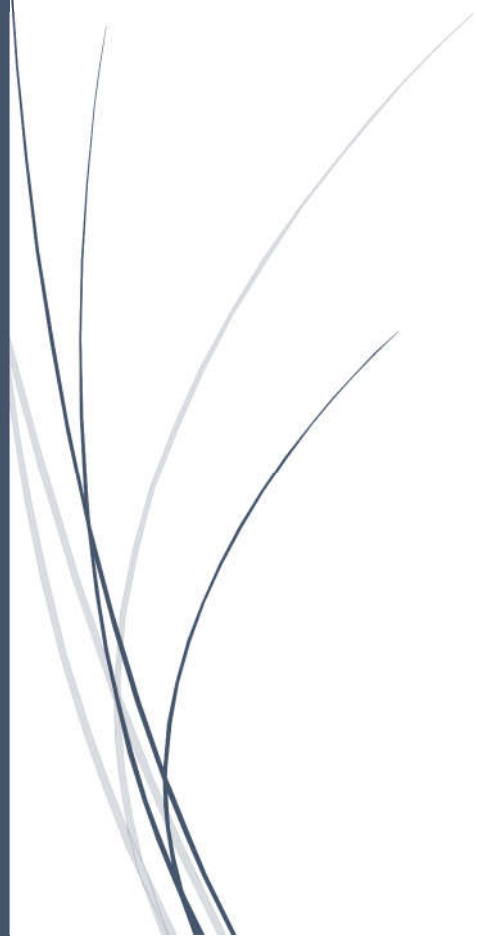
MAVEN

Version 001



@PIGIER CI

SOMMAIRE

I) Présentation MAVEN

Maven est un outil "Open Source" utilisé pour gérer la production de projets logiciels Java de manière général. Maven est un projet de l'organisation Apache Software Foundation et était historiquement une branche de l'organisation Jakarta Project. L'objectif de cet outil peut être comparé au système Make sous Unix, à savoir : produire un logiciel à partir de ses sources, en optimisant les tâches réalisées à cette fin et en garantissant le bon ordre de fabrication. Il est également comparable à l'outil Ant, mais fournit des moyens de configuration plus simples, eux aussi basés sur le format XML.

Ant, Make et bon nombre d'outils similaires s'appuient sur une approche procédurale, pour laquelle on décrit les opérations à accomplir pour construire le logiciel ou exécuter des tâches annexes. Cela se traduit donc par une suite de commandes, qui prendra d'une façon ou d'une autre la forme suivante :

- Initialisation : Préparation des répertoires de travail.
- Compilation : Invocation du compilateur "javac".
- Assemblage : Invocation de l'archiveur "jar".

Cette approche fonctionne très bien mais elle nécessite :

- de répéter pour chaque projet une liste de tâche très similaires.
- de gérer une liste de dépendances entre les étapes clés.

Maven choisit une approche différente, fondée sur le constat suivant : tous les projets Java vont suivre à peu près le même schéma. Ainsi, les développeurs de Maven considèrent qu'il est plus simple de décrire en quoi un projet est différent de ce "scénario type" que de répéter des commandes très comparables d'un projet à l'autre.

Maven exploite donc le concept de "Convention over configuration".

1) Convention

Maven définit donc un scénario type de construction d'un projet Java, avec des étapes clés et un ordre prédéfini. Ce "Cycle de vie" est suffisamment général pour être applicable à quasiment tous les projets. L'avantage des conventions est d'offrir à ceux qui les suivent un outil directement exploitable, sans configuration complémentaire. (Il faut noter qu'il n'est pas nécessaire de respecter ces conventions, mais dès lors que l'on sort des sentiers battus, une configuration supplémentaire doit être effectuée.) Maven propose donc une organisation par défaut, qui peut fonctionner sans plus d'indications pour tout projet qui la respecte.

La force de Maven est de présenter une structure conventionnelle, qui évite à chacun un travail rébarbatif de configuration. Maven reposant sur un scénario type de construction de projet Java, nous n'avons plus besoin d'indiquer la moindre commande. Il suffit de décrire en quoi le projet est différent du cas stéréotypé. Nous passons alors d'une approche programmatique à une solution déclarative.

- Solution déclarative

Maven fait le choix d'une approche déclarative, dans laquelle on indique les particularités du projet et non la manière de le construire. On précise l'emplacement des fichiers sources, les bibliothèques qui sont nécessaires, plutôt que la ligne de commande du compilateur.

Ainsi, il ne s'agit plus de définir les options de "javac", mais de décrire une structure plus générale du projet, qui pourra être exploitée dans un autre contexte. Elle sera, par exemple, utilisée pour s'intégrer dans un IDE (Integrated Development Environment) comme Eclipse ou par les outils d'analyse de code.

L'utilisation des conventions et l'approche fondée sur la description du projet fait de Maven un outil à part, très différent de ANT ou de ses équivalents. En adoptant des conventions, les informations à déclarer pour que le projet soit pris en charge par Maven sont réduites à quelques lignes. La maintenance et l'ajout de nouvelles tâches au cours de la construction du projet s'en trouvent simplifiés. Un développeur, issu d'un contexte très différent mais déjà utilisateur de l'outil, peut prendre le projet en main sans difficulté particulière.

a) Structure des répertoires

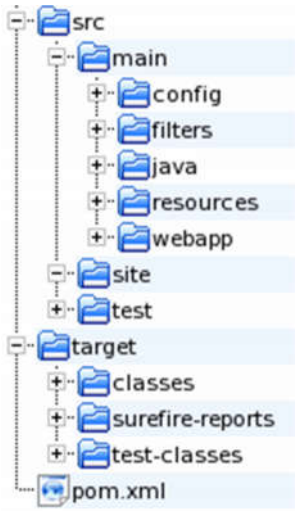
Une des nombreuses conventions proposées par Maven est la structure prédéfinie des répertoires du projet. La logique est plutôt simple : on trouve à la racine le fichier POM qui gère toute la gestion Maven du projet. L'ensemble des sources est placé dans un répertoire *src*, tandis qu'un répertoire *target* sert de zone temporaire pour toutes les opérations réalisées sur le projet. Cela permet de faciliter grandement la configuration de la gestion du code source. Il suffit d'exclure *target* (en plus des fichiers spécifiques de l'IDE) pour être sûr de ne pas inclure des fichiers de travail qui n'ont pas à être partagés.

Sous le répertoire des sources, Maven effectue un découpage explicite entre ce qui fait partie du projet et ce qui sert d'outillage de test. Deux sous-répertoires, *main* et *test*, marquent cette distinction.

Enfin, dans chacune de ces branches, un dernier niveau de répertoires sépare les fichiers sources par langage :

- *"java"* pour le code source des classes Java.
- *"resources"* pour les fichiers de ressources (configuration XML, fichiers de propriétés...).
- *"webapp"* pour les fichiers statiques d'une application web.

Structure des répertoires d'un projet Maven :



La force des conventions Maven n'est pas dans le nom des répertoires qui ont été choisis, mais dans le fait qu'il offre à la communauté des développeurs Java une base commune.

b) Cycle de vie

Le cycle de vie est une suite de phases prédéfinies qui doit couvrir les besoins de tout projet. Ces phases sont symboliques et ne sont associées à aucun traitement particulier, mais elles permettent de définir les étapes clés de la construction du projet. Quels que soient le projet et ses particularités, tout traitement réalisé pour le "construire" fera partie de l'une de ces étapes.

Cycle de vie défini par Maven :

Phase	Description
`validate`	Validation du projet Maven.
`initialize`	Initialisation.
`generate-sources`	Génération de code source.
`process-resources`	Traitement des fichiers de ressources.
`compile`	Compilation des fichiers sources.
`process-classes`	Post-traitement des fichiers binaires compilés.
`test-compile`	Compilation des tests.
`test`	Exécution des tests.
`package`	Assemblage du projet sous forme d'archive Java.
`install`	Mise à disposition de l'archive sur la machine locale pour d'autres projets.
`deploy`	Mise à disposition publique de l'archive Java.

Il s'agit d'une liste simplifiée : le cycle complet définit de nombreuses phases intermédiaires disponibles sur le site officiel de Maven.

c) Project Object Model (POM)

Chaque projet ou sous-projet est configuré par un POM qui contient les informations nécessaires à Maven pour traiter le projet (nom du projet, numéro de version, dépendances vers d'autres projets, bibliothèques nécessaires à la compilation, noms des contributeurs etc.). Ce POM se matérialise par un fichier pom.xml à la racine du projet. Cette approche permet l'héritage des propriétés du projet parent. Si une propriété est redéfinie dans le POM du projet, elle recouvre celle qui est définie dans le projet parent. Ceci introduit le concept de réutilisation de configuration. Le fichier pom du projet principal est nommé pom parent.

Les trois lettres POM sont l'acronyme de *Project Object Model*. Sa représentation XML est traduite par Maven en une structure de données qui représente le modèle du projet. Ces déclarations sont complétées par l'ensemble des conventions qui viennent ainsi former un modèle complet du projet utilisé par Maven pour exécuter des traitements.

La première partie du POM permet d'identifier le projet lui-même :

```
<project>
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.mycompany.app</groupId>
   <artifactId>my-app</artifactId>
   <version>1</version>
</project>
```

- L'élément *modelVersion* permet de savoir quelle version de la structure de données "modèle de projet" est représentée dans le fichier XML. Les futures versions de Maven pourront ainsi exploiter des versions différentes de modèles en parallèle et introduire si nécessaire des évolutions dans le format de ce fichier.
- L'identifiant de groupe *(groupId)* permet de connaitre l'organisation, l'entreprise, l'entité ou la communauté qui gère le projet. Par convention, on utilise le nom de domaine Internet inversé, selon la même logique que celle généralement recommandée pour les noms de packages Java.
- L'identifiant de composant *(artifactId)* est le nom unique du projet au sein du groupe qui le développe. En pratique et pour éviter des confusions, il est bon d'avoir un *artifactId* unique indépendamment de son *groupId*.
- Enfin l'élément *version* permet de préciser quelle version du projet est considérée.La plupart des projets utilisent la formule <*Version Majeure*>.<*Version Mineure*>.<*Correctif*>.

La deuxième partie du POM concerne la construction du projet :

```
<build>
   <sourceDirectory>src</sourceDirectory>
</build>
```

- L'approche déclarative utilisée par Maven permet de définir l'emplacement des fichiers sources. Si les conventions sur les noms de répertoires ont été respectées, ce bloc <*build*> n'est pas nécessaire.

d) Gestion des dépendances

La troisième partie du POM concerne les bibliothèques dont dépend le projet :

```
<dependencies>
   <dependency>
     <groupId>javax.mail</groupId>
     <artifactId>mail</artifactId>
     <version>1.4</version>
   <dependency>
   <dependency>
     <groupId>commons-io</groupId>
     <artifactId>commons-io</artifactId>
     <version>1.4</version>
   <dependency>
```

```
</dependencies>
```

- Une nouvelle fois, l'approche déclarative prend le dessus : il n'est pas necéssaire d'indiquer l'emplacement physique de ces bibliothèques (ex: /lib) mais uniquement les identifiants *groupId* + *arifactId* + *version*. Il s'agit des mêmes identifiants de groupe, de composant et de version utilisés pour un projet mais appliqués à une bibliothèque. Dans cet exemple nous souhaitons utiliser l'API standard JavaMail en version 1.4.

- Dépendances Transitives

Une des nouveautés de Maven 2 est la gestion des dépendances transitives. Avec Maven 1, il fallait déclarer chacun des JAR dont avait besoin, directement ou indirectement, l'application. Avec Maven 2, cela n'est plus nécessaire. En indiquant juste les bibliothèques dont l'application a besoin, Maven se charge des bibliothèques dont les bibliothèques de l'application ont besoin (et ainsi de suite).

- Portée des dépendances

Dans une application d'entreprise du monde réel, il n'est pas nécessaire d'inclure toutes les dépendances dans l'application déployée. Certains des JARs sont nécessaires uniquement pour les tests unitaires, alors que d'autres seront fournis à l'exécution par le serveur d'application. En utilisant la technique de la portée de dépendances, Maven 2 permet d'utiliser certain JAR uniquement en cas de besoin et de les exclure du classpath dans le cas contraire. Maven met à disposition quatre portés de dépendances :

- *compile:* Une dépendance de portée compile est disponible dans toutes les phases. C'est la valeur par défaut.
- *provided:* Une dépendance de portée provided est utilisée pour compiler l'application, mais ne sera pas déployée. Vous utiliserez cette portée quand vous attendez du JDK ou du serveur d'application qu'il vous mette le JAR à disposition. L'API servlet est un bon exemple.
- *runtime:* Les dépendances de portées runtime ne sont pas nécessaires pour la compilation, uniquement pour l'exécution, comme les drivers JDBC (Java Database Connectivity).
- *test:* Les dépendances de portées test sont uniquement nécessaires pour compiler et exécuter les tests (par exemple Junit).

Maven va ainsi télécharger les bibliothèques indiquées, à partir d'une source fiable, plutôt que de se contenter des fichiers JAR presents dans le répertoire */lib* et dont la version et l'origine sont incertaines. L'espace contenant l'ensemble des bibliothèques téléchargées est un dépôt d'archives local *(local repository)* et respecte une convention.

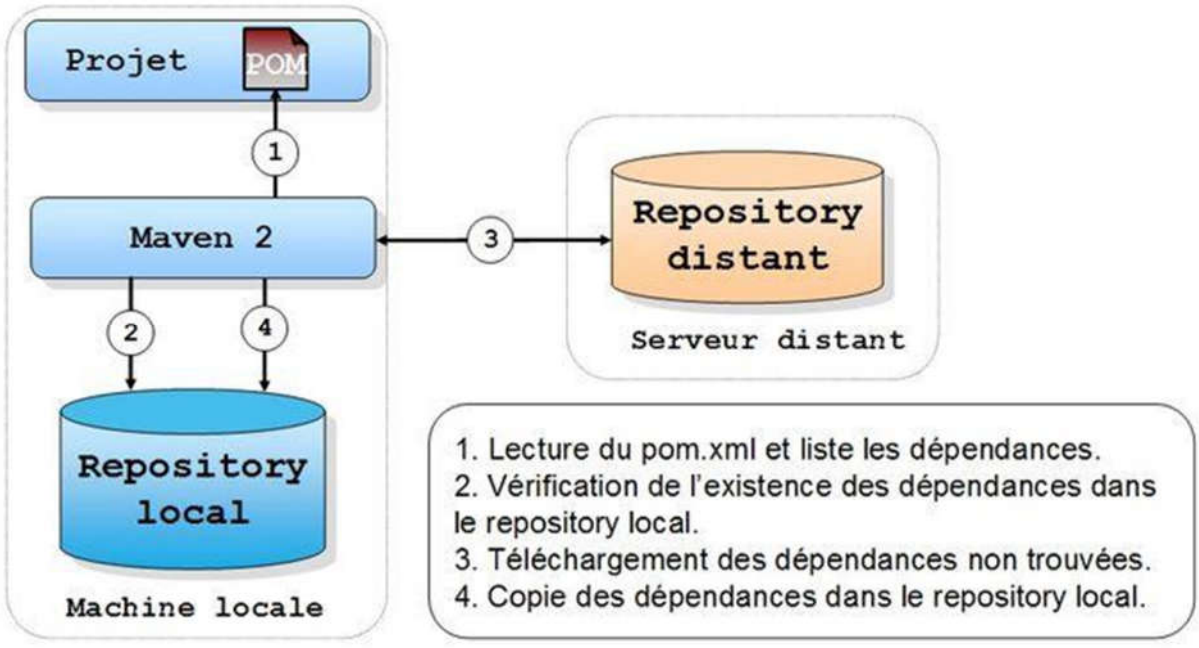
e) Dépôt

Un autre apport de l'outil Maven est son organisation des projets et plugins. Maven dispose de plusieurs référentiels à plusieurs niveaux. Le but du référentiel est de rendre disponible aussi bien les plugins utilisés ou envisagés de l'être que les projets générés par Maven. On peut bien sûr y installer des projets pour les utiliser (sans qu'ils ne soient générés par Maven). Il y a trois référentiels :

- Un au niveau de la machine du développeur, appelé repository local, il inclut tout ce que le développeur a utilisé et a développé.
- Un au niveau du site Maven qui contient l'ensemble des plugins. Il est en augmentation continue. Il est ouvert à tout le monde, en conséquence des mises à jour pourraient être incohérentes ou incompatibles avec d'autres.
- Pour éviter ceci, il y a un troisième référentiel (facultatif) qui peut être déclaré au niveau de l'entreprise ou la structure utilisant Maven, il joue l'intermédiaire entre les deux premiers référentiels, il inclut les plugins (maîtrise des versions et des mises à jour) et les projets (les rendant accessible à l'ensemble des développeurs).

Pour créer un référentiel pour l'entreprise (ou un référentiel commun en général), on peut utiliser les protocoles ftp, scp, file et http.

Gestion des dépendances et dépôts :



La configuration par défaut de Maven utilise le dépôt (ou référentiel) de bibliothèque http://repo1.maven.org/maven2/. Ce site, maintenu par la communauté Maven, compte plusieurs dizaines de giga-octets de bibliothèques libres de diffusion et est mis à jour plusieurs fois par jour.

A partir de notre déclaration de dépendance, Maven va construire l'URL du sous-répertoire dédié à la bibliothèque indiquée :

- *<URL du dépôt> / <groupId en tant que chemin> / <artifactId> / <version>*

En reprenant l'exemple de la dépendance JavaMail, on obtient : http://repo1.maven.org/maven2/javax/mail/mail/1.4

2) Plugin

Quand on télécharge Maven, il ne comprend que le moteur qui sert à télécharger des plugins. Tous les goals Maven sont dans des plugins même les plus indispensables comme le plugin compiler. Ainsi, il faut s'attendre à voir Maven télécharger énormément de plugins lors de la première exécution d'un goal. En rappelant le groupId et l'artifactId du plugin, on identifie de façon certaine le plugin. Il est alors possible de spécifier des paramètres génériques ou des paramètres spécifiques au plugin. Spécifier la version a son importance. Tous les plugins peuvent avoir des mises à jour et par défaut Maven vérifiera une fois par jour si une nouvelle version est disponible. Il arrive que les meilleurs plugins sortent des versions buggées, et sans fixer la version, le projet peut alors poser des problèmes à la compilation d'un jour à l'autre parce que la nouvelle version a été téléchargée automatiquement.

Maven confie chaque opération élémentaire de la construction du projet à un plugin, un fragment du logiciel qui se spécialise dans une tâche donnée. La compilation est un exemple de plugin, comme l'assemblage sous forme de JAR ou encore l'inclusion de fichiers de ressources, etc. Chaque plugin propose un certain nombre d'options et de paramètres qui permettent d'ajuster son fonctionnement, avec des valeurs par défaut qui sont choisies pour coller au mieux aux conventions de Maven et à une utilisation standard.

La modification des options par défaut d'un plugin s'effectue dans le fichier POM du projet, au sein de son bloc <*build*> :

```
<build>
 <plugins>
   <plugin>
     <groupId>org.apache.maven.plugins</groupId>
     <artifactId>maven-compiler-plugin</artifactId>
     <version>2.0.2</version>
     <configuration>
       <source>1.5</source>
       <target>1.5</target>
     </configuration>
   </plugin>
 </plugins>
</build>
```

Chaque plugin peut ainsi être reconfiguré. Un plugin, comme tout artefact manipulé par Maven, est identifié par le triplet (identifiant de groupe, identifant d'artefact, version). Dans cet exemple nous indiquons le plugin dont nous désirons ajuster le fonctionnement. Le bloc <*configuration*> permet de lui passer des valeurs qui vont remplacer celles par défaut. Chaque plugin ayant son propre paramétrage, des documentations sont disponibles afin de connaître toutes les options possibles.

a) Les rapports Maven

Encore une fois, l'intégration des outils d'analyse avec la construction du projet Maven passe par des plugins. Il est possible, par exemple, de lancer la commande *mvn checkstyle:check* pour vérifier le respect des règles de codage, voire de le configurer pour qu'il s'exécute pendant un build normal. Maven propose cependant une autre forme d'intégration pour les outils qui visent à compléter la documentation sur le projet : les rapports.

En parallèle du cycle de vie de construction du projet, Maven propose un cycle de vie documentaire déclenché par la commande : *mvn site*. Cette commande n'exécute qu'un seul plugin, *site* : son rôle est d'exécuter des rapports et de mettre le contenu dans un document unifié sous forme de site web. Ces rapports sont eux-mêmes déclarés dans notre POM, sous un élément dédié <*reporting*>.

Les rapports générés par Maven portent sur :

- Des informations générales sur le projet, comme les entrepôts de source, le suivi des anomalies, les membres de l'équipe, etc.
- Les tests unitaires et les rapports de couverture des tests unitaires
- Des revues de code automatisées avec Checkstyle ou PMD
- Des informations sur la configuration ou le versionnage
- Les dépendances
- La JavaDoc
- Le code source indexé et référençable, sous un format HTML
- La liste des membres de l'équipe
- Et beaucoup plus encore

II) CAS PRATIQUE avec PLUGIN ARCHETYPE

a) Installation

Les différentes versions disponibles de Maven sont téléchargeables sur le site http://maven.apache.org/download.html.

- Installation sous windows

L'installation de Maven sur Windows est très similaire a celle sous Mac OSX. Les principales différences se situent au niveau du répertoire d'installation et de la configuration des variables d'environnement. Dans cet exemple nous installons Maven dans le répertoire *C:\Program Files\apache-maven-2.2.1*, mais il est possible de l'installer autre part du moment que les variables d'environnement sont correctement configurées. Une fois que l'archive a été dézippée dans le répertoire d'installation, il faut configurer les variables d'environnement *M2_HOME* et *PATH* en exécutant les lignes de commande suivantes :

```
C:\Users\javeline > set M2_HOME=c:\Program Files\apache-maven-2.2.1
C:\Users\javeline > set PATH=%PATH%;%M2_HOME%\bin
```

Modifier ces variables d'environnement vous permettra de lancer Maven dans la session courante, mais tant que ces variables ne sont pas ajoutées aux variables d'environnement système, il faudra les exécuter de nouveau à chaque ouverture de session. Ces variables peuvent être modifiées à l'aide du panel de contrôle Microsoft Windows.

- Vérification de l'installation

Afin de vérifier que l'installation a été effectué correctement, exécuter cette ligne de commande dans une console :

```
mvn -version
```

Si la commande indique un numéro de version, c'est que l'installation s'est bien déroulée.

b) Les Archetypes Maven

Un Archetype est un outil pour faire des Template de projet Maven. A l'aide de nombreux Archetypes déjà existants que propose Maven, il est possible de créer un projet prêt à compiler et déployer. Il est possible d'utiliser des Archetypes existants ou de créer ses propres Archetypes Maven.

Pour de plus amples informations concernant les Archetypes Maven disponibles, veuillez consulter le lien suivant : http://maven.apache.org/guides/introduction/introduction-to-archetypes.html

- Utilisation d'un Archetype de Maven

La commande à lancer pour créer un projet à partir d'un Archetype est presque toujours décrite avec l'Archetype. Néanmoins, la commande est la suivante :

```
mvn archetype:create
        -DarchetypeGroupId=org.apache.tapestry
        -DarchetypeArtifactId=quickstart
        -DgroupId=org.example
        -DartifactId=myapp
        -DpackageName=org.example.myapp
        -Dversion=1.0.0-SNAPSHOT
```

L'Archetype est un projet Maven identifié avec un groupId et artifactId comme les autres. Les paramètres à renseigner correspondent donc à la localisation du paquet de l'Archetype auxquels on ajoute les paramètres du projet que l'on va créer : son nom, les packages des sources et sa version. Cette commande va générer tout le nécessaire pour lancer un projet démo avec Tapestry.

Il existe un grand nombre d'Archetypes à disposition. La commande *mvn archetype:generate* permet de créer un projet à partir d'une liste d'Archetype disponible :

```
$ mvn archetype:generate
```

```
[INFO] ------------------------------------------------------------
[INFO] Building Maven Default Project
[INFO] task-segment: [archetype:generate] (aggregator-style)
[INFO] [archetype:generate]
[INFO] Generating project in Interactive mode
[INFO] No archetype defined. Using maven-archetype-quickstart
                Choose archetype:
                1: internal -> appfuse-basic-jsf
                2: internal -> appfuse-basic-spring
                3: internal -> appfuse-basic-struts
                4: internal -> appfuse-basic-tapestry
                5: internal -> appfuse-core
                6: internal -> appfuse-modular-jsf
                7: internal -> appfuse-modular-spring
                8: internal -> appfuse-modular-struts
                9: internal -> appfuse-modular-tapestry
                10: internal -> maven-archetype-j2ee-simple
                11:  internal  ->  maven-archetype-marmalade-
mojo
                12: internal -> maven-archetype-mojo
                13: internal -> maven-archetype-portlet
                14: internal -> maven-archetype-profiles
                15: internal -> maven-archetype-quickstart
                16: internal -> maven-archetype-site-simple
                17: internal -> maven-archetype-site
                18: internal -> maven-archetype-webapp
                19: internal -> jini-service-archetype
                20: internal -> softeu-archetype-seam
                21: internal -> softeu-archetype-seam-simple
                22: internal -> softeu-archetype-jsf
                23: internal -> jpa-maven-archetype
                24: internal -> spring-osgi-bundle-archetype
                25: internal -> confluence-plugin-archetype
                26: internal -> jira-plugin-archetype
                27: internal -> maven-archetype-har
                28: internal -> maven-archetype-sar
                29: internal -> wicket-archetype-quickstart
                30: internal -> scala-archetype-simple
                31: internal -> lift-archetype-blank
                32: internal -> lift-archetype-basic
                33:  internal  ->  cocoon-22-archetype-block-
plain
                34: internal -> cocoon-22-archetype-block
                35: internal -> cocoon-22-archetype-webapp
                36: internal -> myfaces-archetype-helloworld
                37: internal -> myfaces-archetype-helloworld-
facelets
                38: internal -> myfaces-archetype-trinidad
                39:    internal    ->    myfaces-archetype-
jsfcomponents
                40: internal -> gmaven-archetype-basic
                41: internal -> gmaven-archetype-mojo
                Choose a number: 15
```

Une fois qu'un Archetype a été sélectionné, il faut configurer les valeurs des paramètres suivants :

- groupId
- artifactId

- version
- package

```
Define value for groupId: : org.sonatype.mavenbook
Define value for artifactId: : quickstart
Define value for version:  1.0-SNAPSHOT
Define value for package:  org.sonatype.mavenbook
Confirm properties configuration:
groupId: org.sonatype.mavenbook
artifactId: quickstart
version: 1.0-SNAPSHOT
package: org.sonatype.mavenbook
Y: : Y
```

Puis le projet est généré dans le répertoire du nom de l'*artifactId* renseigné précédemment.

```
[INFO] Parameter: groupId, Value: org.sonatype.mavenbook
[INFO] Parameter: packageName, Value: org.sonatype.mavenbook
[INFO] Parameter: basedir, Value: /Users/tobrien/tmp
[INFO] Parameter: package, Value: org.sonatype.mavenbook
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: artifactId, Value: quickstart
[INFO] ********************* End of debug info from resources from \
                                  generated POM **
[INFO] OldArchetype created in dir: /Users/tobrien/tmp/quickstart
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1 minute 57 seconds
[INFO] Finished at: Sun Oct 12 15:39:14 CDT 2008
[INFO] Final Memory: 8M/15M
[INFO] ------------------------------------------------------------------------
```

c) Créer un Plug-in Maven

Il est possible de créer ses propres plugins Maven en utilisant l'Archetype correspondant.

Dans une console de commandes, exécuter la commande suivante :

```
                              mvn archetype:create
                                     -DgroupId=exempleplugin
                                     -DartifactId=exemple-plugin
                                     -
DarchetypeGroupId=org.apache.maven.archetypes
                                     -DarchetypeArtifactId=maven-
archetype-mojo
```

```
                                    -DarchetypeVersion=1.0-alpha-4
```

Vous trouverez une classe Java nommée /exemple-plugin/src/main/java/exempleplugin/MyMojo.java contenant le code du plugin. Ce modèle de plugin généré par l'archetype a créé un goal nommé touch. Pour utiliser ce plugin, il faut maintenant l'installer.

Dans une console de commandes,accéder au répertoire du projet de plugin, et exécuter les commandes suivantes :

```
                                mvn install
                                mvn exempleplugin:exemple-plugin:touch
```

Vous verrez alors un répertoire target créé dans le répertoire projet et un fichier touch.txt à l'intérieur.

- Définir un paramètre Plug-in

On ajoute une variable de classe dans le fichier Java du plugin comme ceci :

```
                                /**
                                * Mon paramètre.
                                * @parameter expression="${mymavenparameter}"
                                * @required
                                */
                                private String myJavaParameter;
```

Nous venons de créer une variable myJavaParameter, de type String, obligatoire(@required) et correspondant au paramètre mymavenparameter du pom.xml du projet exécutant le plugin :

```
                                <project>
                                ...
                                  <build>
                                    <plugins>
                                      <plugin>
                                          <groupId>exempleplugin</groupId>
                                          <artifactId>exemple-
plugin</artifactId>
                                          <configuration>

<mymavenparameter>valeur</mymavenparameter>
                                        </configuration>
                                      </plugin>
                                    </plugins>
```

```
  </build>
...
</project>
```

- Récupérer les propriétés du projet exécutant le Plug-in

On ajoute la variable de classe suivante (avec l'annotation) :

```
/**
* Projet en cours de deploiement.
* @parameter expression="${project}"
*/

private org.apache.maven.project.MavenProject
project;
```

Nous disposons maintenant d'une variable project qui sera automatiquement initialisée par Maven à l'exécution et nous avons maintenant accès aux informations telles que le groupId, l'artifactId, la version du projet, etc.

- Voir les sources d'un Plug-in

Les sources des plugins Maven sont accessibles, ce qui permet à la fois de voir comment est réalisé un plugin, mais également de proposer votre participation. Prenons pour exemple le plugin clean, dont la page principale est http://maven.apache.org/plugins/maven-clean-plugin. Les sources du plugin clean sont visibles dans un navigateur via ViewVC à l'adresse suivante http://svn.apache.org/viewvc/maven/plugins/trunk/maven-clean-plugin.

On peut également créer une copie locale du plugin clean en utilisant Subversion avec la commande suivante :

```
svn checkout http://svn.apache.org/repos/asf/maven/plugins/trunk/maven-clean-plugin maven-clean-plugin
```