-context : 5.3.16 [compile]
 - spring-core : 5.3.16 [compile]
 - jakarta.mail : 1.6.7 [compile]

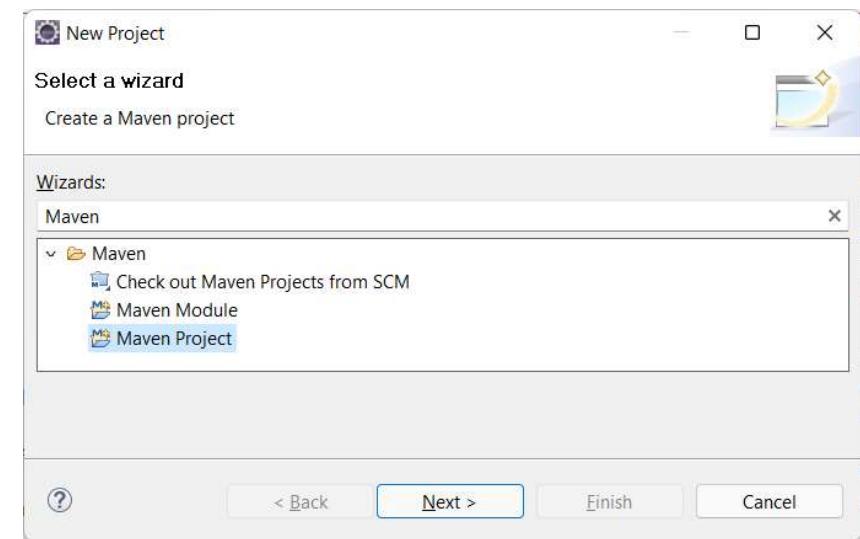
Resolved Dependencies

- accessors-smart : 2.4.8 [compile]
- android-json : 0.0.20131108.vaadin1 [runtime]
- antlr : 2.7.7 [compile]
- apiguardian-api : 1.1.2 [test]
- asm : 9.1 [compile]
- aspectjweaver : 1.9.7 [compile]
- assertj-core : 3.21.0 [test]
- bcpkix-jdk15on : 1.64 [compile]
- bcprov-jdk15on : 1.64 [compile]
- byte-buddy : 1.11.22 [compile]
- byte-buddy-agent : 1.11.22 [test]
- checker-qual : 3.5.0 [runtime]
- classgraph : 4.8.138 [compile]
- classmate : 1.5.1 [compile]
- commons-codec : 1.15 [compile]
- commons-lang3 : 3.12.0 [compile]
- evo-inflector : 1.3 [compile]
- h2 : 1.4.200 [runtime]
- hamcrest : 2.2 [test]
- hibernate-commons-annotations : 5.1.2.Final [compile]
- hibernate-core : 5.6.5.Final [compile]
- HikariCP : 4.0.3 [compile]
- istack-commons-runtime : 3.0.12 [compile]
- jackson-annotations : 2.13.1 [compile]

Overview | Dependencies | Dependency Hierarchy | Effective POM | pom.xml |

Eclipse - Créer un projet

- Pour créer un projet Maven
 - ▶ File / New / Project ...
 - ▶ Filtrez, en tapant Maven
 - ▶ Sélectionnez Maven Project
 - ▶ Suivez ensuite le Wizard



- Il est fortement conseillé de NE PAS faire usage des Archetype Eclipse

Create a simple project (skip archetype selection)

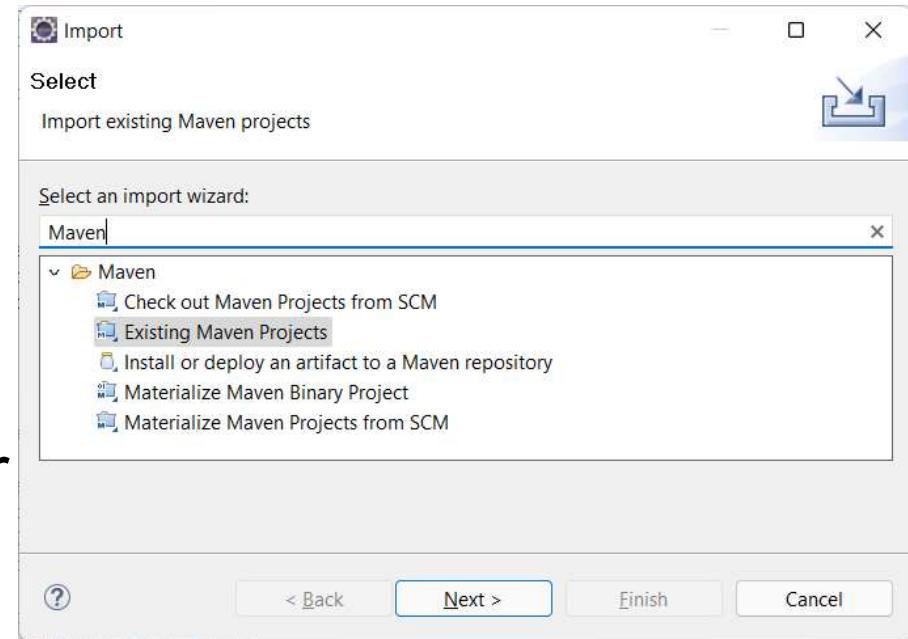
Eclipse - Créer un projet

- Notez que par défaut votre nouveau projet sera en Java 5
- Pensez à ré-ajuster cela en utilisant les propriétés Maven (<properties>)
 - ➡ Et SURTOUT pas les propriétés Eclipse du projet

Eclipse - Importer un projet

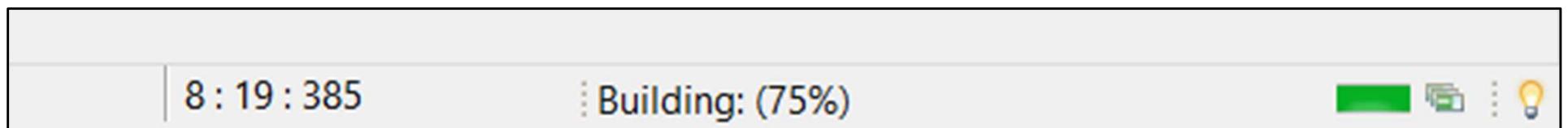
148

- Pour importer un projet Maven
 - ▶ File / Import
 - ▶ Filtrez, en tapant Maven
 - ▶ Sélectionnez Existing Maven Project
 - ▶ Indiquer le dossier contenant le fichier pom.xml
- Important : le projet importé RESTE là où il se trouve (il n'est PAS placé dans le dossier workspace)



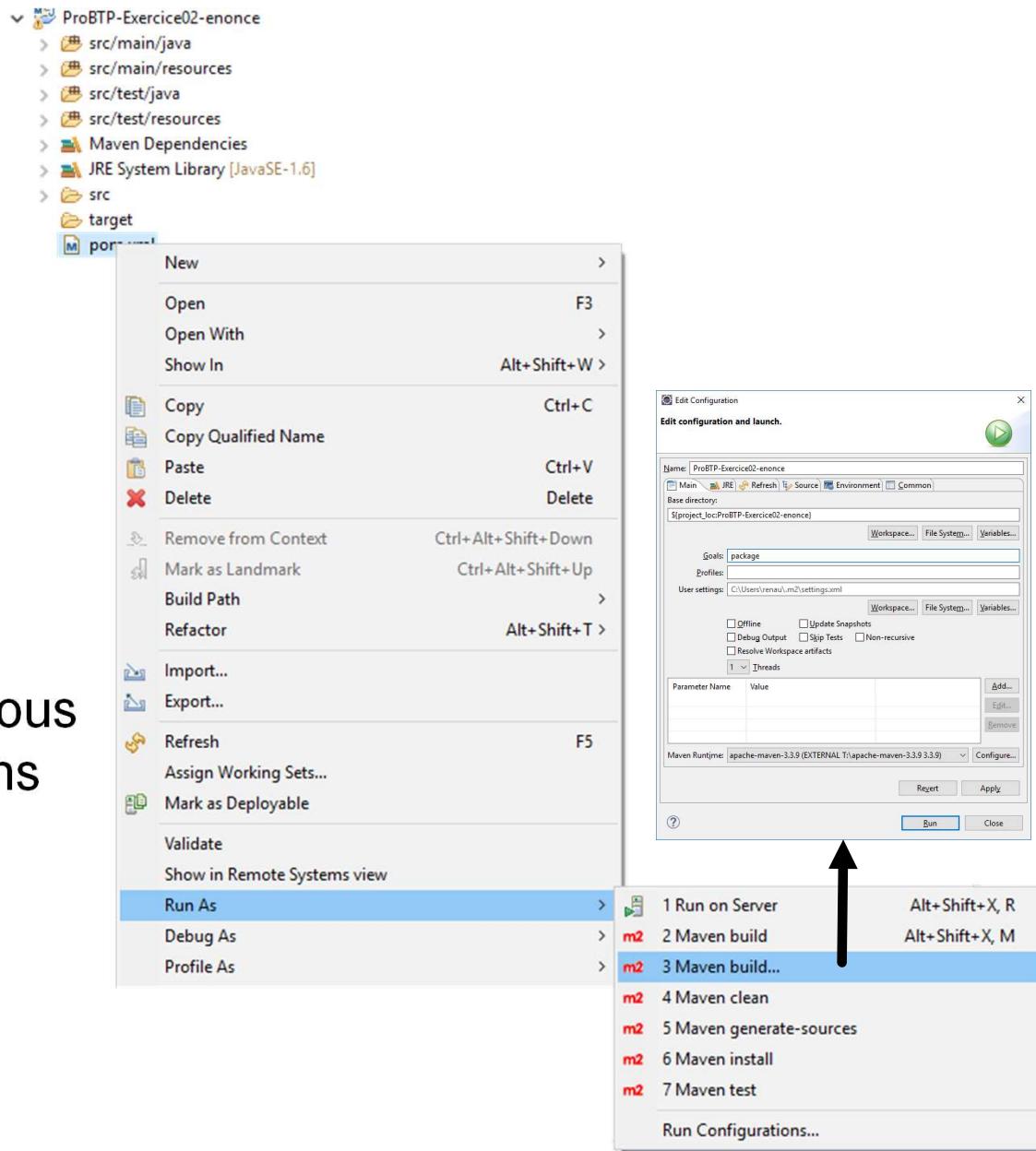
Eclipse - Importer / Créer

- Maven gère les dépendances en allant les télécharger d'Internet
- C'est la même chose dans votre IDE
- ATTENDEZ que toutes les dépendances minimales soient téléchargées



Eclipse

- Pour lancer un goal Maven
 - ➡ Clique droit sur votre fichier pom.xml
 - ➡ Puis Run As (ou Debug As)
 - ➡ Maven xxxx
 - ☐ Build : Indiquez le/s Goal que vous désirez ainsi que d'autres options
 - ☐ clean : Lance le goal clean
 - ☐ install : Lance le goal install
 - ☐ test : Lance le goal test

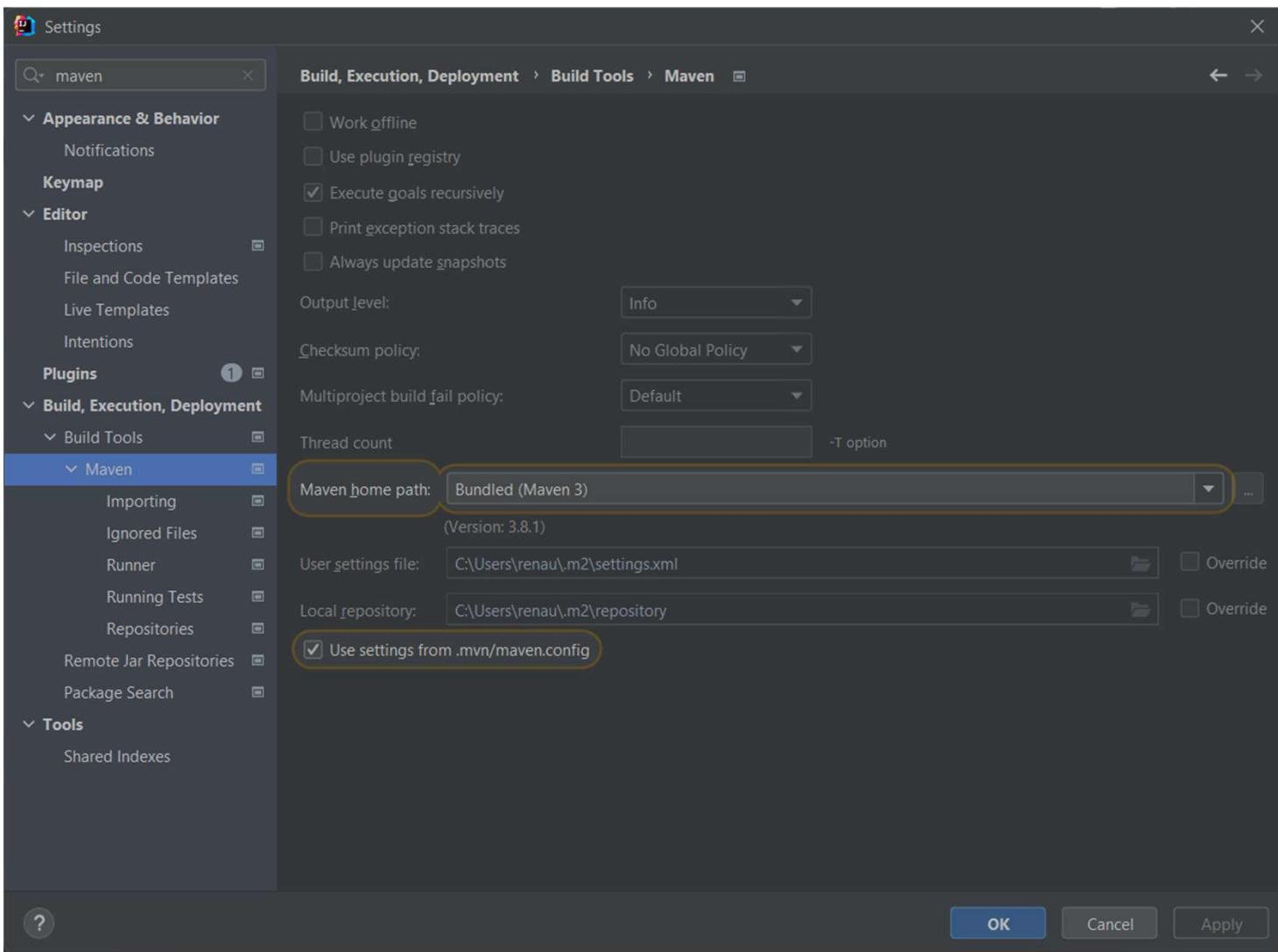


IntelliJ et Maven

- Maven est intégré à IntelliJ
- Il n'y a rien à installer en plus
 - ➡ Vous n'avez même pas à installer Maven sur votre machine, il est 'dans' IntelliJ
- Vous pouvez le configurer dans IntelliJ
 - ➡ Changer les propriétés par défaut
 - ➡ Cibler un Maven externe à IntelliJ si la version embarquée ne vous convient pas

IntelliJ et Maven

152



IntelliJ et Maven

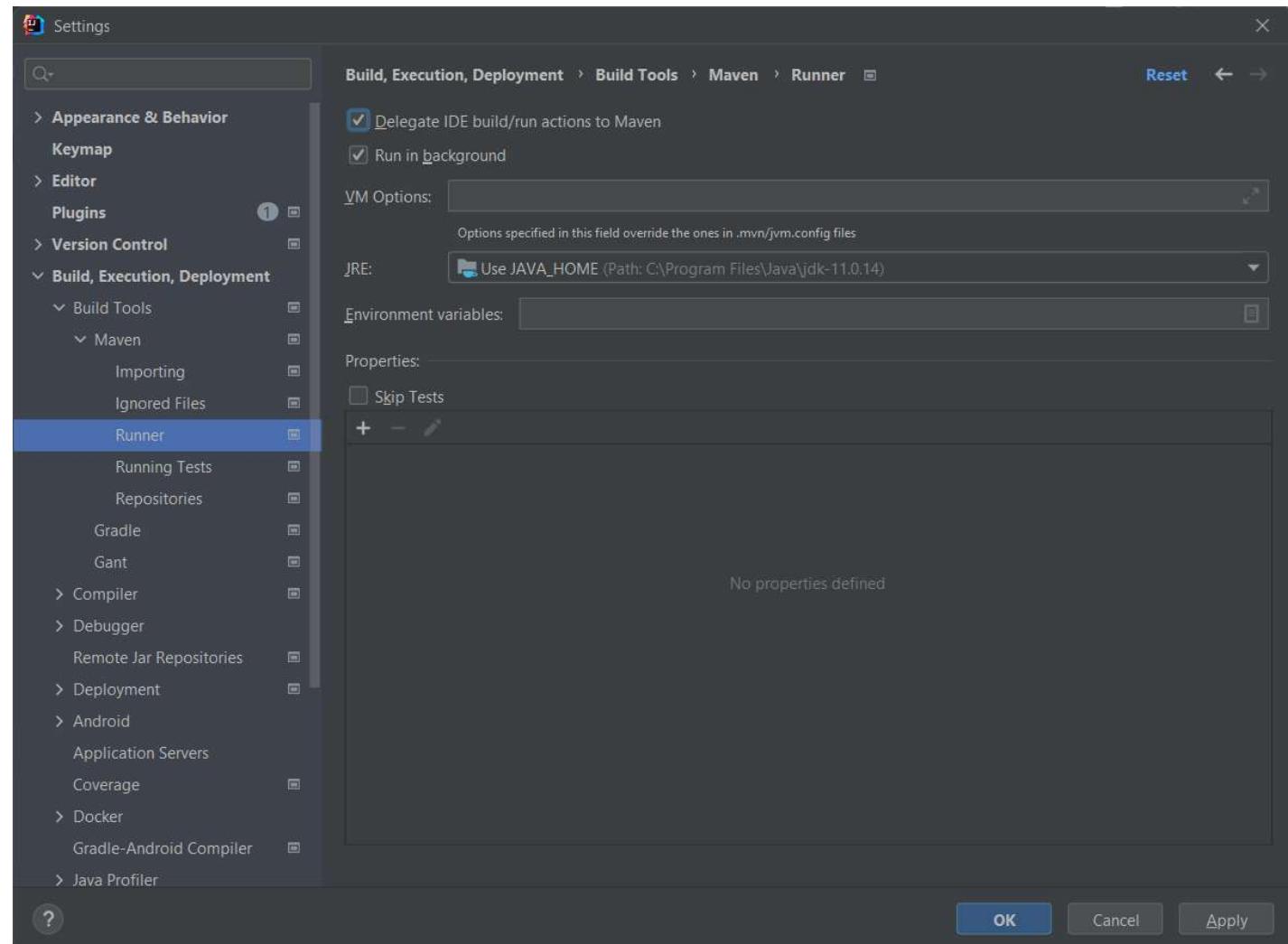
- Attention : par défaut, sous IntelliJ, il n'y a pas de checksum
 - ➡ Potentiellement vous allez télécharger des dépendances incorrectes sans le savoir <=> votre projet marchera 'mal'
- Il est conseillé d'activer le checksum afin d'éviter des erreurs de téléchargement (sur des réseaux lents ou limités)



IntelliJ et Maven - JDK for Ever

154

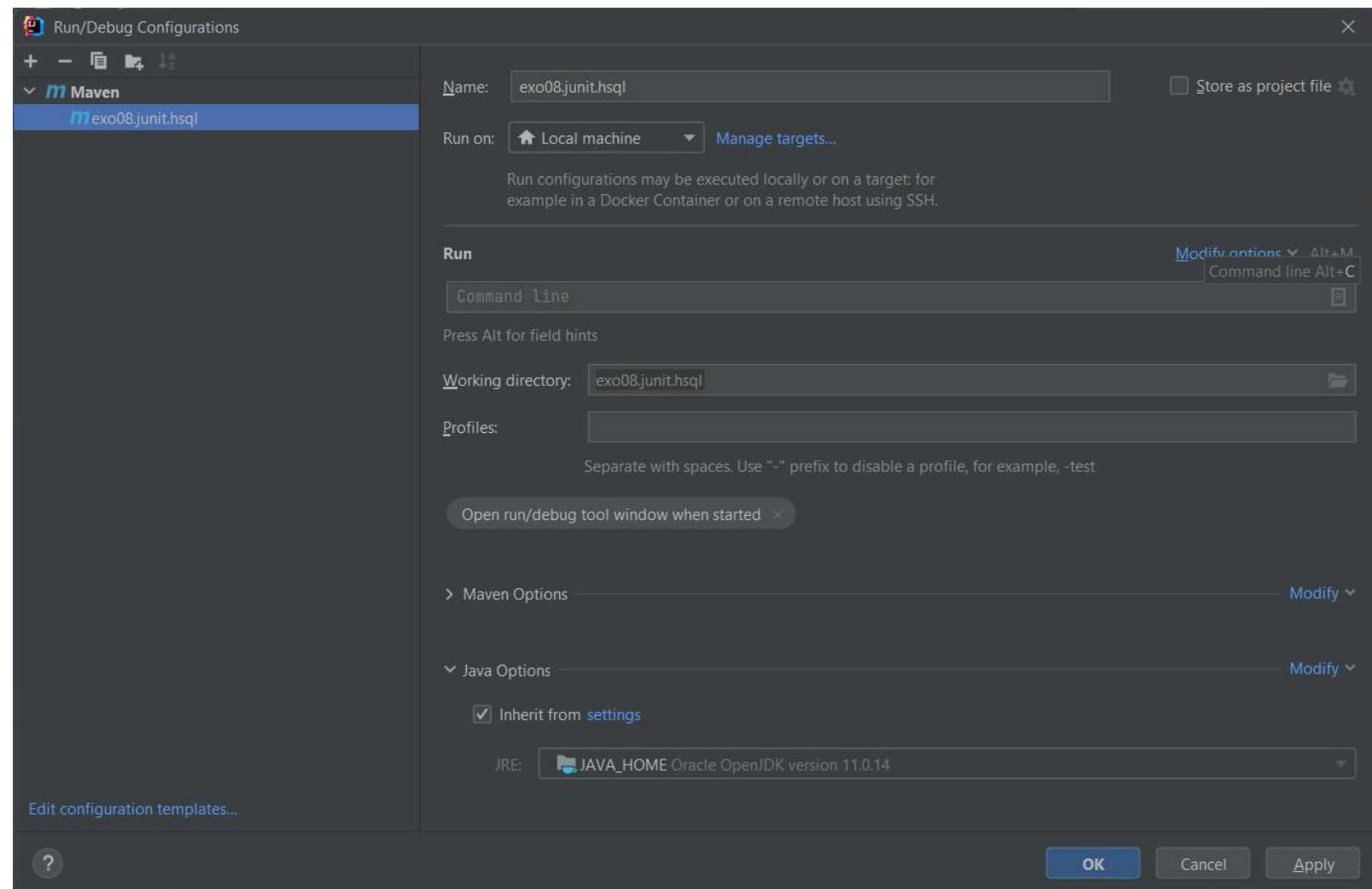
- Même si cela peut vous sembler surprenant, sous IntelliJ vous devrez toujours faire usage d'un JDK pour lancer Maven
- Vérifiez bien que vous pointer vers un JDK (et surtout PAS vers un JRE)



IntelliJ et Maven - JDK for Ever

155

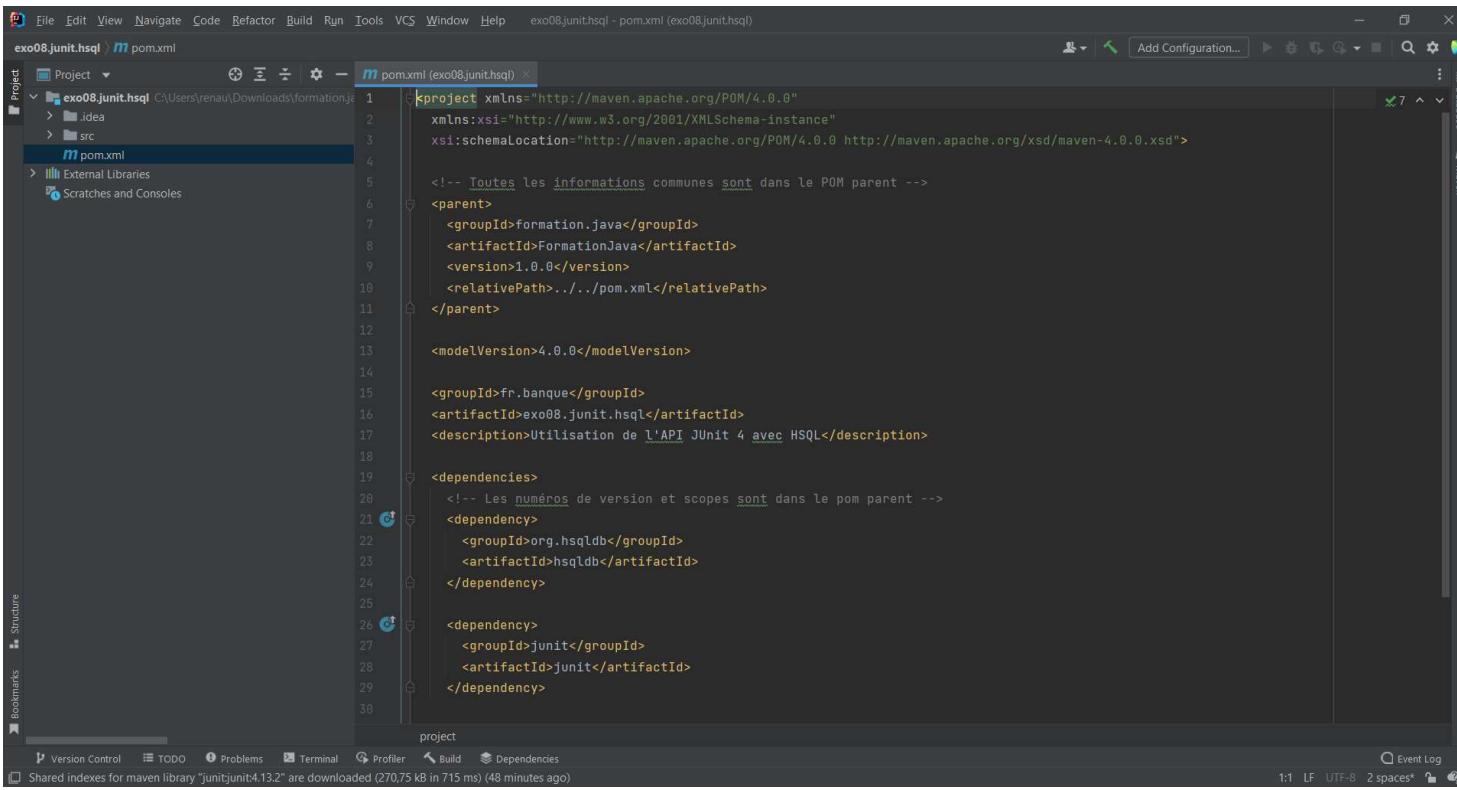
- Vous pouvez aussi gérer cette contrainte de manière plus localisée via le gestionnaire de configuration de lancement Eclipse



IntelliJ

156

- Vous avez un éditeur graphique pour manipuler votre fichier pom.xml



The screenshot shows the IntelliJ IDEA interface with the pom.xml file open. The window title is "exo08.junit.hsql - pom.xml (exo08.junit.hsql)". The left sidebar shows the project structure with a file named "pom.xml". The main editor area displays the XML code for the Maven POM file:

```
<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">

  <!-- Toutes les informations communes sont dans le POM parent -->
  <parent>
    <groupId>formation.java</groupId>
    <artifactId>FormationJava</artifactId>
    <version>1.0.0</version>
    <relativePath>../../pom.xml</relativePath>
  </parent>

  <modelVersion>4.0.0</modelVersion>

  <groupId>fr.banque</groupId>
  <artifactId>exo08.junit.hsql</artifactId>
  <description>Utilisation de l'API JUnit 4 avec HSQL</description>

  <dependencies>
    <!-- Les numéros de version et scopes sont dans le pom parent -->
    <dependency>
      <groupId>org.hsqldb</groupId>
      <artifactId>hsqldb</artifactId>
    </dependency>

    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
    </dependency>
  </dependencies>

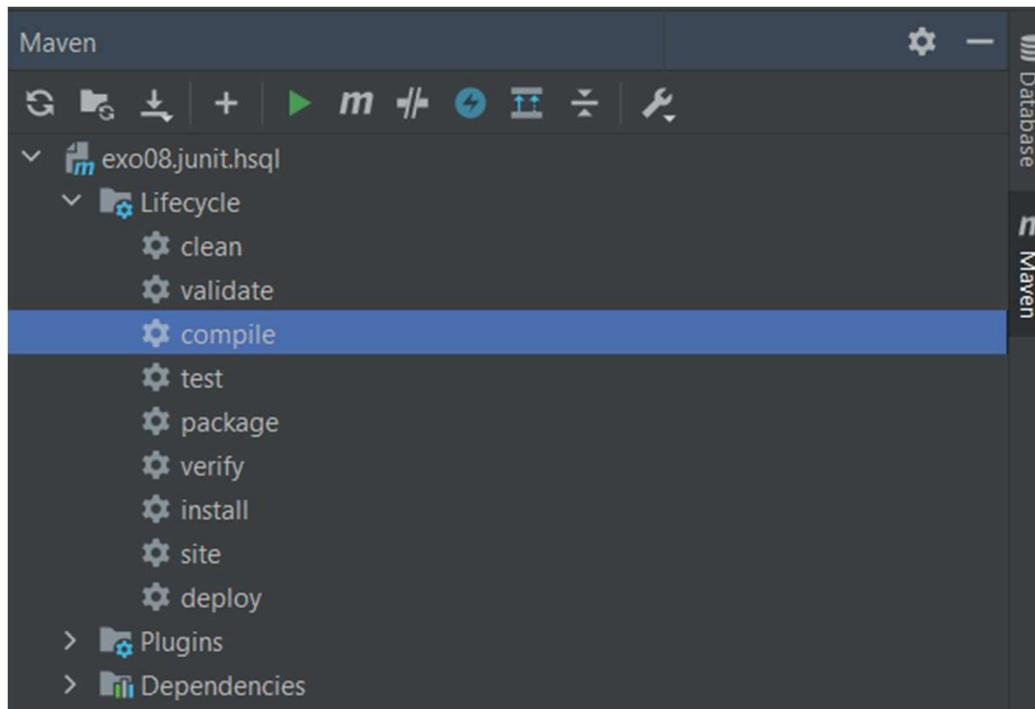
```

The code editor has syntax highlighting and code completion features. The bottom status bar shows "Shared indexes for maven library 'junit:junit:4.13.2' are downloaded (270,75 kB in 715 ms) (48 minutes ago)".

IntelliJ

157

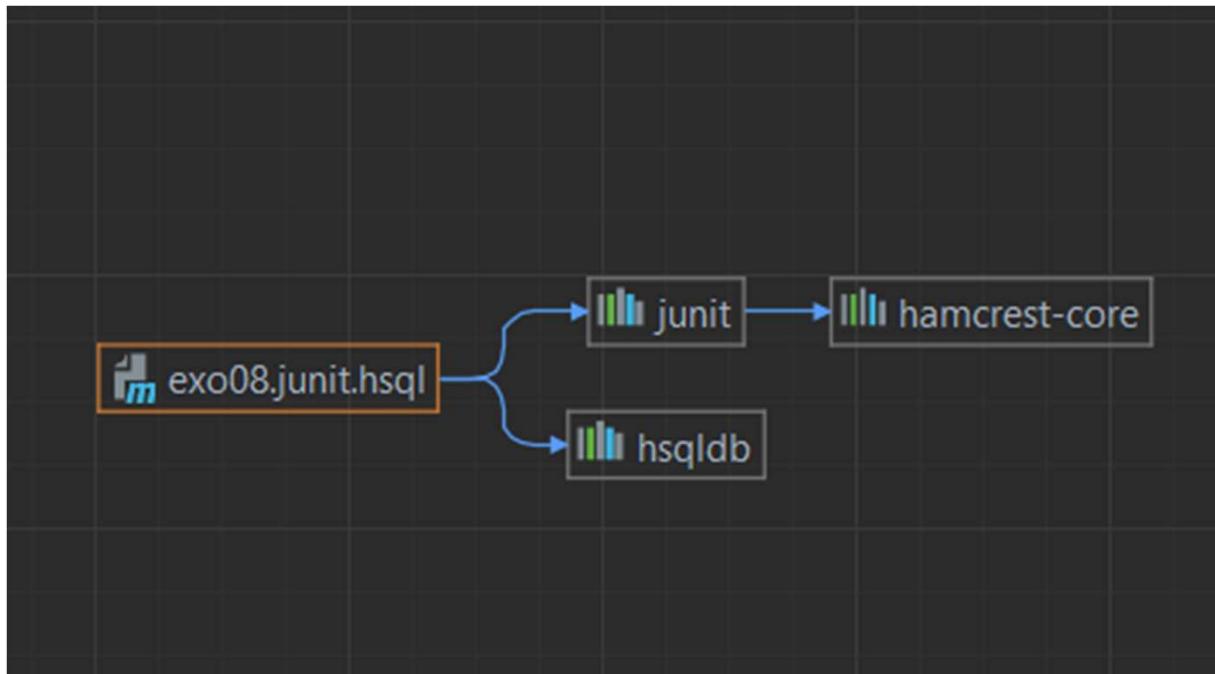
- Contrairement à Eclipse, il n'y a pas d'onglet sous cet éditeur
- Pour consulter les informations supplémentaires, faites usage de l'onglet Maven à droite de votre IDE



IntelliJ

158

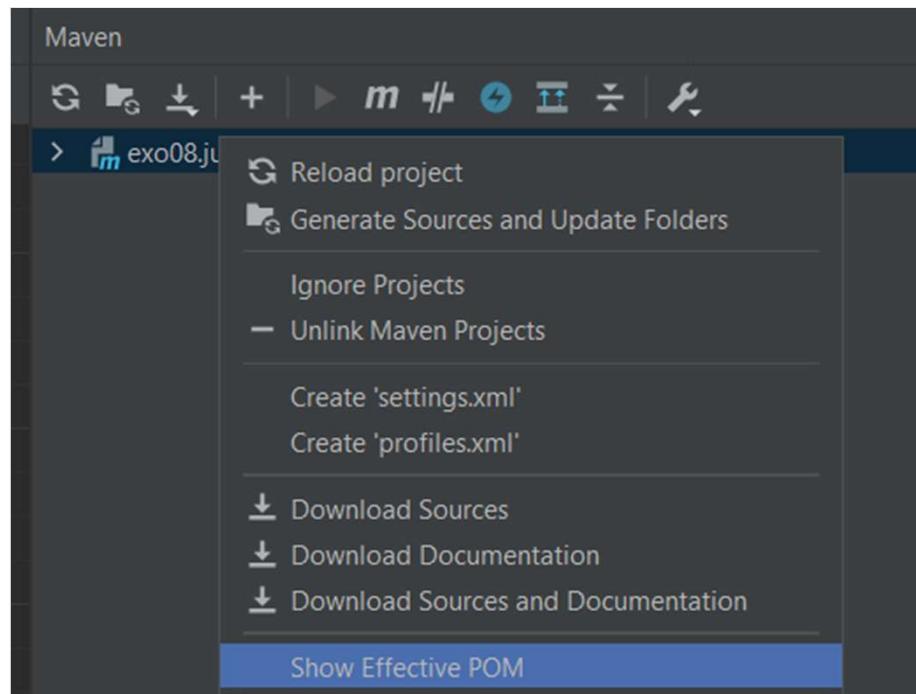
- Vous pouvez consulter vos dépendances sous la forme d'un schéma 



IntelliJ

159

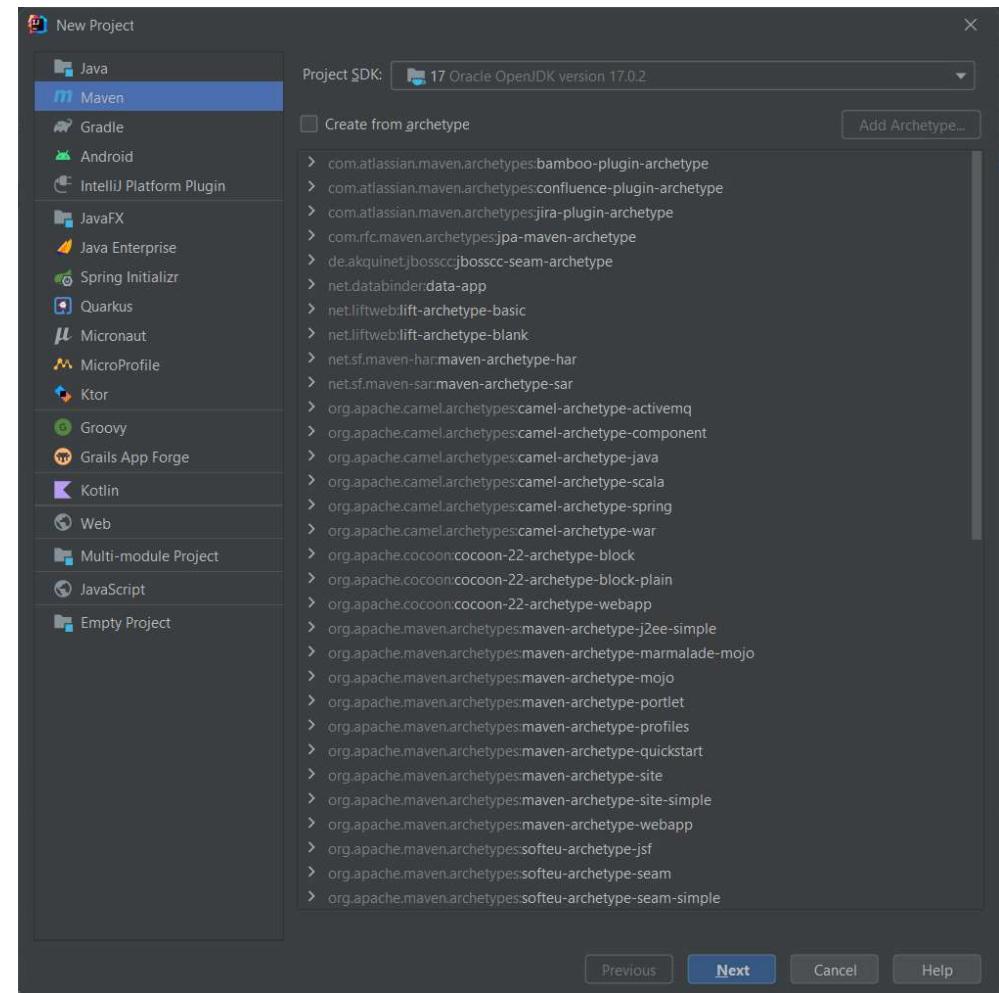
- Pour visualiser un POM résultant de la fusion de tous les POM parent il vous suffit de faire un clic droit sur votre projet puis Show Effective POM



IntelliJ - Créer un projet

160

- Pour créer un projet Maven
 - ➡ File / New / Project ...
 - ➡ Sélectionnez Maven
 - ➡ Suivez ensuite le Wizard
- Par défaut, les Archetype ne sont pas sélectionnable
- Attention à la localisation de votre projet



IntelliJ - Créer un projet

- Contrairement à Eclipse, vous pouvez indiquer tout de suite la version du Java que vous souhaitez utiliser
 - ➡ IntelliJ indiquera directement ses informations dans votre POM

IntelliJ - Importer un projet

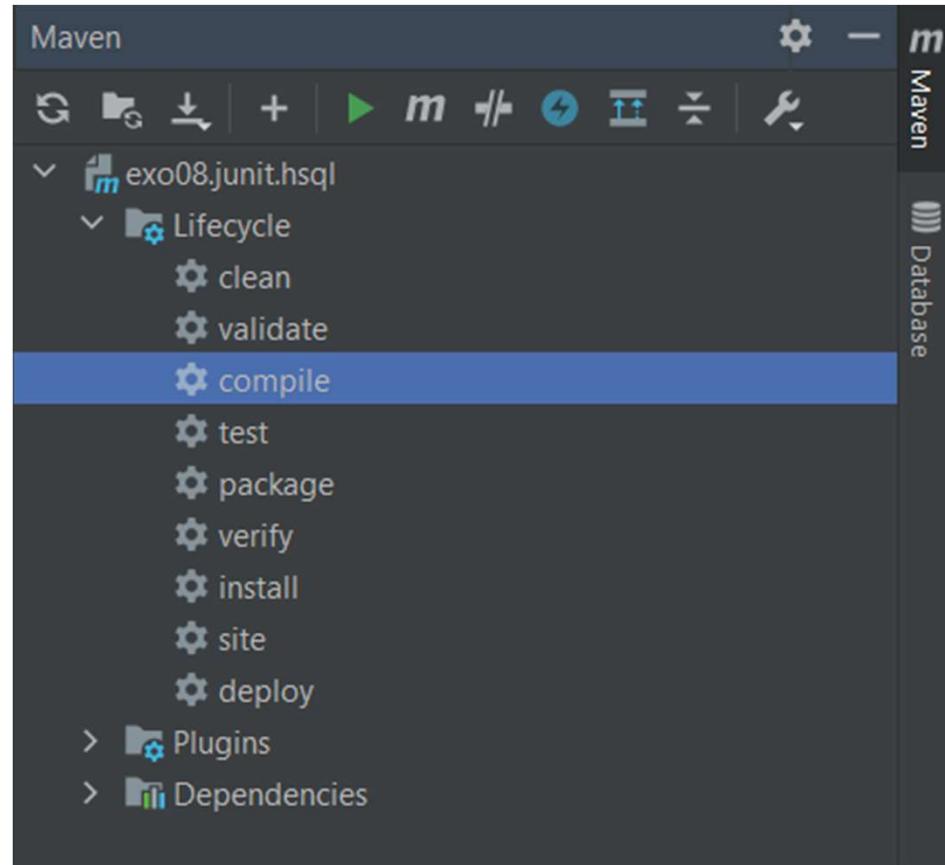
- Pour importer un projet Maven
 - ▶ Open Project
 - ▶ Indiquer le dossier contenant le fichier pom.xml
- Important :
 - ▶ si c'est aussi un projet Eclipse, il est conseillé de supprimer les fichiers Eclipse avant d'importer le projet afin qu'IntelliJ ne le considère pas comme un projet Eclipse
 - ▶ le projet importé RESTE là où il se trouve (il n'est PAS placé dans le dossier worspace)

IntelliJ - Importer / Créer

- Maven gère les dépendances en allant les télécharger d'Internet
- C'est la même chose dans votre IDE
- ATTENDEZ que toutes les dépendances minimales soient téléchargées

IntelliJ

- Pour lancer un goal Maven
 - ▶ Sur votre Onglet Maven à droite
 - ▶ Sélectionnez le projet le goal
 - ▶ Cliquez sur la flèche verte
- Vous pouvez aussi composer vos Goals en cliquant sur le m (à droite de la flèche verte)



IDE - Problème

- Sous vos IDE, selon votre version vous pouvez rencontrer des bugs
 - ▶ Liés à l'IDE
 - ▶ Liés à la version de votre Java & Maven
 - ▶ ...
- Maven vous donnera des messages assez clairs, lisez les.

IDE - Problème

166

- Dans le cas où plus rien ne fonctionne, la plus part du temps c'est le cache local de Maven qui a un problème.
- Pour résoudre un problème de cache Maven
 - ▶ Quittez / fermez votre IDE
 - ▶ Allez dans le dossier C:\Users\[VotreLoginWindows]\.m2
 - ▶ Effacez le contenu du dossier
 - ▶ Relancez votre IDE
 - ▶ Relancez un goal Maven

IDE - Problème

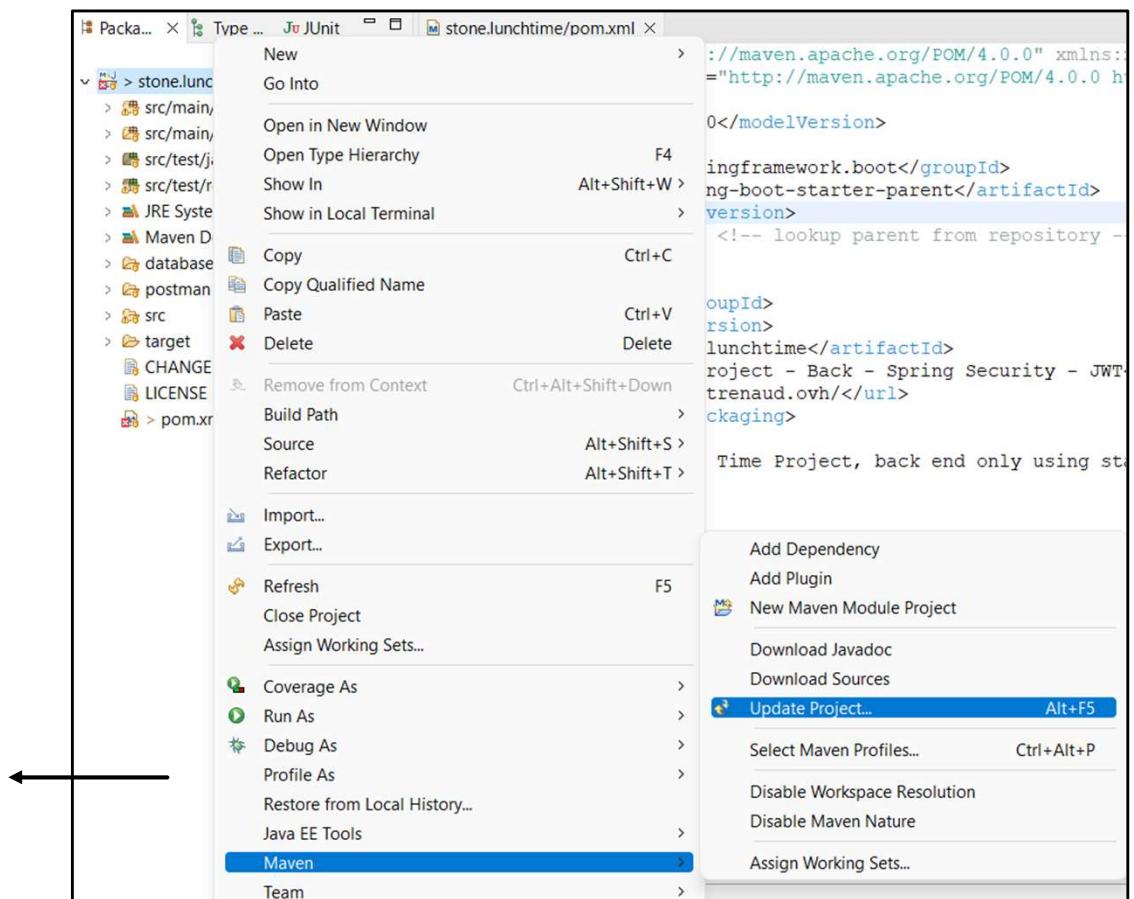
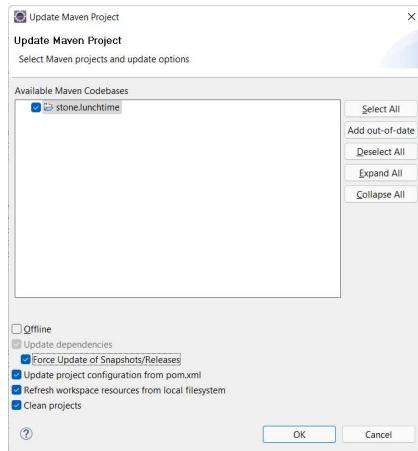
167

➤ Il faudra parfois rafraîchir Maven dans son IDE

➤ Sous Eclipse

► Sur votre projet

► Menu Maven / Update Projet



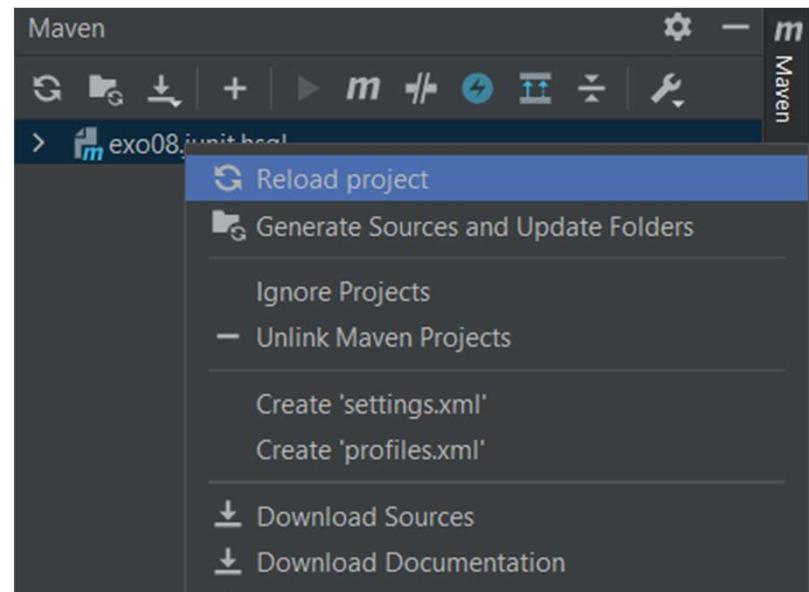
IDE - Problème

168

➤ Il faudra parfois rafraîchir Maven dans son IDE

➤ Sous IntelliJ

- ▶ Sur votre projet, onglet Maven
- ▶ Menu Reload Projet

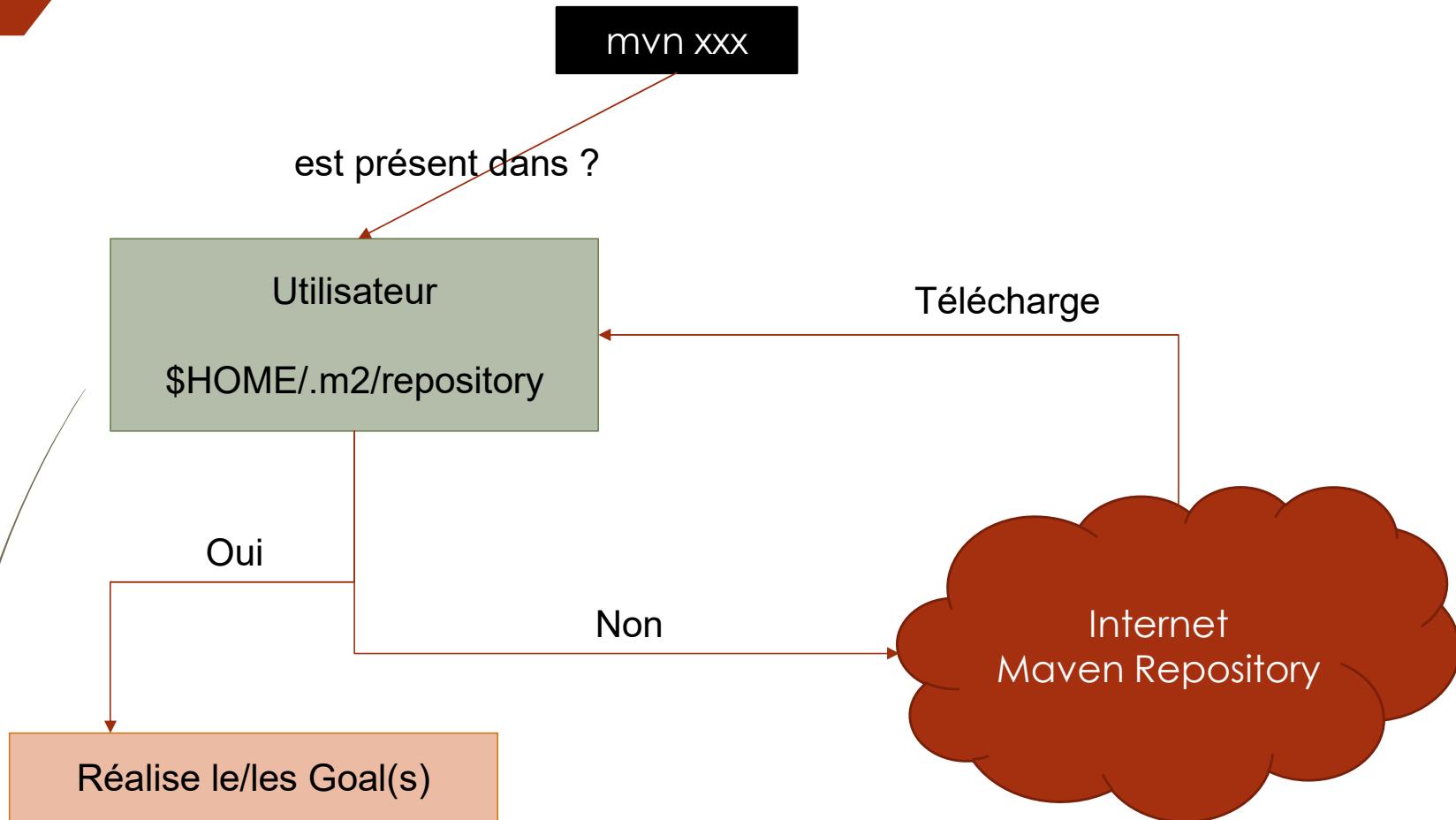


Repository

- Pour chaque dépendance dans Maven
 - ▶ Il regarde dans .m2 si présent
 - ▶ Si pas présent il va le demander à l'extérieur
- Problèmes
 - ▶ Dans une grande entreprise : cela peut générer énormément de flux réseaux
 - ▶ D'un point de vue sécurité, il se peut que certain développeur utilise des repository vérolés
 - ▶ Si l'on souhaite interdire un (ou une version) d'un framework : on ne peut pas
 - ▶ Comment mettre à disposition de son entreprise un projet ?

Repository

170



Repository - Via Nexus

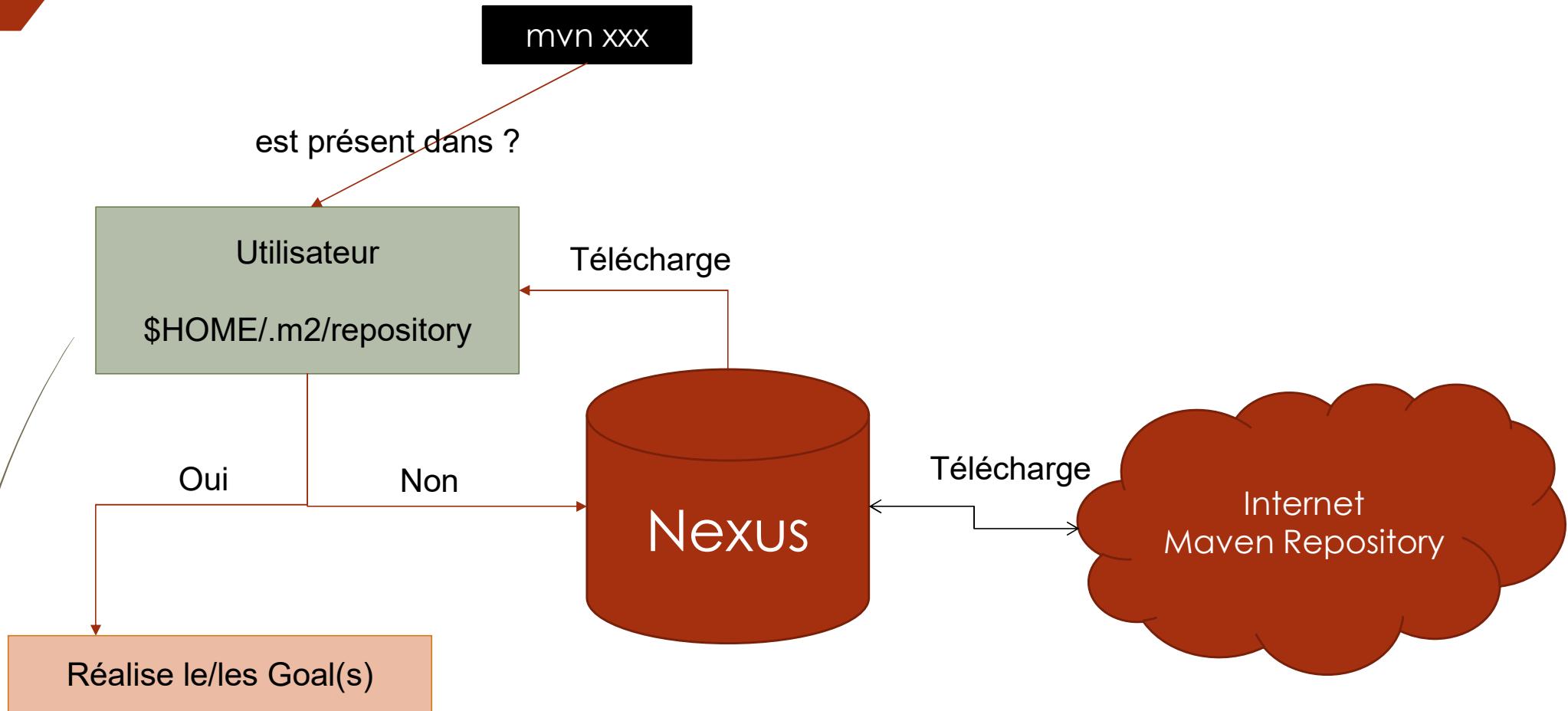
- Le Nexus est un serveur à installer dans son SI
 - Remarque : on dit Nexus comme on dit Frigidaire
- Il aura plusieurs objectifs
 - Il servira de repository interne
 - On pourra y déployer des projets (Java et autres)
 - Il servira de 'proxy'
 - Plutôt que de télécharger les dépendances d'Internet, les projets vont les télécharger du Nexus qui ira lui-même les récupérer d'Internet
 - Il sera possible d'autoriser ou d'interdire une dépendance
 - Exemple historique avec Log4J
 - Il sera possible de le sécurisé via des droits et un anti virus

Repository - Via Nexus

- Les inconvénients d'un Nexus
 - ▶ C'est un serveur à installer
 - ▶ C'est un serveur à administrer
 - ▶ C'est un serveur à sécuriser
- Il faudra donc budgéter sa mise en place et sa maintenance
 - ▶ Il n'y aura pas forcément de coup de licence logiciel à prévoir selon l'implémentation Nexus

Repository

173



Repository - Via Nexus

- Quelques exemples de Nexus pour Java
 - ▶ Nexus OSS de Sonatype (gratuit) (<https://fr.sonatype.com>)
 - Nexus Pro de Sonatype (payant)
 - ▶ JFrog (gratuit* et payant selon l'utilisation)
(<https://jfrog.com/fr/>)
 - * : hors coup d'hébergement Cloud
 - ▶ Archiva (gratuit) (<https://archiva.apache.org/>)
 - ▶ ProGet (gratuit & payant) (<https://inedo.com/proget>)

Repository - Sonatype OSS

175

➤ Installation d'un Sonatype OSS

► Téléchargez la version adaptée à votre plateforme cible

□ <https://help.sonatype.com/repomanager3/product-information/download>

□ Notez qu'il existe aussi des versions dockerisées

► Si vous êtes sous Linux

□ Augmenter le nombre maximum de fichier gérable par un processus

□ Créez le daemon chargé de l'arrêt / relance du service

► <https://help.sonatype.com/repomanager3/product-information/system-requirements>

Repository - Sonatype OSS

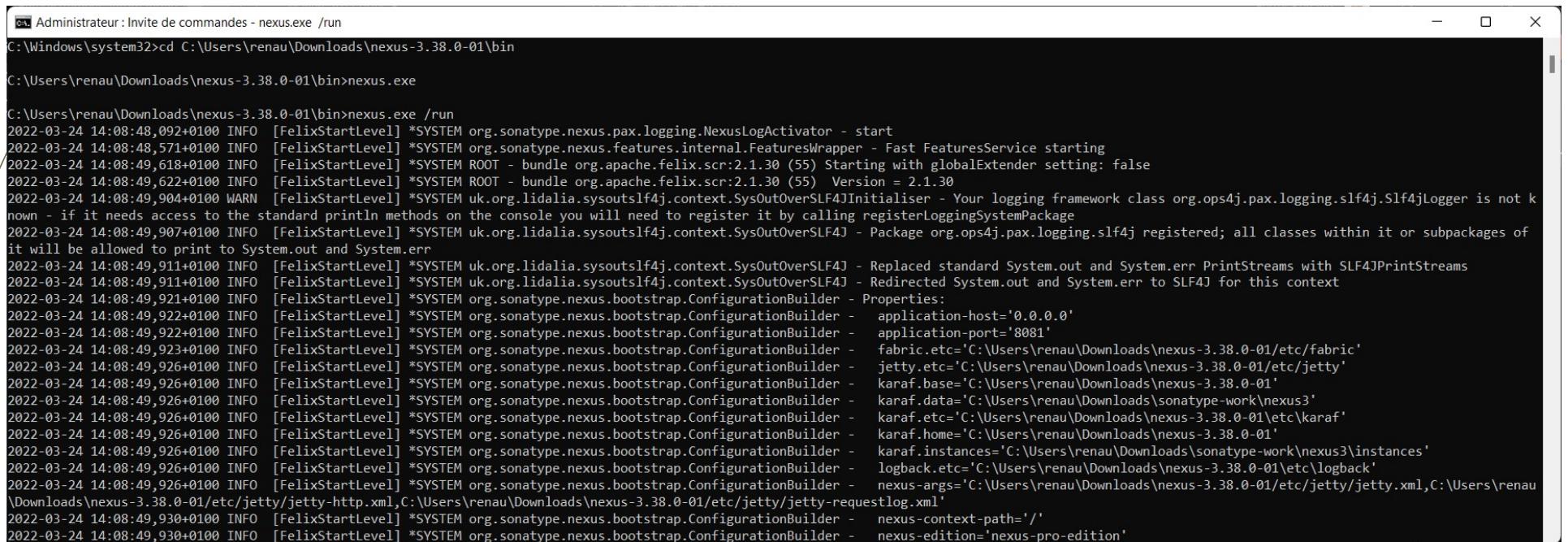
- Vous devez avoir installé un Java (JDK) en version 8 minimum sur la machine qui va lancer le Nexus
- Un minimum de 8giga de Ram
- Un espace disque à la hauteur de vos besoins
 - ➡ Sachant que l'objectif de ce processus est de 'copier' les repository Maven classique
 - ➡ Vous avez des 'recommandations' sur le type de file system utilisable

Repository - Sonatype OSS

177

➤ Sous Windows, vous pouvez lancer le processus via la commande

► /bin/nexus.exe /run



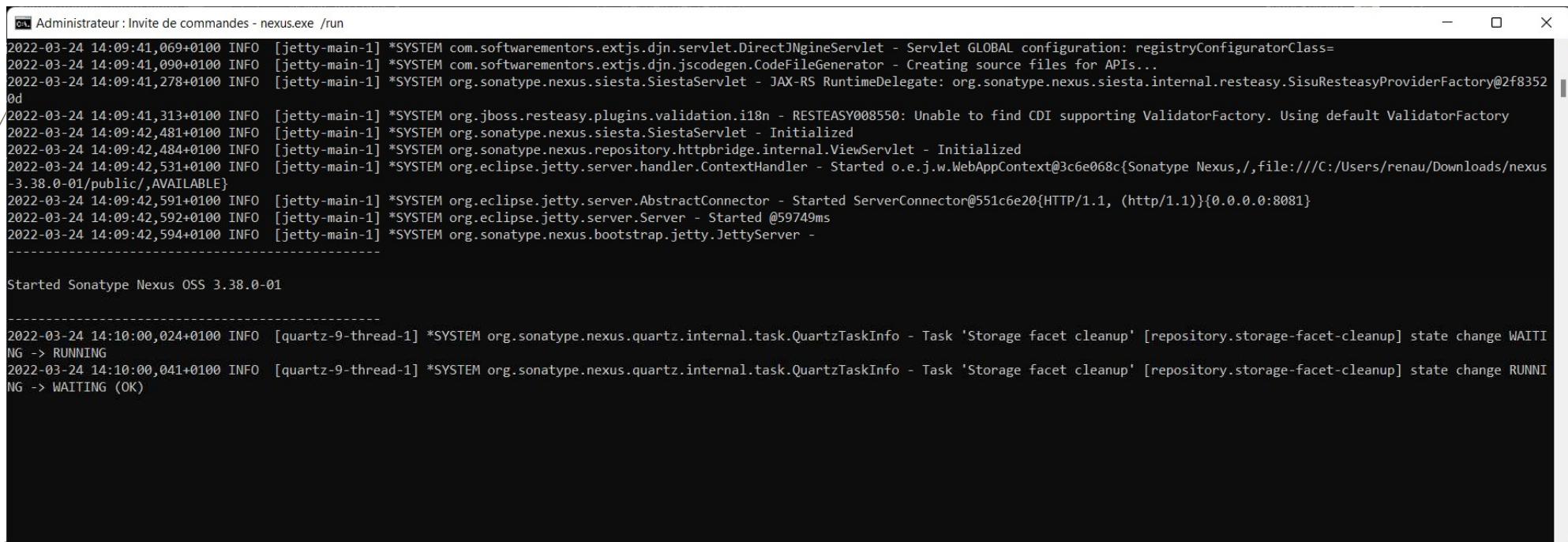
```
Administrator : Invite de commandes - nexus.exe /run
C:\Windows\system32>cd C:\Users\renau\Downloads\nexus-3.38.0-01\bin
C:\Users\renau\Downloads\nexus-3.38.0-01\bin>nexus.exe

C:\Users\renau\Downloads\nexus-3.38.0-01\bin>nexus.exe /run
2022-03-24 14:08:48,092+0100 INFO [FelixStartLevel] *SYSTEM org.sonatype.nexus.pax.logging.NexusLogActivator - start
2022-03-24 14:08:48,571+0100 INFO [FelixStartLevel] *SYSTEM org.sonatype.nexus.features.internal.FeaturesWrapper - Fast FeaturesService starting
2022-03-24 14:08:49,618+0100 INFO [FelixStartLevel] *SYSTEM ROOT - bundle org.apache.felix.scr:2.1.30 (55) Starting with globalExtender setting: false
2022-03-24 14:08:49,622+0100 INFO [FelixStartLevel] *SYSTEM ROOT - bundle org.apache.felix.scr:2.1.30 (55) Version = 2.1.30
2022-03-24 14:08:49,904+0100 WARN [FelixStartLevel] *SYSTEM uk.org.lidalia.sysoutslf4j.context.SysOutOverSLF4JInitialiser - Your logging framework class org.ops4j.pax.logging.slf4j.Slf4jLogger is not known - if it needs access to the standard println methods on the console you will need to register it by calling registerLoggingSystemPackage
2022-03-24 14:08:49,907+0100 INFO [FelixStartLevel] *SYSTEM uk.org.lidalia.sysoutslf4j.context.SysOutOverSLF4J - Package org.ops4j.pax.logging.slf4j registered; all classes within it or subpackages of it will be allowed to print to System.out and System.err
2022-03-24 14:08:49,911+0100 INFO [FelixStartLevel] *SYSTEM uk.org.lidalia.sysoutslf4j.context.SysOutOverSLF4J - Replaced standard System.out and System.err PrintStreams with SLF4JPrintStreams
2022-03-24 14:08:49,911+0100 INFO [FelixStartLevel] *SYSTEM uk.org.lidalia.sysoutslf4j.context.SysOutOverSLF4J - Redirected System.out and System.err to SLF4J for this context
2022-03-24 14:08:49,921+0100 INFO [FelixStartLevel] *SYSTEM org.sonatype.nexus.bootstrap.ConfigurationBuilder - Properties:
2022-03-24 14:08:49,922+0100 INFO [FelixStartLevel] *SYSTEM org.sonatype.nexus.bootstrap.ConfigurationBuilder - application-host='0.0.0.0'
2022-03-24 14:08:49,922+0100 INFO [FelixStartLevel] *SYSTEM org.sonatype.nexus.bootstrap.ConfigurationBuilder - application-port='8081'
2022-03-24 14:08:49,923+0100 INFO [FelixStartLevel] *SYSTEM org.sonatype.nexus.bootstrap.ConfigurationBuilder - fabric/etc='C:\Users\renau\Downloads\nexus-3.38.0-01/etc/fabric'
2022-03-24 14:08:49,926+0100 INFO [FelixStartLevel] *SYSTEM org.sonatype.nexus.bootstrap.ConfigurationBuilder - jetty/etc='C:\Users\renau\Downloads\nexus-3.38.0-01/etc/jetty'
2022-03-24 14:08:49,926+0100 INFO [FelixStartLevel] *SYSTEM org.sonatype.nexus.bootstrap.ConfigurationBuilder - karaf.base='C:\Users\renau\Downloads\nexus-3.38.0-01'
2022-03-24 14:08:49,926+0100 INFO [FelixStartLevel] *SYSTEM org.sonatype.nexus.bootstrap.ConfigurationBuilder - karaf.data='C:\Users\renau\Downloads\sonatype-work\nexus3'
2022-03-24 14:08:49,926+0100 INFO [FelixStartLevel] *SYSTEM org.sonatype.nexus.bootstrap.ConfigurationBuilder - karaf/etc='C:\Users\renau\Downloads\nexus-3.38.0-01/etc/karaf'
2022-03-24 14:08:49,926+0100 INFO [FelixStartLevel] *SYSTEM org.sonatype.nexus.bootstrap.ConfigurationBuilder - karaf.home='C:\Users\renau\Downloads\nexus-3.38.0-01'
2022-03-24 14:08:49,926+0100 INFO [FelixStartLevel] *SYSTEM org.sonatype.nexus.bootstrap.ConfigurationBuilder - karaf.instances='C:\Users\renau\Downloads\sonatype-work\nexus3\instances'
2022-03-24 14:08:49,926+0100 INFO [FelixStartLevel] *SYSTEM org.sonatype.nexus.bootstrap.ConfigurationBuilder - logback/etc='C:\Users\renau\Downloads\nexus-3.38.0-01/etc\logback'
2022-03-24 14:08:49,926+0100 INFO [FelixStartLevel] *SYSTEM org.sonatype.nexus.bootstrap.ConfigurationBuilder - nexus-args='C:\Users\renau\Downloads\nexus-3.38.0-01/etc/jetty/jetty.xml,C:\Users\renau\Downloads\nexus-3.38.0-01/etc/jetty/jetty-requestlog.xml'
2022-03-24 14:08:49,930+0100 INFO [FelixStartLevel] *SYSTEM org.sonatype.nexus.bootstrap.ConfigurationBuilder - nexus-context-path='/'
2022-03-24 14:08:49,930+0100 INFO [FelixStartLevel] *SYSTEM org.sonatype.nexus.bootstrap.ConfigurationBuilder - nexus-edition='nexus-pro-edition'
```

Repository - Sonatype OSS

178

- Attendez que la console (ou que le fichier de log) vous indique
 - Started Sonatype Nexus OSS



```
Administrator : Invite de commandes - nexus.exe /run
2022-03-24 14:09:41,069+0100 INFO  [jetty-main-1] *SYSTEM com.softwarementors.extjs.djn.servlet.DirectJNginxServlet - Servlet GLOBAL configuration: registryConfiguratorClass=
2022-03-24 14:09:41,090+0100 INFO  [jetty-main-1] *SYSTEM com.softwarementors.extjs.djn.jscodegen.CodeFileGenerator - Creating source files for APIs...
2022-03-24 14:09:41,278+0100 INFO  [jetty-main-1] *SYSTEM org.sonatype.nexus.siesta.SiestaServlet - JAX-RS RuntimeDelegate: org.sonatype.nexus.siesta.internal.resteasy.SisuResteasyProviderFactory@2f8352
0d
2022-03-24 14:09:41,313+0100 INFO  [jetty-main-1] *SYSTEM org.jboss.resteasy.plugins.validation.i18n - RESTEASY008550: Unable to find CDI supporting ValidatorFactory. Using default ValidatorFactory
2022-03-24 14:09:42,481+0100 INFO  [jetty-main-1] *SYSTEM org.sonatype.nexus.siesta.SiestaServlet - Initialized
2022-03-24 14:09:42,484+0100 INFO  [jetty-main-1] *SYSTEM org.sonatype.nexus.repository.httpbridge.internal.ViewServlet - Initialized
2022-03-24 14:09:42,531+0100 INFO  [jetty-main-1] *SYSTEM org.eclipse.jetty.server.handler.ContextHandler - Started o.e.j.w.WebAppContext@3c6e068c{Sonatype Nexus,/,file:///C:/Users/renau/Downloads/nexus-3.38.0-01/public/,AVAILABLE}
2022-03-24 14:09:42,591+0100 INFO  [jetty-main-1] *SYSTEM org.eclipse.jetty.server.AbstractConnector - Started ServerConnector@551c6e20{HTTP/1.1, (http/1.1)}{0.0.0.0:8081}
2022-03-24 14:09:42,592+0100 INFO  [jetty-main-1] *SYSTEM org.eclipse.jetty.server.Server - Started @59749ms
2022-03-24 14:09:42,594+0100 INFO  [jetty-main-1] *SYSTEM org.sonatype.nexus.bootstrap.jetty.JettyServer -
-----
Started Sonatype Nexus OSS 3.38.0-01
-----
2022-03-24 14:10:00,024+0100 INFO  [quartz-9-thread-1] *SYSTEM org.sonatype.nexus.quartz.internal.task.QuartzTaskInfo - Task 'Storage facet cleanup' [repository.storage-facet-cleanup] state change WAITING -> RUNNING
2022-03-24 14:10:00,041+0100 INFO  [quartz-9-thread-1] *SYSTEM org.sonatype.nexus.quartz.internal.task.QuartzTaskInfo - Task 'Storage facet cleanup' [repository.storage-facet-cleanup] state change RUNNING -> WAITING (OK)
```

Repository - Sonatype OSS

- Prenez votre navigateur et allez sur
 - ➡ <http://localhost:8081/>
- Connectez vous en tant qu'administrateur
 - ➡ Login : admin
 - ➡ Password : sonatype-work\nexus3\admin.password

Repository - Sonatype OSS

- Dans l'élément #admin/repository/repositories
- Ajouter deux repository
 - ➡ De type Maven 2 / hosted
 - ➡ L'un portera le nom my-released, l'autre my-snapshots
- Ils nous serviront pour le goal **deploy**

Repository - Sonatype OSS

181

The screenshot shows the Sonatype Nexus Repository Manager interface version 3.38.0-01. The left sidebar is titled "Administration" and contains the following sections:

- Repository
 - Repositories** (selected)
 - Blob Stores
 - Cleanup Policies
 - Content Selectors
 - Proprietary Repositories
 - Routing Rules
- Security
 - Privileges
 - Roles
 - Users
 - Anonymous Access
 - LDAP
 - Realms

The main content area is titled "Repositories" and shows the creation of a new repository. The steps are:

- Step 1: "Select Recipe" (highlighted in blue) → "Create Repository: maven2 (hosted)"
- Step 2: "Name": my-snapshots
- Step 3: "Online": checked
- Step 4: "Maven 2" section
 - "Version policy": Snapshot
 - "Layout policy": Strict
 - "Content Disposition": Inline
- Step 5: "Storage" section
 - "Blob store": default
- Step 6: "Strict Content Type Validation": checked

Repository - Sonatype OSS

182

- Ajouter un repository de type Maven2 / proxy
 - ➡ Qui portera comme nom my-proxy
 - ➡ De type Maven 2 / proxy
 - ➡ Sur <https://repo1.maven.org/maven2/>

- Il nous servira pour jouer le rôle de cache global externe

The screenshot shows the Sonatype Nexus Repository Manager interface version 3.38.0-01. The left sidebar is titled 'Administration' and contains sections for 'Repository' (selected), 'Blob Stores', 'Cleanup Policies', 'Content Selectors', 'Proprietary Repositories', 'Routing Rules', 'Security' (with 'Privileges', 'Roles', 'Users', 'Anonymous Access', 'LDAP', 'Realms', and 'SSL Certificates'), and 'SSL Certificates'. The main right panel is titled 'Repositories / my-proxy'. It shows a summary of the repository settings:

- Name:** my-proxy
- Format:** maven2
- Type:** proxy
- URL:** http://localhost:8081/repository/my-proxy/
- Online:** If checked, the repository accepts incoming requests

Under the 'Maven 2' section, the 'Version policy:' is set to 'Mixed' and the 'Layout policy:' is set to 'Validate that all paths are maven artifact or metadata paths' with 'Strict' selected.

Under the 'Content Disposition:' section, it says 'Add Content-Disposition header as 'Attachment' to disable some content from being inline in a browser.' with 'Inline' selected.

Under the 'Proxy' section, the 'Remote storage:' field contains the URL 'https://repo1.maven.org/maven2/'.

Repository - Sonatype OSS

183

- Ajouter un repository de type Maven2 / group
 - ➡ Qui portera comme nom my-group
 - ➡ De type Maven 2 / group
 - ➡ Il regroupera nos 3 repos

- Il nous servira d'interlocuteur global

The screenshot shows the Sonatype Nexus Repository Manager interface. The left sidebar has a dark header "Administration" and a list of sections: "Repository" (selected), "Repositories" (highlighted in green), "Blob Stores", "Cleanup Policies", "Content Selectors", "Proprietary Repositories", "Routing Rules", "Security" (expanded), "Privileges", "Roles", "Users", "Anonymous Access", "LDAP", "Realms", and "SSL Certificates". The main content area shows a "Repositories" section with a "my-group" folder. Below it are buttons for "Delete repository" and "Invalidate cache". A "Settings" button is also present. The "Storage" section shows a "Blob store:" dropdown set to "default". Under "Strict Content Type Validation", there is a checked checkbox for validating MIME types. The "Group" section has a "Member repositories:" heading and a "Available" list containing "maven-snapshots", "maven-central", "maven-releases", and "maven-public". To the right, a "Members" list shows "my-release", "my-proxy", and "my-snapshots" with up and down arrows for reordering. A search bar at the top right says "Search components".

Repository - Sonatype OSS

184

➤ Nos 4 repository :

The screenshot shows the Sonatype Nexus Repository Manager interface. The top navigation bar includes the logo, version (OSS 3.38.0-01), a cube icon, a gear icon, and a search bar. The left sidebar has a 'Administration' section with options: Repository (selected, highlighted in green), Repositories (also highlighted in green), Blob Stores, Cleanup Policies, Content Selectors, and Proprietary Repositories. The main content area is titled 'Repositories Manage repositories' and features a 'Create repository' button. A table lists four repositories:

| Name ↑ | Type | Format | Status |
|--------------|--------|--------|---------------------------|
| my-group | group | maven2 | Online |
| my-proxy | proxy | maven2 | Online - Remote Available |
| my-release | hosted | maven2 | Online |
| my-snapshots | hosted | maven2 | Online |

Repository - Sonatype OSS

➤ Après avoir potentiellement créé des utilisateurs dédiés

1. S'authentifier (pour le deploy)

- ▶ Modifier le fichier [USER]/.m2/settings.xml
- ▶ Pour chaque repository associé au deploy, indiquer
 - L'id <=> le name du repository (dans notre exemple my-xxx)
 - Le username (le login de l'utilisateur Nexus), dans notre cas admin)
 - Le password (le password de l'utilisateur Nexus)

Repository - Sonatype OSS

186

➤ S'authentifier

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.1.0 http://maven.apache.org/xsd/settings-1.1.0.xsd">

<servers>
  <server>
    <id>my-snapshots</id> <!-- bien conserver le nom donne -->
    <username>admin</username>
    <password>votre-mdp</password>
  </server>

  <server>
    <id>my-release</id>
    <username>admin</username>
    <password> votre-mdp </password>
  </server>
</servers>
</settings>
```

Repository - Sonatype OSS

187

➤ Ajouter le mirror global

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.1.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.1.0 http://maven.apache.org/xsd/settings-1.1.0.xsd">

  <servers>
    <!-- vos identifiants -->
  </servers>

  <mirrors>
    <mirror>
      <id>my-group</id>
      <url>http://localhost:8081/repository/my-group/</url>
      <mirrorOf>*</mirrorOf>
    </mirror>
  </mirrors>

</settings>
```

Repository - Sonatype OSS

- Dans le fichier pom.xml du projet
 - ➡ Vous pouvez indiquer les informations relatives au goal deploy

```
<distributionManagement>
    <repository>
        <id>my-release</id>
        <url>http://localhost:8081/repository/my-release/</url>
    </repository>

    <snapshotRepository>
        <id>my-snapshots</id>
        <url>http://localhost:8081/repository/my-snapshots/</url>
    </snapshotRepository>
</distributionManagement>
```

Repository - Sonatype OSS

- Dans le fichier pom.xml du projet
 - ➡ Vous pouvez indiquer les informations relatives au mirror

```
<repositories>
  <repository>
    <id>my-group</id>
    <url>http://localhost:8081/repository/my-group/</url>
  </repository>
</repositories>
```

Repository - Sonatype OSS

➤ Notez que

- ▶ les deux URL's correspondent à ceux que l'on a créé dans le Nexus
- ▶ Quand vous allez lancer un goal, notez que les URLs devraient faire référence à votre Nexus

```
Downloading from my-group: http://localhost:8081/repository/my-group/org/codehaus/plexus/plexus/3.3.2/plexus-3.3..  
Downloaded from my-group: http://localhost:8081/repository/my-group/org/codehaus/plexus/plexus/3.3.2/plexus-3.3.2  
Downloading from my-group: http://localhost:8081/repository/my-group/org/vafer/jdependency/2.4.0/jdependency-2.4.0  
Downloaded from my-group: http://localhost:8081/repository/my-group/org/vafer/jdependency/2.4.0/jdependency-2.4.0  
Downloading from my-group: http://localhost:8081/repository/my-group/org/ow2/asm/asm-util/8.0/asm-util-8.0.pom  
Downloaded from my-group: http://localhost:8081/repository/my-group/org/ow2/asm/asm-util/8.0/asm-util-8.0.pom (0 B at 0 B/s)  
Downloading from my-group: http://localhost:8081/repository/my-group/com/google/guava/guava/28.2-android/guava-28.  
Downloaded from my-group: http://localhost:8081/repository/my-group/com/google/guava/guava/28.2-android/guava-28.  
Downloading from my-group: http://localhost:8081/repository/my-group/com/google/guava/guava-parent/28.2-android/gu  
Downloaded from my-group: http://localhost:8081/repository/my-group/com/google/guava/guava-parent/28.2-android/gu  
Downloading from my-group: http://localhost:8081/repository/my-group/com/google/guava/failureaccess/1.0.1/failureac  
Downloaded from my-group: http://localhost:8081/repository/my-group/com/google/guava/failureaccess/1.0.1/failureac  
Downloading from my-group: http://localhost:8081/repository/my-group/com/google/guava/guava-parent/26.0-android/gu  
Downloaded from my-group: http://localhost:8081/repository/my-group/com/google/guava/guava-parent/26.0-android/gu  
Downloading from my-group: http://localhost:8081/repository/my-group/com/google/guava/listenablefuture/9999.0-empt  
ure-9999.0-empty-to-avoid-conflict-with-guava.pom  
Downloaded from my-group: http://localhost:8081/repository/my-group/com/google/guava/listenablefuture/9999.0-empt  
re-9999.0-empty-to-avoid-conflict-with-guava.pom (0 B at 0 B/s)  
Downloading from my-group: http://localhost:8081/repository/my-group/org/checkerframework/checker-compat-qual/2.5  
Downloaded from my-group: http://localhost:8081/repository/my-group/org/checkerframework/checker-compat-qual/2.5
```