

Formation MAVEN

The logo for Maven, featuring the word "maven" in a bold, sans-serif font. The letter "a" is orange, while the other letters are black. The logo is positioned to the left of a vertical grey line, which is part of a larger graphic element consisting of a horizontal line and a vertical line meeting at a right angle.

- Partie 1 : principes MAVEN p.2
- Partie 2 : premier projet MAVEN p.26
- Partie 3 : projets multi modules p.40
- Partie 4 : rapports p.55
- Partie 5 : maven 3 p.67

Formation MAVEN : Partie 1

les principes de Maven

Présentation

- MAVEN est une « framework » de gestion de projets
- Plus précisément, MAVEN est :
 - Un ensemble de standards
 - Une « repository » d'un format particulier
 - Du logiciel pour gérer et décrire un projet
- Il fournit un cycle de vie standard pour
 - Construire, Tester, Déployer des projets
 - Selon une logique commune
- Il s'articule autour d'une déclaration commune de projet que l'on appelle le POM
 - Project Object Model
- MAVEN, c'est une façon de voir un produit comme une collection de composants inter-dépendants qui peuvent être décrits sous un format standard

Origines de MAVEN

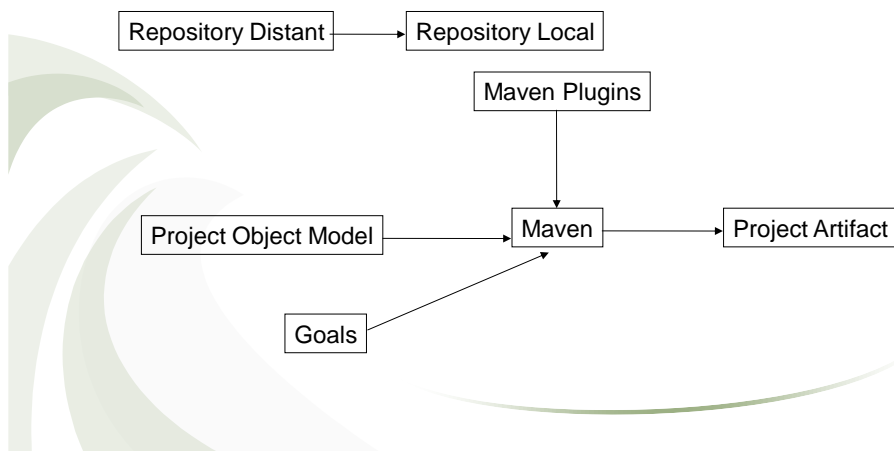
- Jusque très récemment, les processus de construction de chaque prouit pouvaient être différents
 - Ex : le processus de construction pour le projet Tomcat est différent est différent de celui de STRUTS
 - Les développeurs au sein d'un même projet à « l'Apache Software Foundation » utilisaient chacun une approche différente.
 - Le manque de vue globale entraîne l'approche
 - « copy » and « paste » pour construire un projet à partir d'un autre.
 - Génère rapidement des incohérences
 - Finalement, les développeurs passent beaucoup de temps à la construction de leur application plutôt qu'à leur contenu.
- MAVEN a permis aussi de faciliter les tâches de documentation, tests, rapports d'exécution (metrics) et de déploiement
- MAVEN est très différent de ANT
 - ANT s'attarde à la notion de tâche
 - MAVEN n'est pas qu'un outil de construction (build)
 - Il voit le projet dans son ensemble
 - Ceci dit, des plugins MAVEN permettent de maintenir les scripts ANT depuis MAVEN

Principes

- MAVEN fournit un modèle détaillé applicable à n'importe quel projet
 - Il fournit un langage commun
 - Il fournit une abstraction comme celle de l'interface en JAVA ou autre :
 - Un camion et une voiture de tourisme ont des « implémentations » différentes ; ils se conduisent pourtant de la même façon...
- La structure d'un projet MAVEN et son contenu sont déclarés dans un POM
 - Ce POM est purement déclaratif
 - Le développeur va déclarer des objectifs dans ce POM et des dépendances
 - L'orchestration des tâches qui suivront (compile, test, assemblage, installation) sera gérée par des plugins appropriés et le POM
- MAVEN utilise beaucoup de valeurs par défaut, rendant la génération plus facile
- En résumé, MAVEN apporte :
 - Cohérence, réutilisabilité, agilité, maintenabilité
- Les trois éléments importants dans MAVEN sont :
 - Directory standard pour les projets
 - Facilite la lecture de tout nouveau projet, structure connue
 - Peut être modifiée par de nouvelles directores, mais impacte la complexité du POM
 - Un projet MAVEN fournit une seule sortie (output)
 - MAVEN par contre incite à découper un super projet (ex : client/serveur) en autant de projets individuels facilement réutilisables, donc une sortie par projet réutilisable, c'est le SoC (Separation of Concern)
 - Une convention de nommage standard
 - Directories, fichiers

Architecture Maven

- Les éléments principaux de Maven



Le POM

■ Le POM

- ❑ Maven introduit le concept d'un descripteur de projet, ou bien POM.
- ❑ Le POM (Project Object Model) décrit le projet, fournissant toutes les informations nécessaires pour accomplir une série de tâches préconstruites.
- ❑ Ces tâches ou *goals*, utilisent le POM pour s'exécuter correctement.
- ❑ Des plugins peuvent être développés et utilisés dans de multiples projets de la même manière que les tâches préconstruites.

■ Les Goal

- ❑ En introduisant un descripteur de projet, Maven nécessite qu'un développeur fournisse des informations *sur ce qui est construit*.
- ❑ Cela diffère des systèmes de construction traditionnels où les développeurs informent le système sur la *manière de construire*.
- ❑ Les développeurs peuvent se concentrer sur la logique de développement.

Les Goals

■ *Goals Communs :*

L'introduction des cycles de vies dans Maven a considérablement réduit le nombre de goals qu'un développeur doit absolument savoir maîtriser.

■ Les goals suivants seront toujours considérés comme utiles (et peuvent être utilisés dès que le descripteur de projet a été généré):

■ Il sont à fournir au lancement de la commande maven mvn

- ❑ `clean:clean` Supprime tous les artéfacts et les fichiers intermédiaires créés par Maven
- ❑ `eclipse:eclipse` Génère des fichiers projets pour Eclipse
- ❑ `javadoc:javadoc` Génère la documentation Java pour le projet
- ❑ `antrun:run` Exécute une cible ANT
- ❑ `clover:check` Génère un rapport d'analyse du code
- ❑ `checkstyle:checkstyle` Génère un rapport sur le style de codage du projet
- ❑ `site:site` Crée un site web de documentation pour le projet. Ce site inclura beaucoup de rapports d'informations concernant le projet.

Principes

- MAVEN s'est appliqué Le SoC (Separation of Concern) aux phases mêmes de construction.
 - Les phases de construction par MAVEN sont fournies dans des Plugins, c'est de la responsabilité de MAVEN d'exécuter ces plugins dans l'ordre adéquat.
 - Il y a des plugins pour :
 - Compiler, lancer les test, créer les JARs, créer les JAVADOCS, etc...
 - L'exécution de ces plugins est coordonnée par le POM
 - Exemple de POM qui permettra de compiler, tester, documenter l'appli.
 - Comment est-ce possible ?
 - Grâce au « SUPER POM »

```
<project>
<modelVersion>4.0.0</modelVersion>
<groupId>com.mycompany.app</groupId>
<artifactId>my-app</artifactId>
<packaging>jar</packaging>
<version>1.0-SNAPSHOT</version>
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
</dependencies>
</project>
```

- Le super POM est à MAVEN ce que `java.lang.Object` est à JAVA
 - Vous vous évitez ainsi à réécrire tout ce qui est décrit dans le SUPER POM, et qui se répète pour chaque nouveau POM

Principes

- L'exemple précédent est simple mais possède les essentiels :
 - `modelVersion` : version du modèle objet que ce POM utilise .
 - `groupId` : nom qualifié de l'organisation qui a créé ce projet (un peu le namespace de xml)
 - `artifactId` : nom de l'artefact (sortie) excepté les versions.
 - MAVEN produira `artifactId-version.extension`
 - Ex : `myapp-1.0.jar`
 - `packaging` : ce qui est indiqué ici générera des cycles de fabrication par MAVEN très différents selon ce type
 - `version` : en liaison avec l'`artifactId`
 - `name` : pour la documentation générée par MAVEN
 - `url` : où le site du projet peut être trouvé
 - `description` : simple description du projet
- Pour une complète description des éléments, voir
 - <http://maven.apache.org/maven-model/maven.html>

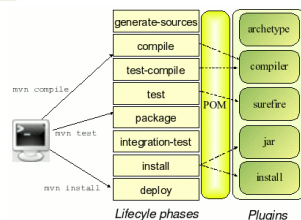
Principes

■ Les snapshots

- Par convention, une version en cours de développement d'un projet voit son numéro de version suivi d'un -SNAPSHOT.
- Ainsi un projet en version 2.0-SNAPSHOT signifie que cette version est une pré-version de la version 2.0, en cours de développement.
- Ce concept de SNAPSHOT est particulièrement important pour Maven.
 - En effet, dans la gestion des dépendances, Maven va chercher à mettre à jour les versions SNAPSHOT régulièrement pour prendre en compte les derniers développements.
 - Utiliser une version SNAPSHOT permet de bénéficier des dernières fonctionnalités d'un projet, mais en contre-partie, cette version peut être (et est) appelée à être modifiée de façon importante, sans aucun préavis.

Principes

- Pour construire une application, MAVEN s'appuie sur un cycle de construction, constitué de plusieurs phases.
 - Si vous demandez d'exécuter une phase, alors toutes les phases précédentes seront forcement d'abord appelées, mais on peut changer ce mode grâce aux conventions
 - Ex : demander à MAVEN de compiler entraînera forcement les phases précédentes de :
§ Validate, initialize, generate-sources, process-sources, generate-resources
- Si vous devez ajouter des phases, vous passez par un plugin, soit livré, soit que vous développez
- Ci-dessous, le plugin archetype va permettre de proposer des « modèles » tout faits dans des domaines connus (web, jar, etc), mais vous pourrez aussi créer vos archetypes à vous.



- Voilà quelques-unes des phases les plus utiles du cycle de vie Maven 2 :

- **generate-sources**: Génère le code source supplémentaire nécessaire par l'application, ce qui est généralement accompli par les plug-ins appropriés.
- **compile**: Compile le code source du projet
- **test-compile**: Compile les tests unitaires du projet
- **test**: Exécute les tests unitaires (typiquement avec Junit) dans le répertoire src/test
- **package**: Mets en forme le code compilé dans son format de diffusion (JAR, WAR, etc.)
- **integration-test**: Réalise et déploie le package si nécessaire dans un environnement dans lequel les tests d'intégration peuvent être effectués.
- **install**: Installe les produits dans l'entrepôt local, pour être utilisé comme dépendance des autres projets sur votre machine locale.
- **deploy**: Réalisé dans un environnement d'intégration ou de production, copie le produit final dans un entrepôt distant pour être partagé avec d'autres développeurs ou projets.

Les plugins

- Maven est construit sur un noyau fournissant les fonctionnalités de base pour gérer un projet, et un ensemble de plug-in qui implémentent les tâches de construction d'un projet. On peut considérer Maven comme un socle gérant une collection de plug-ins.
- En d'autres termes, les plug-ins sont là pour implémenter les actions à exécuter.
- Ils sont utilisés pour : créer des fichiers JAR, des fichiers WAR, compiler le code, créer les tests, créer la documentation du projet, et bien plus encore.
- Toute nouvelle tâche non encore présente dans la distribution des plug-ins de Maven, que l'on aimerait utiliser, peut être développée comme un plug-in.

Plug-in est une fonctionnalité importante de Maven puisqu'elle permet la réutilisation de code dans un multiple projet.

- Un plug-in peut être considéré comme une action telle la création d'un WAR que l'on déclare dans un fichier de description de projet POM.
- Le comportement d'un plug-in est paramétrable à l'aide de paramètres que l'on spécifie dans sa déclaration dans le fichier POM.

Un des plug-ins les plus simples de Maven est le plug-in *Clean*.

- Le plug-in Clean a pour fonction de supprimer les objets qui ont été créés lors du lancement du fichier POM.
- Suivant le standard Maven, ces objets sont placés sous le répertoire target.
- En lançant `mvn clean:clean` les goals du plug-in correspondant sont exécutés et les répertoires target sont supprimés.

Un des avantages des plug-in de Maven est le fait qu'ils peuvent être entièrement développés en Java, laissant ainsi l'accès à un large ensemble d'outils et composants réutilisables pour en faciliter le développement.

Les plugins

- Les principaux plugins de Maven :
 - Ant : Produit un fichier Ant pour le projet.
 - Antrun : Lance un ensemble de tâches Ant à partir d'une phase de la construction.
 - Clean : Nettoie après la construction.
 - Compiler : compile des sources Java.
 - Deploy : Construit et Déploie les artefacts dans un repository.
 - Ear : Génère un fichier EAR à partir du projet.
 - Eclipse : Génère un fichier projet Eclipse du projet courant.
 - Install : Installe les artefacts construits dans un repository.
 - Jar : Génère un fichier Jar à partir du projet.
 - Javadoc : Crée la documentation Java du projet.
 - Projecthelp : fournit des informations sur l'environnement de travail du projet.
 - Project-infos-reports : Génère un rapport standard de projet.
 - Rar : construit un RAR à partir du projet.
 - Release : Libère le projet en cours – mise à jour du POM et étiquetage dans le SCM (Source Control Management).
 - Resources : Copie les fichiers sous le répertoire "resources" dans un fichier JAR
 - Site : Génère un site pour le projet courant.
 - Source : Génère un JAR contenant les sources du projet.
 - Surefire : Exécute les jeux de test.
 - War : construit un fichier WAR du projet courant.

Exemple de lancement de tâche ANT

- Exécuter un script Ant sur une phase
- Ajouter dans le fichier pom.xml les balises suivantes :
- Exemple : associer un script Ant à la phase validate

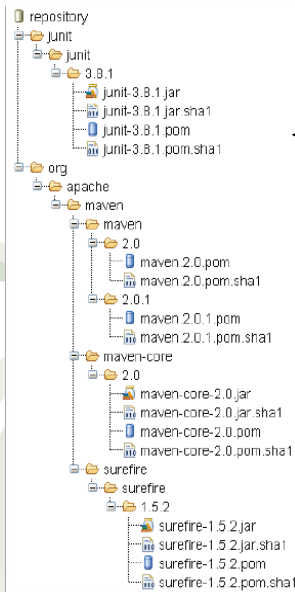
```
<project>
...
<build>
  <plugins>
    <plugin>
      <artifactId>maven-antrun-plugin</artifactId>
      <executions>
        <execution>
          <phase>validate</phase>
          <configuration>
            <tasks>
              <echo file="${basedir}/hello.txt">hello world</echo>
            </tasks>
          </configuration>
          <goals>
            <goal>run</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
...
```

- Dans une console de commandes, accéder au répertoire du projet, et exécuter la commande suivante :
mvn validate
- Le script Ant redirigeant "hello world" dans un fichier hello.txt à la racine du projet, a été traité en même temps que la phase validate

principes

- Dans l'exemple présenté, on voit une dépendance du JUNIT
 - Mais où est-elle ?
 - La réponse est dans les repositories et artefacts de MAVEN
- Repositories et artefacts dans MAVEN
 - Une dépendance est une référence à un artefact (jar, ear, etc...) dans un repository
 - MAVEN doit savoir dans quel repository il doit chercher pour assurer la dépendance
 - Dans le POM, vous ne dites pas où sont les dépendances, vous dites ce que votre projet attend
 - MAVEN prend donc la liste des dépendances du POM et les fournit à son gestionnaire de dépendances
 - On ne parle plus alors de dépendance du fichier junit-3.8.1.jar, mais de dépendance de la version 3.8.1 de l'artefact junit
 - MAVEN essaie de résoudre les dépendances en regardant dans tous les repositories disponibles.
 - Si il y a correspondance, MAVEN transporte alors cette dépendance remote sur sa repository locale, qui sera alors toujours utilisée.
 - MAVEN a donc deux types de repositories : locale et remote
 - Si la dépendance n'est pas trouvée en local, elle est cherchée en remote
 - Par défaut sur <http://www.ibiblio.org/maven2>
 - puis sur les remote repositories déclarées dans votre POM

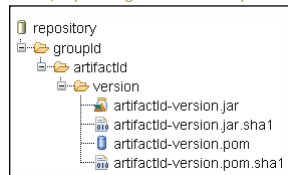
principes



■ Les repositories MAVEN

- A l'installation du produit, une repository locale est créée et remplie de dépendances par défaut.
- La repository est créée par défaut sur :

- `/.m2/repository`
 - Elle doit exister pour que MAVEN fonctionne
- Ci-contre, exemple de repository avec junit-3.8.1.jar comme dépendance
- Ci-dessous, le pattern général dans le repository :



■ Ci-dessus :

- **groupid** est le nom qualifié du domaine (a.b.c)
- **Artifacitd**

■ Ci-contre, le groupid de maven est **org.apache.maven**

principes

■ Pour satisfaire aux dépendances, MAVEN essaie d'abord de trouver l'artefact en faisant de la sorte :

□ Génération d'un path de recherche sur la repository locale :

- Ex : pour le groupid junit artifactId junit version 3.8.1, il cherchera :

○ `/.m2/repository/junit/junit/3.8.1/junit-3.8.1.jar`

§ Trouvé dans X:\documents and settings\user

- Si il n'est pas trouvé, il sera recherché dans une repository remote

○ D'abord <http://www.ibiblio.org/maven2>

○ puis les autres remote définies et dans l'ordre dans le POM ou le fichier settings.xml de la directory conf de l'installation maven

Emplacement du repository local

■ Il est possible de modifier l'emplacement du repository local

□ La valeur par défaut indiquant le repository local est
%USER_HOME%/.m2/repository

● mais elle peut être modifiée.

● Dans le fichier settings.xml, situé dans %MVN_HOME%/conf ou
%USER_HOME%/.m2/settings.xml :

○ <settings>

○ ...

○ <localRepository>chemin du répertoire</localRepository>

\$...

○ </settings>

Les dépendances

Element
« scope » de
dépendancy
Du POM
↓

Compilation

Exécution

**Fournie à la
distribution**

**Empaqueté
lors
du
déploiement**

Compile Oui

Oui

Oui

Oui

Provided Oui

Non

Non

■ La Portée des dépendances

□ Les descripteurs de projets de Maven tiennent compte de la portée des dépendances des projets.
Les portées tiennent compte des dépendances et ne sont pas nécessaires dans chaque environnement.

■ Les quatre portées disponibles sont:

□ **Compile**

● Indique que la dépendance est nécessaire pour la compilation et l'exécution. Ces dépendances seront souvent fournies dans les distributions et empaquetées dans les déploiements.

□ **Provided**

● Indique que la dépendance est nécessaire pour la compilation mais ne devrait pas être empaquetée dans les déploiements et distributions.
● Ces dépendances devraient être fournies par une source externe (typiquement un container) durant le runtime.

□ **Runtime**

● Indique qu'une dépendance n'est pas nécessaire pour la compilation, mais l'est pour le runtime.
● Souvent les dépendances de runtime sont des implémentations d'API externes et sont injectées au moment de l'exécution.
● Un exemple de dépendance d'exécution est le pilote JDBC.

□ **Test**

● Ces dépendances sont requises pour l'exécution des tests unitaires et ne sont pas rendues disponibles dans les distributions et les déploiements.

Test

Non

Non

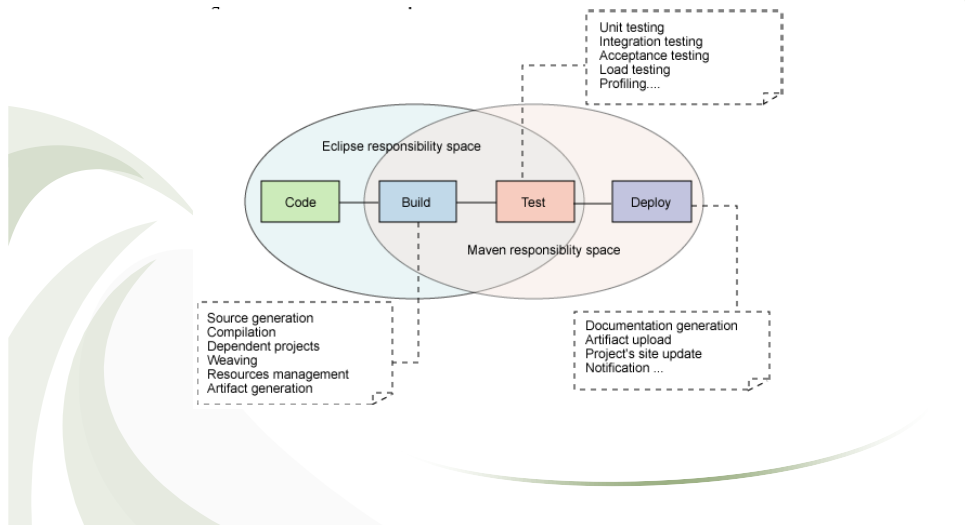
Gestion du projet

■ Gestion de Projet

- Une partie de la difficulté en installant un nouveau projet est de créer « l'infrastructure de développement ».
- En utilisant Maven, il est possible d'accélérer ce processus en générant un squelette de projet qui peut être utilisé comme modèle pour de nouvelles applications.
- Voilà quelques questions qu'il faut poser avant de commencer un nouveau projet :
 - Quels sont les attributs de mon projet (Nom, Version, Equipe...) ?
 - Quelle est la structure de fichiers de mon projet ?
 - Quels sont les composants, objets et artefacts de mon projet ?
 - Quelles sont les dépendances de mon projet ?
 - De quoi a besoin mon projet pour se construire ?
 - De quoi a besoin mon projet pour fonctionner ?
 - De quoi a besoin mon projet pour être testé ?
 - De quels plug-ins aura besoin mon projet ?
 - Quels rapports ai-je envie de générer pour mon projet en construisant mon site Web ?
- Après avoir répondu aux questions, il est possible de réaliser les tâches suivantes:
 - 1. Faire le design de la structure de fichiers du projet.
 - 2. Commencer à créer/modifier la structure du principal descripteur de projet : *pom.xml*
 - 3. Définir les dépendances du projet

Maven et Eclipse

■ Responsabilités de l'un et de l'autre

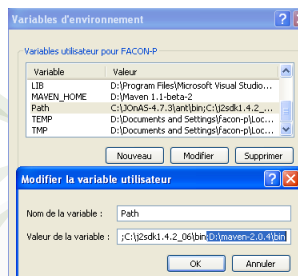


Installation de MAVEN

- Téléchargez ce fichier à cette adresse et dézippez-le.
 - Il s'installera par défaut dans C:\maven-2.0.4, mais nous y reviendrons

Adresse: <http://maven.apache.org/download.html>

- Modifiez la variable système PATH utilisateur pour lui ajouter ce path, pointez la JDK 1.5 dans JAVA_HOME



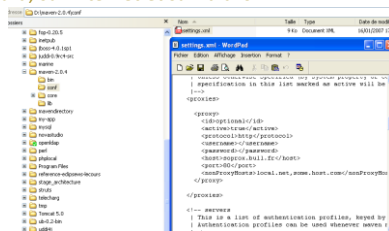
- Ajoutez le path de maven dans la variable path utilisateur
- Pour vous assurer du bon fonctionnement, tapez, dans une invite de commande :
 - C:\mvn -version

Formation MAVEN 2 : partie 2

Un premier projet Maven

Création de votre premier projet Maven

- Un fichier est important dans l'installation de Maven, il s'agit de settings.xml, qui va contenir les éléments de proxy pour pouvoir accéder au repository central, qui est, dans le cas standard, sur internet et sur ibiblio

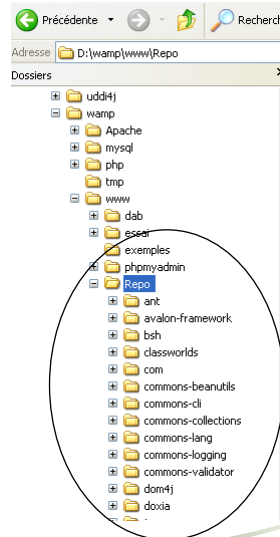


- Dans le cas où le repository central n'est pas accessible (pas de connexion, etc...), il va falloir :
 - Soit avoir dans le repository local tous les artefacts nécessaires (copie)
 - Soit avoir, par ex, un serveur Apache qui va contenir l'ensemble des artefacts nécessaires.

Cas du repository sur Internet inaccessible

- Tout d'abord installer un serveur Apache

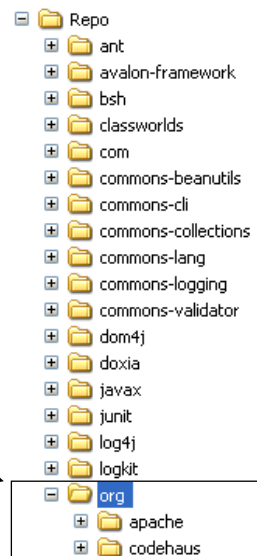
- Via Easyphp ou Wamp, par ex.
- Copier, dans la ressource partagée Apache, l'ensemble des dépendances qui, normalement, devraient être installées depuis le WEB.
 - Un zip vous sera fourni qui contient les dépendances les plus communes.
- Ci-contre, le nom de la repository alternative est Repo



Cas du repository sur Internet inaccessible

- Le repository local se trouvera, par défaut, sous Windows, dans :

- X:\Documents and Settings\user\.m2\repository
- Il est nécessaire, à cause de certaines conditions de version, d'avoir, dans la repository locale, les deux packages suivants :
- Il faut donc les copier dans la repository locale par défaut



Cas du repository sur Internet inaccessible

- Il faudra ensuite modifier le fichier settings.xml de l'installation de Maven
 - Possible de le faire aussi dans le POM.
- Particularités à appliquer :
 - Dans le cas où la repository locale est la valeur par défaut, alors rien n'est à configurer dans
 - <localRepository>
 - Laisser en commentaires
 - Rien n'est à déclarer dans <proxies>, <server>, <mirror>
 - Par contre, il faudra ajouter, en fin de fichier les éléments <repositories> et <pluginRepositories> ci-contre :
- Bien faire pointer l'entrée url vers le serveur Apache et sa ressource
- Bien utiliser le nom intenal

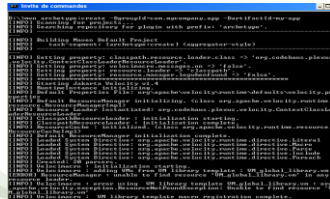
```
<profiles>
<profile>
<id>default-repositories</id>
<repositories>
<repository>
<id>internal</id>
<name>Internal Repository</name>
<url>http://localhost:80/Repo/</url>
<releases><enabled>true</enabled></releases>
<snapshots><enabled>true</enabled></snapshots>
</repository>
</repositories>
</profile>
</profiles>
<pluginRepositories>
<pluginRepository>
<id>internal</id>
<name>Internal Plugin Repository</name>
<url>http://localhost:80/Repo/</url>
<releases><enabled>true</enabled></releases>
<snapshots><enabled>true</enabled></snapshots>
</pluginRepository>
</pluginRepositories>
</profile>
</profiles>
<activeProfiles>
<activeProfile>property-overrides</activeProfile>
<activeProfile>default-repositories</activeProfile>
</activeProfiles>
</settings>
```

Création de votre premier projet Maven

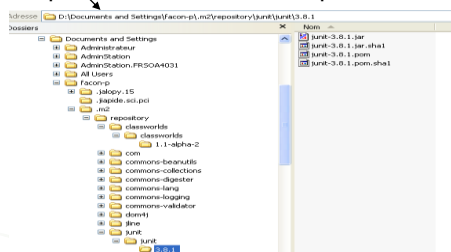
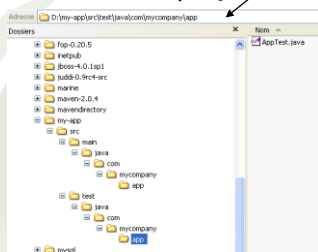
- Maven utilise la notion d'archetype.
 - Un archetype est un modèle (pattern)
 - Un archetype est constitué d'un descripteur (archetype.xml)
 - Des fichiers qui seront copiés par l'archetype
 - Du pom
 - Voir la page <http://maven.apache.org/guides/mini/guide-creating-archetypes.html>
- Il existe des archetypes déjà construits dont voici la liste :
 - [maven-archetype-archetype/](#)
 - [maven-archetype-j2ee-simple/](#)
 - [maven-archetype-mojo/](#)
 - [maven-archetype-plugin/](#)
 - [maven-archetype-plugin-site/](#)
 - [maven-archetype-pordlet/](#)
 - [maven-archetype-profiles/](#)
 - [maven-archetype-quickstart/](#)
 - [maven-archetype-simple/](#)
 - [maven-archetype-site/](#)
 - [maven-archetype-site-simple/](#)
 - [maven-archetype-webapp/](#)
 - [pom.xml](#)
- Le type d'archetype à créer sera à passer dans le sélecteur – DarchetypeArtifacld de la commande mvn

Création de votre premier projet Maven

- Pour créer votre premier projet Maven, tapez par exemple dans la ligne de commande :
 - `mvn archetype:create-DgroupId=com.mycompany.app-DartifactId=my-app`
 - Le passage du sélecteur `-DarchetypeArtifactId` est ici inutile, car nous prenons une valeur par défaut
- Suite à cette commande, les fichiers nécessaires (dépendances, fichiers) sont téléchargés selon leur présence ou non sur le site local

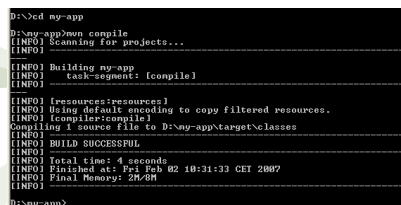


- Suite à la création de ce projet, vous vous retrouvez avec deux directories concernées :
 - L'une contenant les répertoires STANDARDS de Maven qui vont recevoir votre code
 - L'autre qui reçoit et va recevoir les dépendances dont vous aurez besoin pour Maven

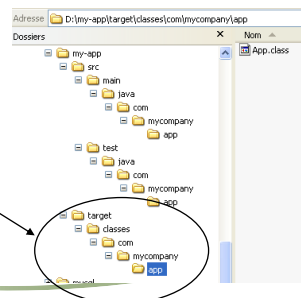


Création de votre premier projet Maven

- La compilation se fera avec la commande `mvn compile`.
 - En lançant cette commande, il faut retenir les principes suivants, qui guident tous les aspects de Maven :
 - Les conventions plutôt que les configurations
 - Réutilisation des logiques de construction
 - Exécutions déclaratives
 - Organisation cohérente des dépendances
 - D'autre part, il faut signaler le fait que moins vous modifierez les structures par défaut générées par Maven, plus facile sera la gestion des fichiers POM, et plus facile sera la transmission d'un projet vers d'autres personnes, qui retrouveront la même organisation.
 - Lancement de la compilation, étant positionné sur le répertoire préalablement créé :
 - `mvn compile`



- Résultat au niveau des directories :



Création de votre premier projet Maven

- Comment cette compilation a-t-elle pu fonctionner ?
 - Maven fonctionne par convention :
 - Les sources et les résultats de compilation seront toujours mis au même endroit (src/main/java et target/classes)
 - Ceci est décrit dans le « super POM »
 - La compilation est lancée grâce au plugin de compilation et à sa configuration par défaut
 - Ce plugin est choisi grâce à la notion de cycle de vie de construction
 - Si ce plugin n'est pas présent, il sera téléchargé comme une autre ressource
- Le lancement des tests pourra se faire avec la commande mvn test :
- Plusieurs dépendances sont maintenant téléchargées, si nécessaire
- La compilation des sources de test sans leur exécution peut se faire avec les commande suivante mvn test-compile

```
[INFO] [java:compile]
[INFO] Scanning for projects...
[INFO] Building my-app
[INFO] task:segment: (test)
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [surefire:test]
[INFO] Surefire report directory: D:\my-app\target\surefire-reports

RESULTS
Running com.mycompany.app.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.05 sec
Results: 1, Failures: 0, Errors: 0, Skipped: 0

BUILD SUCCESSFUL
[INFO] Total time: 7 seconds
[INFO] Finished at: 2007-02-05 16:13:07 CET 2007
[INFO] Final Memory: 25M/58M
[INFO]
```

Création de votre premier projet Maven

- Le packaging de votre application peut se faire avec la commande :
 - mvn package
 - Vous vous retrouvez avec un fichier jar dans la directory target, tel que demandé dans le POM
- Vous pouvez installer maintenant votre fichier jar (artefact) dans la repository locale
 - Elle pourra ainsi devenir dépendance pour d'autres projets.


```
[INFO] Total time: 4 seconds
[INFO] Finished at: 2007-02-05 16:26:06 CET 2007
[INFO] Final Memory: 25M/58M
[INFO]

D:\my-app>mvn install
[INFO] [java:install]
[INFO] Scanning for projects...
[INFO] Building my-app
[INFO] task:segment: (install)
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [surefire:test]
[INFO] Surefire report directory: D:\my-app\target\surefire-reports

RESULTS
Running com.mycompany.app.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.05 sec
Results: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] [jar:jar]
[INFO] Building jar: D:\my-app\target\my-app-1.0-SNAPSHOT.jar
[INFO] [install:install]
[INFO] Installing D:\my-app\target\my-app-1.0-SNAPSHOT.jar to D:\Documents and Settings\pascal\repository\com\mycompany\app\my-app\1.0-SNAPSHOT\my-app-1.0-SNAPSHOT.jar
[INFO] BUILD SUCCESSFUL
[INFO] Total time: 3 seconds
[INFO] Finished at: 2007-02-05 16:30:42 CET 2007
[INFO] Final Memory: 45M/58M
[INFO]
```

- La commande `mvn site` permet de créer un site internet basique de votre site



The screenshot shows the GitHub interface for the 'ny-app' repository. The 'Project Information' tab is active, displaying a message that there is currently no description associated with the project. The left sidebar shows various navigation options, and the main content area is titled 'Project Information'.

-
- The screenshot shows the Eclipse IDE with the 'Java' project selected. The 'src' folder contains the 'com.mycorp.sample' package, which includes the 'App' class. The 'test' folder contains the 'com.mycorp.sample' package, which includes the 'AppTest' class. The 'AppTest' class is the focus of the image, showing the test code for the sample application.
- ```

import junit.swingui.TestRunner;
import org.junit.Test;
import org.junit.runners.model.FrameworkField;
import org.junit.runners.model.FrameworkMethod;
import org.junit.runners.model.FrameworkField;
import org.junit.runners.model.FrameworkMethod;

/**
 * Unit Test for sample App.
 */
public class AppTest extends TestCase {

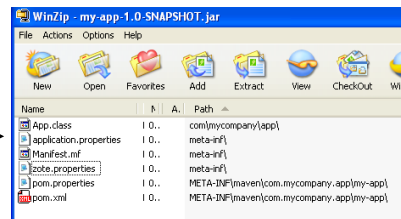
 /**
 * Create the test case
 */
 @Override protected void setUp() throws Exception {
 super.setUp();
 }

 /**
 * Maximize testName name of the test case
 */
 public AppTest(String testName) {
 super(testName);
 }
}

```

- Il est possible d'ajouter un **directory** nommé **resources** au même niveau que **main**, et y inclure tous les fichiers ressources que vous désirez
  - Ils se retrouveront **packagés** directement sous la racine du jar, exemple :

### Après mvn install ou package



- Il est bien-sûr possible de faire la même chose pour la directory test, d'y inclure le fichier resources
  - La lecture de cet éventuel fichier pouvant se faire de la sorte dans le programme AppTest.java :
    - `InputStream is=getclass().getResourcesAsStream(« /fichier.test.properties »)`



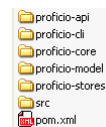
## Formation MAVEN : partie 3

### Projet multi modules

## Création d'applications avec Maven

- Nous allons maintenant regarder une application beaucoup plus lourde.
- Important : Maven valorise la notion de « Separation of concern », ou SoC
  - L'objectif est d'encapsuler des morceaux de soft d'un domaine bien identifié, d'où la notion de module, dans ce que l'on va voir maintenant.
- L'exemple que l'on va voir ici se nomme Proficio, dont l'objectif est de gérer des FAQ.

- Il est constitué de plusieurs modules :
  - Model : modèle de données
  - API l'ensemble des interfaces
  - Core : l'implémentation des API
  - Stores : module de stockage des données
  - CLI : interface ligne de commande.



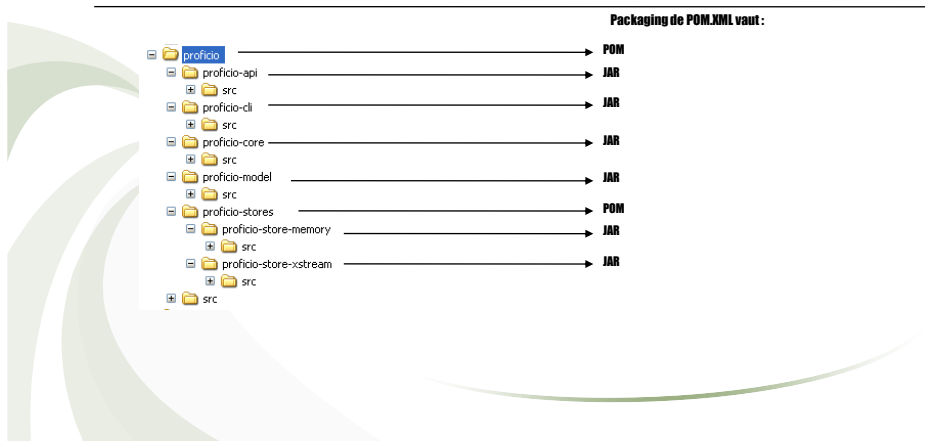
```
<project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>com.merger.mwbook</groupId>
 <artifactId>proficio</artifactId>
 <packaging>pom</packaging>
 <version>1.0-SNAPSHOT</version>
 <name>Proficio</name>
 <url>http://library.merger.com/mwbook/proficio</url>
 ...
 <modules>
 <module>proficio-model</module>
 <module>proficio-api</module>
 <module>proficio-core</module>
 <module>proficio-stores</module>
 <module>proficio-cli</module>
 </modules>
 <build>
 <sourceRoot>src</sourceRoot>
 <testSourceRoot>test</testSourceRoot>
 <plugins>
 <plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-compiler-plugin</artifactId>
 <version>3.1</version>
 <configuration>
 <source>1.8</source>
 <target>1.8</target>
 </configuration>
 </plugin>
 </plugins>
 </build>
 <distributionManagement>
 <repository>
 <id>central</id>
 <url>http://central.maven.org/maven2</url>
 </repository>
 </distributionManagement>
</project>
```

Nom physique  
des modules

- Ci-contre, quelques lignes du pom.xml principal, qui montre la référence à différents modules livrés.
- Notez aussi la version, qui sera (qui devrait) être commune à tous les modules.
- Notez la valeur de packaging à pom (et non jar, war, ect...), spécifiant à Maven qu'il doit consulter les modules inférieurs pour connaître le type de packaging désiré.

# POM ET SOUS POM

- En résumé, dès que la valeur du packaging vaut pom, cela vous permet d'insérer un ensemble de projets différents qui auront leur propre vie
- Exemple avec proficio
  - Voyez les valeurs de packaging pour chaque niveau :
    - Ouvrez la directory et observez



## Héritage des POM au niveau projet

- Une valeur donnée à un élément de POM est donnée en héritage au niveau inférieur.
  - C'est le cas par exemple de la dépendance de la librairie junit, qui n'est pas répétée aux niveaux inférieurs, mais qui est pourtant bien déclarée, car héritée.
- Il existe une commande très utile de Maven, qui permet de connaître, à chaque niveau de projet, la résultante de l'application de tous les POMs jusqu'à ce niveau.
  - Cette commande est `mvn help:effective`
    - Etant placé à un niveau donné.
  - Exemple :
    - Vous pouvez aussi créer un fichier résultat, avec le caractère >

```
D:\work\proficio\proficio-api>mvn help:effective -pl:api
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'help'.
[INFO]
[INFO] Building Proficio API
[INFO] Task-sgment: [help:effective-pom] (aggregator-style)
[INFO]
[INFO] [help:effective-pom]
[INFO]
=====
Effective POM for project 'com.norgero.nunbook:proficio:proficio-api:1.0-SNAPSHOT'
=====
<?xml version="1.0"?><project>
 <parent>
 <groupId>com.norgero.nunbook:proficio</groupId>
 <artifactId>proficio-api</artifactId>
 <version>1.0-SNAPSHOT</version>
 </parent>
 <modelVersion>4.0.0</modelVersion>
 <groupId>com.norgero.nunbook:proficio</groupId>
 <artifactId>proficio-api</artifactId>
 <name>Proficio API</name>
 <version>1.0-SNAPSHOT</version>
 <description>Proficio sample application from "Better Builds with Maven"</description>
 <url>http://library.norgero.com/nunbook/proficio/proficio-api</url>
 <scm>
 <url>http://library.norgero.com/nunbook/proficio/proficio-api</url>
 </scm>
 <dependencies>
```

## Gestion des dépendances

- Concernant les dépendances, il y a deux types d'entrée jouant un rôle au niveau des dépendances dans le POM.
  - ❑ Tout d'abord, l'entrée <dependencyManagement> n'interfère pas dans le graphe de dépendances, mais joue le rôle de préférences appliquées aux entrées (exemple la version)
  - ❑ Par contre, l'entrée <dependencies> constitue le graphe des dépendances, où l'héritage va jouer un rôle essentiel.
  - ❑ Exemple :

## POM top niveau

```
<dependencyManagement>
<dependencies>
<dependency>
<groupId>com.mergere.mvbook.proficio</groupId>
<artifactId>proficio-mode</artifactId>
<version>{project.version}</version>
</dependency>
<dependency>
<groupId>org.codehaus.plexus</groupId>
<artifactId>plexus-container-default</artifactId>
<version>1.0-alpha-9</version>
</dependency>
</dependencies>
</dependencyManagement>
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
</dependencies>
```

## POM de proficio-api

```

<project>
 <parent>
 <groupId>com.mergere.vmvbook.profilico</groupId>
 <artifactId>profilico</artifactId>
 <version>1.0-SNAPSHOT</version>
 </parent>
 <modelVersion>4.0.0</modelVersion>
 <artifactId>profilico-api</artifactId>
 <packaging>jar</packaging>
 <name>Profilico API</name>
 <dependencies>
 <dependency>
 <groupId>com.mergere.vmvbook.profilico</groupId>
 <artifactId>profilico-model</artifactId>
 </dependency>
 </dependencies>
</project>

```

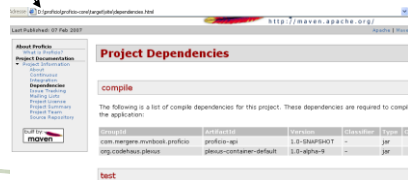
La version de profilico-model sera la prochaine version (1.0-SNAPSHOT)

**La version de proficlo-model  
sera la project.version (1.0-SNAPSHOT)  
Dépendance  
non héritée, mais préférence récupérée**

### Dépendance héritée ici

## Résolution des conflits de dépendance et utilisation des versions

- Maven facilite la gestion des dépendances grâce à la transitivité.
  - Par exemple, pouvez vous lister les JARs nécessaires à une application Hibernate ? Avec Maven 2, vous n'avez pas besoin. Vous dites juste à Maven les bibliothèques dont vous avez besoin, et Maven se charge des bibliothèques dont vos bibliothèques ont besoin (et ainsi de suite).
  - Cependant, si le graphe évolue, il se peut que deux artefacts nécessitent deux versions d'un même dépendance.
    - Maven doit choisir
      - Dans cas, il choisit la dépendance « la plus proche »
        - \$ Celle dont le nombre de dépendances à traverser avant d'y arriver est le plus petit.
    - Ce n'est pas toujours une bonne solution
      - A-t-elle les fonctionnalités requises par d'autres dépendances?
      - Si plusieurs versions sont sélectionnées au même niveau, quid n ?
  - C'est ainsi qu'il existe un moyen manuel de résoudre le conflit :
    - En retirant la version incorrecte ou en la remplaçant par une autre version.
    - Pour identifier la source de la mauvaise version, un outil existe :
      - Mvn -X -goal>
        - \$ Qui est le lancement de la cible (compile, test, package, etc...) en mode debug (-X)
      - Mvn site sur une directory donnée (ex : proficio-core) génère aussi un site dans lequel les dependances seront montrées

[illegible]

## Résolution des conflits de dépendance et utilisation des versions

### ■ Exemple pratique d'exclusion :

- Dans ce qui suit, plexus-util apparaît deux fois, et Proficio nécessite la version 1.1.
  - Voilà les deux façons de résoudre ce problème dans le pom.xml de proficio-core :
    - A gauche, avec l'exclusion
    - A droite, en forçant l'utilisation d'un version



## Résolution des conflits de dépendance et utilisation des versions

- Il y a une autre possibilité de recherche de version de dépendances qui existe dans Maven, et qui est à conseiller :

- C'est la recherche « la plus proche », utilisable avec des patterns, tels que ceci :

```

<dependency>
 <groupId>org.codehaus.plexus</groupId>
 <artifactId>plexus-utils</artifactId>
 <version>[1.1,)</version>
</dependency>

```

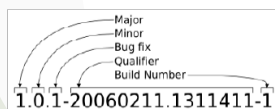
- Ici, la version 1.1 est préférée si elle est trouvée

- Mais toute version supérieure à 1.1 sera recherchée dans la repository si nécessaire

- Ci-dessous, des exemples de plages applicables

(,1.0]	inférieur ou égal à 1
[1.2,1.3]	Entre 1.2 et 1.3 (inclusif)
[1.0,2.0)	Supérieur ou égal à 1.0, mais inférieur à 2.0
[1.5,)	Supérieur ou égal à 1.5
(,1.1),(1.1,)	Toute version, excepté 1.1

- Signification de chaque position :



- Par défaut, la repository est contrôlée une fois par jour pour MAJ des versions des artefacts utilisés, mais ceci peut être configuré dans le POM avec :

```

<repository>
 ...
 <releases>
 <updatePolicy>interval:60</updatePolicy>
 </releases>
</repository>

```

# Dépendances

- Un autre rapport de la commande `mvn site` permet de montrer, par dépendance, les projets maven qui l'utilisent

- Lancer `mvn site` sur proficio (home directory)

- Et regarder la page `dependency-convergence.html`

Apache Maven

### Reactor Dependency Convergence

Legend:

- All projects share one version of the dependency.
- At least one project has a differing version of the dependency.

Statistics	
Number of sub-projects:	8
Number of dependencies (NOD):	8
Number of unique artifacts (NOA):	8
Number of SNAPSHOT artifacts (NOIS):	5
Convergence (NOA/NOIS):	100%

Ready for database (100% Convergence and no SNAPSHOTs):

com.mergere.mvnbook:proficio:proficio-store-memory

1.0-SNAPSHOT	a. com.mergere.mvnbook:proficio:proficio-cli b. com.mergere.mvnbook:proficio:proficio-store-xstream
--------------	--------------------------------------------------------------------------------------------------------

com.mergere.mvnbook:proficio:proficio-store-xstream

1.0-SNAPSHOT	a. com.mergere.mvnbook:proficio:proficio-cli
--------------	----------------------------------------------

JUnit:junit

3.8.1	a. com.mergere.mvnbook:proficio:proficio-api b. com.mergere.mvnbook:proficio:proficio-cli c. com.mergere.mvnbook:proficio:proficio-core d. com.mergere.mvnbook:proficio:proficio-model e. com.mergere.mvnbook:proficio:proficio-store-memory f. com.mergere.mvnbook:proficio:proficio-store-xstream g. com.mergere.mvnbook:proficio:proficio-stores h. com.mergere.mvnbook:proficio:proficio
-------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

# Dépendances

- Une fois que vous avez identifié le path de la version, vous pouvez l'exclure de la dépendance en ajoutant une exclusion

- Ex :

```
...
<dependency>
 <groupId>org.codehaus.plexus</groupId>
 <artifactId>plexus-container-default</artifactId>
 <version>1.0-alpha-9</version>
 <exclusions>
 <exclusion>
 <groupId>org.codehaus.plexus</groupId>
 <artifactId>plexus-utils</artifactId>
 </exclusion>
 </exclusions>
</dependency>
...
```

- Ou alors vous pouvez forcer une certaine version

```
...
<dependencies>
 <dependency>
 <groupId>org.codehaus.plexus</groupId>
 <artifactId>plexus-utils</artifactId>
 <version>1.1</version>
 <scope>runtime</scope>
 </dependency>
</dependencies>
...
```



# Les profiles

- Les profiles permettent de créer des variations dans le cycle de construction, afin de s'adapter à des particularités, comme
  - des construction pour des plateformes différentes
  - Tester sur différentes DB
  - Référencer un Système de fichiers local
  - Etc...
- Les profiles sont déclarés dans des entrées du POM
  - Ils peuvent être « activés » de différentes manières
  - Ils modifient le POM au moment du build, prenant en considération les paramètres passés
- Les profiles pourront être définis à trois endroits :
  - Le fichier settings.xml de la directory .m2 (repository)
  - Un fichier nommé profiles.xml dans la même directory que le POM
  - Le POM lui-même
- Dans un de ces fichiers, vous pouvez définir les éléments suivants :
  - Repositories, pluginRepositories, dependencies, plugins,modules, reporting
  - dependencyManagement, distributionManagement
- Vous pouvez activer un profile avec
  - le sélecteur -P de la commande mvn
    - qui prendra en argument la liste des ids de profile que vous voulez activer, ex :
      - `Mvn -Pprofile1, profile2 install`
  - Via la section activeProfile, qui référence les profiles définis dans settings.xml, par ex (voir ci-dessous à droite)
  - Via un « déclencheur » ou trigger, qui, si vérifié, activera le profile déclaré
    - Ci-dessous à gauche, le profile1 est activé si la variable système nommé environnement vaut test
      - On peut déclencher sur la valeurs de propriétés systèmes (debug par ex), des propriétés systèmes à des valeurs particulières

```
<profile>
 <id>profile1</id>
 ...
 <activation>
 <property>
 <name>environnement</name>
 <value>test</value>
 </property>
 </activation>
</profile>
```

```
<settings>
 ...
 <profiles>
 <profile>
 <id>profile1</id>
 ...
 </profile>
 </profiles>
 <activeProfiles>
 <activeProfile>profile1</activeProfile>
 </activeProfiles>
 ...
</settings>
```

# Les profiles

- Observez les profiles du POM de profico-cli

```
<profiles>
 <!-- Profile which creates an assembly using the memory based store -->
 <profile>
 <id>memory</id>
 <build>
 <plugins>
 <plugin>
 <artifactId>mvn-assembly-plugin</artifactId>
 <configuration>
 <descriptor>
 <descriptor>src/main/assembly/assembly-store-memory.xml</descriptor>
 </descriptor>
 </configuration>
 </plugin>
 </plugins>
 </build>
 <activation>
 <property>
 <name>memory</name>
 </property>
 </activation>
 </profile>
```

```
<!-- Profile which creates an assembly using the xstream based store -->
<profile>
 <id>xstream</id>
 <build>
 <plugins>
 <plugin>
 <artifactId>mvn-assembly-plugin</artifactId>
 <configuration>
 <descriptor>
 <descriptor>src/main/assembly/assembly-store-xstream.xml</descriptor>
 </descriptor>
 </configuration>
 </plugin>
 </plugins>
 </build>
 <activation>
 <property>
 <name>xstream</name>
 </property>
 </activation>
 <activation>
 <property>
 <name>xstream</name>
 </property>
 </activation>
 <profile>
```

- Deux profiles existent , l'un avec memory et l'autre avec xstream

- Chacun de ces profiles pointe un descripteur permettant de créer une version adaptée.
  - Observez aussi le déclenchement par la présence d'une propriété système
- Le lancement dans une configuration donnée se fera par :
  - `mvn -Dmemory clean assembly:assembly`
  - `mvn -Dxstream clean assembly:assembly`

## Déploiement de l'application

- Le déploiement (partage) peut se faire via plusieurs méthodes telles que :
  - ❑ Copie de fichier, via ssh2, sftp, ftp ou ssh externe
  - ❑ Vous devez dans ce cas configurer l'entrée distributionManament du POM (le plus élevé dans la hiérarchie, afin que les POM enfants suivent)
  - ❑ Exemple de configuration par Système de fichier (gauche) et par ftp :

```
<project>
...
<distributionManagement>
 <repository>
 <id>proficio-repository</id>
 <name>Proficio Repository</name>
 <url>file://${basedir}/target/deplo</url>
 </repository>
</distributionManagement>
...
</project>
```

```
<project>
...
<distributionManagement>
 <repository>
 <id>proficio-repository</id>
 <name>Proficio Repository</name>
 <url>http://ftpserver.yourcompany.com/deploys/</url>
 </repository>
</distributionManagement>
<build>
 <extensions>
 <extension>
 <groupId>org.apache.maven.wagon</groupId>
 <artifactId>wagon-ftp</artifactId>
 <version>1.0-alpha-6</version>
 </extension>
 </extensions>
</build>
...
</project>
```

- La commande déclenchant le déploiement est :
  - mvn deploy

# Création du site web pour votre application

- **Enfin, le site de votre bannière.** En site standard, vous pouvez constituer un site en insérant, dans la directory site de src, les différents formats de documentation, ainsi que le descripteur de votre site.
  - ☐ Les formats disponibles sont :
    - XDOC (format XML très utilisé par Apache)
    - APT (Almost plain text), utilisé par les wiki et très simple
    - FML, utilisé par les FAQ
    - DocBook, plus complexe
- **Un point important est le fichier de description site.xml, qui va configurer :**
  - ☐ L'apparence de la bannière
  - ☐ La texture
  - ☐ Le format de la date
  - ☐ Les liens sous la bannière
  - ☐ Des informations supplémentaires pour inclure dans la balise <HEAD>
  - ☐ Les menus dans la colonne de navigation
  - ☐ L'apparence des rapports
- **Exemple pour ce projet**
  - ☐ Et à droite, après la commande mvn site

[illegible]

# Création du site web pour votre application

- Un des avantages majeurs de Maven est génération de rapports.
- Par défaut, les rapports suivants sont générés :
  - Dépendances, mailing list, intégration continue, repository source, suivi de problèmes, Equipe projet, licence.
- Vous pouvez configurer dans le POM.XML principal l'entrée <reporting> afin de configurer les rapports d'édition
  - Puis lancer la commande mvn site pour obtenir les rapports
    - Voyez dans target/site le résultat de la génération de site (index.html)

```
<reporting>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-project-info-reports-plugin</artifactId>
<reportSet>
<reportSet>
<reports>
<report>dependencies</report>
<report>project-team</report>
</reports>
</reportSet>
</reportSet>
</plugins>
</reporting>
</project>
```

Autre exemple

```
...
<reporting>
...
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-project-info-reports-plugin</artifactId>
<reportSet>
<reportSet>
<reports>
<report>dependencies</report>
<report>project-team</report>
<report>mailing-list</report>
<report>ci</report>
<!--
Issue tracking report will be omitted
<report>issue-tracking</report>
-->
<report>license</report>
<report>team</report>
</reports>
</reportSet>
</reportSet>
</plugins>
</reporting>
...
</reporting>
```

## Formation MAVEN : partie 4

### Rapports

# Tenue du projet

- Grâce à la structure déclarative du POM, Maven a accès aux informations constituant le projet, et, en utilisant différents outils, peut analyser, rapporter, afficher des information
  - L'intégration des outils est facilitée grâce à cet aspect déclaratif du POM
- Le site Proficio complémentaire (ch 6) intègre de nouveaux rapports que l'on peut voir ci-dessous
  - Ces rapports sont déclenchés par les déclarations ajoutées dans le POM.XML, et que mvn site génère

The screenshot shows the Apache Maven Project website with a section titled 'Maven Generated Reports'. It lists various reports generated by Maven, such as 'Overview', 'Dependencies', 'Code Coverage', etc. To the right, a snippet of POM XML configuration is shown, highlighting the configuration for the `maven-surefire-report-plugin`.

```

<reporting>
 <plugins>
 <plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-surefire-report-plugin</artifactId>
 <configuration>
 <showSuccess>false</showSuccess>
 </configuration>
 </plugin>
 </plugins>
</reporting>

```

- Il est également possible de ne lancer qu'un report, avec le type de commande suivante :
  - `mvn surefire-report:report`
    - Etant positionné sur un directory donné, par ex, profico-core

# Configuration des reports

- L'ajout et la configuration d'un report se fait comme dans le cas d'ajout de plugins, mais au lieu que l'ajout se fasse dans l'entrée `<build>` puis `<plugins>`, elle se fait dans l'entrée `<reporting>`
  - La configuration est simple, par exemple, ci-dessous, on demande d'avoir le report uniquement dans le cas d'erreurs :

```

<reporting>
 <plugins>
 <plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-surefire-report-plugin</artifactId>
 <configuration>
 <showSuccess>false</showSuccess>
 </configuration>
 </plugin>
 </plugins>
</reporting>

```

- L'ajout de ce plugin déclenche l'inclusion du rapport sur le site WEB, et l'entrée configuration permet de changer l'apparence de la sortie.
- On peut être amené à déclarer des plugins de report dans l'entrée `build`, par exemple dans le cas où l'on voudrait générer le rapport HTML aussi dans la directory de build (target), à chaque build, et pas seulement dans site, on pourrait ajouter ainsi dans l'entrée `build` :
  - Hormis ce cas, la configuration des plugins de report se fait uniquement dans

```

<build>
 <plugins>
 <plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-surefire-report-plugin</artifactId>
 <configuration>
 <outputDirectory>
 ${project.build.directory}/surefire-reports
 </outputDirectory>
 </configuration>
 </plugin>
 </plugins>
</build>

```

# Configuration des reports

- Il est possible de demander plusieurs reports d'un même plugin.
  - Dans ce cas, on utilisera l'élément <reportsets>, qui permettra l'exécution de plusieurs reports différents.

- Exemple :

```
...
<reporting>
 <plugins>
 <plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-surefire-report-plugin</artifactId>
 <reportSets>
 <reportSet>
 <id>unit</id>
 <configuration>
 <reportDirectory>
 $project.build.directory/surefire-reports/unit
 </reportDirectory>
 <outputName>surefire-report-unit</outputName>
 </configuration>
 <reports>
 <report>report</report>
 </reports>
 </reportSet>
 <reportSet>
 <id>perf</id>
 <configuration>
 <reportDirectory>
 $project.build.directory/surefire-reports/perf
 </reportDirectory>
 <outputName>surefire-report-perf</outputName>
 </configuration>
 <reports>
 <report>report</report>
 </reports>
 </reportSet>
 </reportSets>
 </plugin>
 </plugins>
</reporting>
```

- Le lancement de la commande mvn site générera ainsi deux fichiers de génération :
  - surefire-report-perf.html
  - Et
  - surefire-report-unit.html
- Il est possible de séparer les reports pour les développeurs et les utilisateurs
  - Voir l'entrée 6.4 de « better builds with maven », qui réutilise la notion d'archétype

# Configuration des reports

- Voici une liste des rapports que l'on peut générer avec Maven :

Report	Description	Visual	Check	Notes
Javadoc	Produces an API reference from Javadoc.	Yes	N/A	✓ Useful for most Java software ✓ Important for any projects publishing a public API
JDK	Produces a source cross reference for any Java code.	Yes	N/A	✓ Companion to Javadoc that shows the source code ✓ Important to include when using other reports that can refer to it, such as Checkstyle ⚠ Doesn't handle JDK 5.0 features
Checkstyle	Checks your source code against a standard descriptor for formatting issues.	Yes	Yes	✓ Use to enforce a standard code style ✓ Recommended to enhance readability of the code ⚠ Not useful if there are a lot of errors to be fixed – it will be slow and the result unhelpful.
PMD	Checks your source code against known rules for code smells.	Yes	Yes	✓ Should be used to improve readability and identify simple and common bugs. ✓ Some overlap with Checkstyle rules.
CPD	Part of PMD checks for duplicate source code blocks that indicates it was copy/pasted.	Yes	No	✓ Can be used to identify lazy copy/pasted code that might be refactored into a shared method. ✓ Avoids issues when one piece of code is fixed/updated and the other forgotten.
Tag List	Simple report on outstanding todo's or other markers in source code	Yes	No	✓ Useful for tracking TODO items ✓ Very simple, convenient set up ✓ Can be implemented using Checkstyle rules instead.
Cobertura	Analyze code statement coverage during unit tests or other code execution.	Yes	Yes	✓ Recommended for teams with a focus on tests ✓ Can help identify untested or even unused code. ⚠ Doesn't identify all missing or inadequate tests, so additional tools may be required.
Surefire Report	Show the results of unit tests visually.	Yes	Yes	✓ Recommended for easier browsing of results. ✓ Can also show any tests that are long running and slowing the build. ✓ Check already performed by surefire.test.
Dependency Convergence	Examine the state of dependencies in a multiple module build	Yes	No	✓ Recommended for multiple module builds where consistent versions are important. ✓ Can help find suspicious prior to release.
Clirr	Compare two versions of a JAR for binary compatibility	Yes	Yes	✓ Recommended for libraries and frameworks with a public API ✓ Also important for reviewing changes to the internal structure of a project that are still exposed publicly.
Changes	Produce release notes and road maps from issue tracking systems	Yes	N/A	✓ Recommended for all publicly released projects. ✓ Should be used for keeping teams up to date on external projects also.

# Configuration des reports

- L'outil JXR gère affiche les cross-références
  - Etant dans proficio-core, lancez `mvn jxr:jxr`, et observez le fichier `jxr.html` généré dans proficio-core, qui pointe vers les sources main et test
  - L'inclusion du plugin `jxr` dans le `pom.xml` permet la génération automatique des reports `jxr` sur lancement de la commande `mvn` site

**All Classes**

**Packages**

`com.marcos.merobooks.profiles`

---

**Classes**

`ProficioBuilder`

**View Javadoc**

```

package com.marcos.merobooks.profiles;

/**
 * Copyright 2012-2013 The Apache Software Foundation.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

import com.marcos.merobooks.profiles.model.Profile;

import java.io.IOException;
import java.io.OutputStream;

/**
 * The default implementation of Profile.
 *
 * @author Pili in the gaps
 */
public class ProficioBuilder implements Profile {

 /**
 * Number of lines to skip in the log.
 */
 private static final int SKIP = 10;

 /**
 * Get the log.
 */
 private String log;

 private ProficioBuilder() {}

 /**
 * Get a log entry.
 */

```

```

<reporting>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-jxr-plugin</artifactId>
</plugin>
...
</plugins>
</reporting>

```

- L'utilisation de Javadoc est également possible
  - Lancer `mvn javadoc:javadoc` depuis proficio-core, par ex
  - On pourrait également ajouter le plugin `javadoc` dans le `pom.xml` général :
    - Le report `javadoc` est très configurable, comme la commande `javadoc java`

```

...
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-javadoc-plugin</artifactId>
</plugin>
...

```

# Configuration des reports

- Un élément intéressant est l'entrée `links` de la configuration du plugin `javadoc`.
  - Elle permet de référencer les sites, ci-dessous Sun et Plexus dans le cas de classes s'y référant.
    - Ex : les références aux classes `java.lang.String` dans les pages du site pointeront automatiquement sur le site de Sun :

```

...
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-javadoc-plugin</artifactId>
<configuration>
<links>
<link>http://java.sun.com/j2se/1.4.2/docs/api</link>
<link>http://plexus.codehaus.org/ref/1.0-alpha-9/apidocs</link>
</links>
</configuration>
</plugin>
...

```

- Le Plugin `PMD` permet autorise aussi une configuration
  - Vous pouvez définir vos règles dans le `POM`, ou alors faire référence à un fichier contenant les spécificités de règle :

**SOIT**

```

...
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-pmd-plugin</artifactId>
<configuration>
<rulesets>
<ruleset>rulesets/basic.xml</ruleset>
<ruleset>rulesets/imports.xml</ruleset>
<ruleset>rulesets/unusedcode.xml</ruleset>
<ruleset>rulesets/finalisers.xml</ruleset>
</rulesets>
</configuration>
</plugin>
...

```

**SOIT**

```

...
<reporting>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-pmd-plugin</artifactId>
<configuration>
<ruleset>rulesets/arc/main/pmd/custom.xml</ruleset>
</ruleset>
</configuration>
</plugin>
</plugins>
</reporting>
...

```

**ET**

```

<?xml version="1.0"?>
<ruleset name="Custom">
<description>
Default rules, no unused private field warning
</description>
<rule ref="rulesets/basic.xml" />
<rule ref="rulesets/imports.xml" />
<rule ref="rulesets/unusedcode.xml" />
<exclude name="UnusedPrivateField" />
</rule>
</ruleset>

```

## Configuration des reports

- Vous pouvez demander le contrôle (check) de PMD lors de la phase de build :

```
<build>
<plugins>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-pmd-plugin</artifactId>
<executions>
<execution>
<goals>
<goal>check</goal>
</goals>
</execution>
</executions>
</plugin>
...
</plugins>
</build>
```

- Le contrôle peut se faire avec la commande
  - mvn pmd:check sur proficio-core
- Après avoir identifié les erreurs (dans le fichier DefaultProficio.java), vous pouvez demander à PMD de les éviter :
  - // NOPMD sur la ligne qui a provoqué l'erreur
- mvn verify pour contrôler
- Ce type de contrôle (couteux)  
Pourrait parfaitement s'inscrire  
Dans un profile

```
...
// Trigger PMD and checkstyle
int i: // NOPMD
...
int j: // NOPMD
...
private void testMethod() // NOPMD
{
}
...
```

## Configuration des reports

- Un autre plugin est checkstyle, permettant de contrôler les normes au code.
  - Lancer mvn checkstyle:checkstyle sur proficio-core
    - Plusieurs erreurs apparaissent car les conventions adoptées dans le source sont celles de maven, tandis que le plugin checkstyle est configuré sur les normes Sun par défaut.

Summary			
Files	Infos	Warnings	Errors
2	0	0	107

Files			
Files	I	W	E
com/mergere/mvnbook/proficio/DefaultProficio.java	0	0	107

Details		
com/mergere/mvnbook/proficio/DefaultProficio.java		
Violation	Message	Line
	Line has trailing spaces.	30
	'/' should be on the previous line.	30
	Missing a javadoc comment.	37
	Method 'addEntry' is not designed for extension - needs to be abstract, final or empty.	43
	'/' is followed by whitespace.	43
	Parameter entry should be final.	43

## Configuration des reports

- Ajouter cette configuration dans le pom général :

```
...
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-checkstyle-plugin</artifactId>
<configuration>
<configLocation>config/maven_checks.xml</configLocation>
</configuration>
</plugin>
```

- Les configurations autorisées (construites dans le plugin checkstyle) sont les suivantes :

Configuration	Description	Reference
config/sun_checks.xml	Sun Java Coding Conventions	<a href="http://java.sun.com/docs/codeconv/">http://java.sun.com/docs/codeconv/</a>
config/maven_checks.xml	Maven team's coding conventions	<a href="http://maven.apache.org/guides/development/guide/m2-development.html#code-style">http://maven.apache.org/guides/development/guide/m2-development.html#code-style</a>
config/turbine_checks.xml	Conventions from the Jakarta Turbine project	<a href="http://jakarta.apache.org/turbine/common/code-standards.html">http://jakarta.apache.org/turbine/common/code-standards.html</a>
config/avalon_checks.xml	Conventions from the Apache Avalon project	No longer online - the Avalon project has closed. These checks are for backwards compatibility only.

- Vous pouvez être amené à créer votre propre configuration
  - Voir <http://checkstyle.sf.net/config.html>
- Les éléments concernant la documentation du plugin checkstyle peuvent être trouvés sur
  - <http://maven.apache.org/plugins/maven-checkstyle-plugin/tips.html>

## Configuration des reports

- Un petit plugin, nommé tag list, permet simplement de récupérer les lignes où sont déclarées les tags
  - TODO et @todo
- Vous pouvez ajouter de nouveaux patterns dans le POM général
  - Le report vous sortira toutes les lignes où ce pattern est trouvé

```
...
<plugin>
<groupId>org.codehaus.mojo</groupId>
<artifactId>taglist-maven-plugin</artifactId>
<configuration>
<tags>
<tag>TODO</tag>
<tag>@todo</tag>
<tag>FIXME</tag>
<tag>XXX</tag>
</tags>
</configuration>
</plugin>
...
```



## Autres plugins

- D'autres plugins existent tels que
  - Cobertura
  - Clirr
  - Junit
- Vous trouverez des exemples à partir du chapitre 6.8 dans « better builds with maven » et sur internet

## Formation Maven : partie 5

### Maven 3

## Limites de Maven 2

Maven 2, qui existe depuis 2005, connait les limites suivantes.

- Grande complexité de son noyau. Difficile de se l'approprier pour faire évoluer
  - Ex : moteur IoC 'Plexus'
- pom.xml verbeux
  - Exemple pour la définition de dépendances
- Trop grande utilisation de plugin, y compris pour réaliser tâches simples
  - Ex : demander de compiler en java 5
  - Ex : l'écriture d'une seule dépendance nécessite au moins 5 lignes !
- Documentation officielle peu fournie et peu intuitive
- Support pas optimal des environnements de développement (IDE)
- Développement et support des plugins 'officiels' inégaux
- Manque de souplesse sur certains principes (Ex : cycle de vie)

## Maven 3

Maven 3, qui existe depuis 2010, a les caractéristiques suivantes.

- Réécriture du noyau, basé sur conteneur IoC Google Guice /
- Possibilité d'écrire des POM dans autres langages (ex : YAML, GROOVY)
- Possibilité de créer des plugin par héritage
  - Évite les copier/coller !
- Composition de POM (comme les 'fragments' web.xml de servlet 3.0)
- Notion de 'plan de build' : un plugin peut connaître opérations avant / après lui
- Développement et support des plugins 'officiels' inégaux
- Manque de souplesse sur certains principes (Ex : cycle de vie)