INTRODUCTION A MAVEN

OBJET :	Aide au developpement
RÉFÉRENCE :	Maven, eclipse, maven for eclipse, plugin maven
ÉQUIPE :	Méthode
AUTEUR(S) :	POITEVINEAU Romain
DESTINATAIRE(S) :	Developpeurs
VERSION / DATE :	1.1, le 18/01/2012
STATUT :	REALEASE CANDIDATE
VALIDÉ PAR :	Developpeurs

Remarque

Cette documentation est faite pour apporter une introduction a maven, ces et principes de fonctionnements et pourquoi nous allons l'utiliser dans le projet .

En cas de probléme rencontrer ne figurant pas dans la dans la documentation merci de transmettre le probléme à l'équipe méthode. Merci également d'en faire parvenir la solution si vous l'avez trouvé.

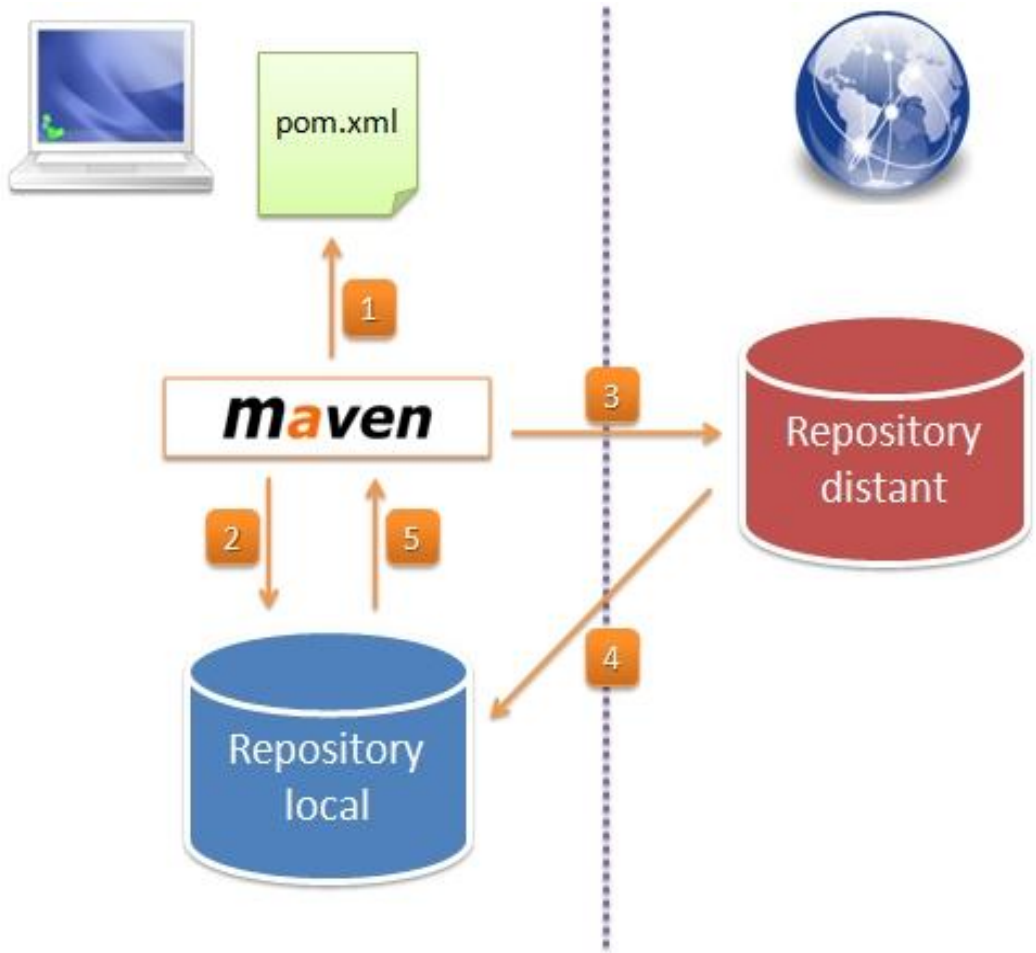
Sommaire

I – Introduction

Maven était une branche de l'organisation Jakarta Project avant de devenir un projet sous licence Apache.

Cet outils permet de gérer l'automatisation de production des projets logiciels Java/J2ee. Autrement dit, il va permettre de produire un logiciel à partir de ses sources, en optimisant les tâches réalisées tout en assurant le bon ordre de fabrication du projet.

Maven peut être comparé à l'outil Ant, il possède un fichier de paramétrage (.pom ou Project Object Model de la forme d'un .xml) qui va décrire le projet, ses dépendances avec des modules externalisés ainsi que son ordre d'execution.



II – Le ficher POM.xml

Le POM contient une description détaillée de votre projet, avec en particulier des informations concernant le versionnage et la gestion des configurations, les dépendances, les ressources de l'application, les tests, les membres de l'équipe, la structure et bien plus encore. Le POM prend la forme d'un fichier XML (*pom.xml*) qui est placé dans le répertoire de base de votre projet. Un fichier pom.xml est présenté ci-dessous :

```
<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.javaworld.hotels</groupId>
   <artifactId>HotelDatabase</artifactId>
   <packaging>war</packaging>
   <version>1.0-SNAPSHOT</version>
   <name>Base de données des hotels</name>
   <url>http://www.hoteldatabase.com</url>
   <dependencies>
      <dependency>
         <groupId>junit</groupId>
         <artifactId>junit</artifactId>
         <version>3.8.1</version>
         <scope>test</scope>
      </dependency>
   </dependencies>
</project>
```

III – Maven, Un standard du developpement java

Une partie de la puissance de Maven vient des pratiques standardisées qu'il encourage. Un développeur qui a déjà travaillé sur un projet Maven se sentira tout de suite familier avec la structure et l'organisation d'un autre projet Maven.

il est réellement intéressant de respecter la structure de répertoire standard de Maven 2 autant que possible, et ce pour plusieurs raisons :

- Cela rend le fichier POM plus court et plus simple
- Cela rend le projet plus simple à comprendre et rend la vie plus simple au pauvre développeur qui devra maintenir le projet quand vous partirez.
- Cela rend l'intégration de plug-ins plus simple

Maven va être utile sur les points suivants pour le projet UmlArchitect :

- Il va faciliter le processus de construction du projet

- Il va fournir un système de construction uniforme

- Il va fournir des informations projets de qualité (description détaillée en GroupeID/Artifacts/Version pour déterminer la version précise du projet)

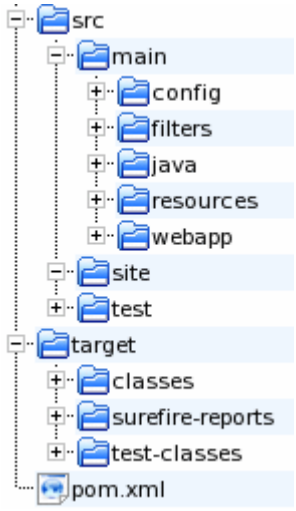
- Il va définir une ligne directive pour les bonnes pratiques de programmation

- Il va permettre l'ajout de nouvelle fonctionnalité de manière transparente et facile à implémenter.

Cet outils nous permet donc d'être performant pour le développement de la solution, suivant des normes et règles de programmation.

Ces derniers avantages apportent une aide au développement d'un projet sur lequel 11 développeurs se penchent.

IV - Architecture



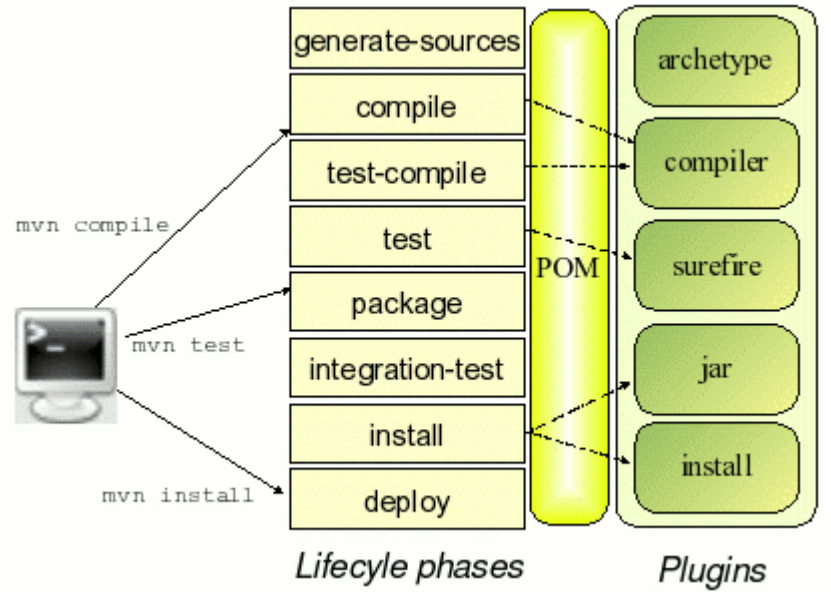
Le répertoire src contient plusieurs sous-repertoires, chacun avec une utilité précise :

- **src/main/java:** Votre code java va ici
- **src/main/resources:** Les autres ressources dont votre application a besoin
- **src/main/filters:** Les filtres de ressources, sous forme de fichier de propriétés, qui peuvent être utilisés pour définir des variables connues uniquement au moment du build.
- **src/main/config:** Les fichiers de configuration
- **src/main/webapp:** Le répertoire d'application web pour les projets WAR.
- **src/test/java:** Les tests unitaires
- **src/test/resources:** Les ressources nécessaires aux tests unitaires, qui ne seront pas déployées
- **src/test/filters:** Les filtres nécessaires aux tests unitaires, qui ne seront pas déployées
- **src/site**: Les fichiers utilisés pour générer le site web du projet Maven

V – Notions de cycle de vie

Les cycles de vie des projets sont un concept central de Maven 2.

Dans Maven 2, cette notion est standardisée dans un groupe de phases de cycle de vie bien définies et déterminées.



Au lieu de lancer des plug-ins, dans Maven 2, le développeur lance l'action liée à une phase du cycle de vie :

```
mvn compile
```

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

- *generate-sources*: Génère le code source supplémentaire nécessité 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.

Beaucoup d'autres phases du cycle de vie sont disponibles voir :

http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

VI – Dependances transitives

Une des nouveautés de Maven 2 est la gestion des dépendances transitives.

Vous dites juste à Maven les bibliothèques dont vous avez besoin, et Maven se charge des bibliothèques dont vos bibliothèques ont besoin.

Supposons que vous voulez utiliser Hibernate dans votre projet. Vous auriez simplement à ajouter une nouvelle dépendance à la section dependencies de votre pom.xml, comme suit :

```
<dependency>
    <groupId>hibernate</groupId>
    <artifactId>hibernate</artifactId>
    <version>3.0.3</version>
    <scope>compile</scope>
</dependency>
```

VII – Petit Manuel pour débuter

Démarrer un projet

Pour démarrer un projet, Maven2 propose en standard un certain nombre d'"Archetypes" (templates) d'applications.

L'archetype par défaut permet de produire une bibliothèque de classes Java (.jar) :

```
mvn archetype:create -DgroupId=esgi.gl.al -DartifactId=MonAppli
```

Cette commande génère l'arborescence du code source et crée un fichier pom.xml de description du projet dont l'entête est :

```
    <groupId>esgi.gl.al</groupId>
    <artifactId>MonAppli</artifactId>
    <packaging>jar</packaging>
    <version>1.0</version>
```

Dans cet exemple, on crée une librairie de classes Java .jar, appelée “MonAppli” (artifact), dans le groupe esgi.gl.al. Ces noms sont arbitraires et permettent d'organiser les projets.

Un projet Maven est conçu pour être lui-même réutilisé dans d'autres projets Maven, au même titre que les librairies standard dont on déclare les dépendances. Ainsi, un autre projet utilisant les classes développées dans le projet “MonAppli” pourra tout à fait déclarer la dépendance suivante :

```
   <dependency>
          <groupId>esgi.gl.al</groupId>
          <artifactId>MonAppli</artifactId>
          <version>1.0</version>
     </dependency>
```

Un autre archetype d'applications intéressant est “webapp” :

```
mvn archetype:create -DgroupId=esgi.gl.al -DartifactId=MaWebapp -
Dpackagename= esgi.gl.al -DarchetypeArtifactId=maven-archetype-webapp
```

Dans ce cas, le pom.xml déclare ceci :

```
      <groupId>esgi.gl.al</groupId>
      <artifactId>MaWebapp</artifactId>
      <packaging>war</packaging>
      <version>1.0-SNAPSHOT</version>
      <name>Maven Webapp Archetype</name>
```

Noter le tag <packaging>war</packaging> qui définit le produit final de la compilation : le fichier MaWebApp.war, prêt à être déployé.

Compiler, tester, installer

Une fois le projet créé, il faut se placer dans le répertoire racine de l'application (au même niveau que le fichier pom.xml) pour executer toutes les autres commandes mvn. En voici quelques unes très utiles :

mvn compile : compilation du code source. Le code source se trouve dans le répertoire src/main/java et se retrouve compilé dans le répertoire target, où on retrouve l'arborescence de packages

mvn test : execution des tests unitaires, localisés dans le répertoire src/test/java

mvn clean install : purge du répertoire target, compilation, execution des tests unitaires, mise en forme du code dans son packaging (jar, war) et publication dans l'entrepôt local du poste du développeur, pour être réutilisable par d'autres projets.

mvn deploy : copie du produit final dans un repository d'entreprise pour réutilisation par d'autres projets, d'autres développeurs sur d'autres postes de travail.

A noter : les commandes install ou deploy rejouent toutes les commandes précédentes (test, compile…) : si l'une d'entre elles échoue, la cible échoue. Ainsi, on a toujours l'assurance de déployer une application dont tous les tests unitaires ont été exécutés.

Chercher et ajouter des dépendances à un projet

Au fil du développement, on peut être amené à vouloir utiliser telle ou telle classe issue de librairies Java. Par exemple, si on souhaite utiliser un parser XML comme Digester, la première démarche à mener est de trouver dans quel jar (projet maven) cette classe se trouve. Une recherche sur Google permet de trouver dans quelle partie du repository il faut la trouver :

site:www.ibiblio.org digester maven2

On tombe sur la page http://www.ibiblio.org/maven2/commons-digester/commons-digester/ qui permet de constater que la librairie recherchée (artifact jar) s'appelle commons-digester, qu'elle est positionnée dans le group “commons-digester”. Il ne reste qu'à choisir la version de la librairie que l'on souhaite utiliser. Pour ajouter cette dépendance au projet, il faut éditer le pom.xml avec les lignes suivantes :

```
<dependency>
        <groupId>commons-digester</groupId>
        <artifactId>commons-digester</artifactId>
        <version>1.7</version>
</dependency>
```

Suite à cet ajout, la commande mvn compile télécharge le fichier commons-digester-1.7.jar depuis Maven Proxy (qui ira sur Internet si besoin), si cette librairie n'est pas déjà présente sur le poste du développeur.

Quelques Plugins intéréssants

Maven fonctionne par plug-ins. Voici quelques uns des plus intéressants :

Eclipse

Générer les fichiers .classpath et .project à partir d'un descripteur de projet pom.xml pour travailler sous Eclipse

`mvn eclipse:eclipse` : regénérer les fichiers .project et .classpath

`mvn eclipse:clean` : supprimer les fichiers .project et .classpath

Maven écrit les fichiers .classpath et .project. Il s'appuie sur une variable d'Eclipse **M2_REPO**, qu'il faut configurer en allant dans Window>Preferences>Java>Build Path>Classpath Variables

Créer une nouvelle variable **M2_REPO** et sélectionner le répertoire où le repository local de maven se trouve (en principe dans $HOME/.m2/repository)

Dependency

Ce plugin permet de générer l'arbre de dépendances des librairies Java utilisées dans le projet.

Exemple :

```
Rp706248@pc222181:~/workspace_insaa/portlet_cam$ mvn dependency:tree
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'dependency'.
[INFO] ------------------------------------------------------------------------
----
[INFO] Building Portlets AMEL et CODA
[INFO]    task-segment: [dependency:tree]
[INFO] ------------------------------------------------------------------------
----
[INFO] [dependency:tree]
[INFO] fr.***.***.applications.insaa.portlet:Cam:war:1.0.3.3-SNAPSHOT
[INFO] +- fr.***.***.applications.insaa:insaa:jar:1.0.6:provided
[INFO] |  +- javax.servlet:servlet-api:jar:2.4:provided
[INFO] |  +- log4j:log4j:jar:1.2.11:provided
[INFO] |  +- tomcat:catalina:jar:5.5.9:provided
[INFO] |  +- mx4j:mx4j-jmx:jar:3.0.1:provided
[INFO] |  +- portlet-api:portlet-api:jar:1.0:provided
[INFO] |  +- javax.servlet:jsp-api:jar:2.0:provided
[INFO] |  +- axis:axis:jar:1.3:provided
[INFO] |  |  +- axis:axis-jaxrpc:jar:1.3:provided
[INFO] |  |  +- axis:axis-saaj:jar:1.3:provided
[INFO] |  |  +- wsdl4j:wsdl4j:jar:1.5.1:provided
[INFO] |  |  +- commons-logging:commons-logging:jar:1.0.4:provided
[INFO] |  |  \- commons-discovery:commons-discovery:jar:0.2:provided
[INFO] |  +- lucene:lucene-core:jar:1.9.1:provided
[INFO] |  +- lucene:lucene-misc:jar:1.9.1:provided
[INFO] |  \- lucene:lucene-analyzers:jar:1.9.1:provided
[INFO] +- divers:cam:jar:1.0:compile
[INFO] \- commons-lang:commons-lang:jar:2.1:compile
```

Assembly

Lorsque les packaging par défaut prévus dans le pom.xml ne sont pas suffisants, le plug-in assembly permet de générer n'importe quel type de package pour l'application. Voir la doc du plugin assembly

Ce plugin nécessite quelques modifications dans le pom.xml pour déclarer son utilisation, et l'écriture d'un descripteur de l'assembly déclarant ce qu'il faut mettre dans le package.

Utilisation :

```
mvn assembly:assembly
```

Site

Maven permet de générer un site web décrivant le projet. (voir doc, et également ce lien)

Utilisation : mvn site:site pour générer des pages HTML dans (target/site)donnant des informations techniques sur le projet (dépendances, tests unitaires…)

La génération de la Javadoc peut être prise en charge dans le site, en ajoutant la section suivante au fichier pom.xml

```
<reporting>
     <plugins>
       <plugin>
         <artifactId>maven-javadoc-plugin</artifactId>
       </plugin>
       <plugin>
         <groupId>org.codehaus.mojo</groupId>
         <artifactId>jxr-maven-plugin</artifactId>
       </plugin>
     </plugins>
   </reporting>
```

Checkstyle

On peut également ajouter des rapports sur les tests unitaires, checkstyle (doc plugin checkstyle)…

Pour checkstyle, par exemple :

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

VIII - Troubleshooting

illegal character in path at index

Dans certains cas, les tests ne se lancent pas, et vous obtenez le message cryptique suivant : `illegal character in path at index`. Dans ce cas, vous êtes probablement sur un chemin de répertoire avec des espaces. Si c'est effectivement le cas, essayez de travailler dans un chemin de répertoire sans espace, et cela devrait marcher mieux.

Historique des versions

VERSION	DATE	AUTEUR	DESCRIPTION	STATUT
1.0	11/01/2012	POITEVINEAU ROMAIN	Documentation introduction à maven	Check
1.1	18/01/2012	POITEVINEAU ROMAIN	Ajout d'aide pour débuter	TOCHECK