Développons en Java

93. Maven

Niveau : Supérieur

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

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

Il permet notamment :

- d'automatiser certaines tâches : compilation, tests unitaires et déploiement des applications qui composent le projet
- de gérer des dépendances vis-à-vis des bibliothèques nécessaires au projet
- de générer des documentations concernant le projet

Au premier abord, il est facile de croire que Maven fait double emploi avec Ant. Ant et Maven sont tous les deux développés par le groupe Jakarta, ce qui prouve bien que leur utilité n'est pas aussi identique que cela. Ant, dont le but est d'automatiser certaines tâches répétitives, est plus ancien que Maven. Maven propose non seulement les fonctionnalités d'Ant mais en propose de nombreuses autres.

Pour gérer les dépendances du projet vis-à-vis de bibliothèques, Maven utilise un ou plusieurs dépôts qui peuvent être locaux ou distants.

Maven est extensible grâce à un mécanisme de plugins qui permettent d'ajouter des fonctionnalités.

Ce chapitre contient plusieurs sections :

93.1. Les différentes versions de Maven

Plusieurs versions majeures de Maven ont été diffusées :

- version 1.0, juillet 2004
- version 1.1, juin 2007
- version 2.0, octobre 2005

- version 2.1, mars 2009
- version 3.0, octobre 2010
- version 3.1, juillet 2013

Maven 2 est très différent de Maven 1.

Maven 3 est compatible avec Maven 2 et apporte notamment de meilleures performances.

93.2. Maven 1

Pour assurer la construction d'un projet, Maven propose notamment de prendre en charge :

- La compilation
- Le packaging
- La gestion des dépendances
- La génération de la documentation
- L'accès au gestionnaire de sources
- L'accès aux dépôts ou aux gestionnaires de dépendances
- Le déploiement
- ... et de très nombreuses autres tâches requises lors d'un build

Maven impose par défaut l'emploi de conventions notamment dans la structuration du projet.

Maven utilise une approche déclarative où la structure du projet et son contenu sont décrits dans un document XML. De plus il convient de se conformer à une structure de projets standards et de bonnes pratiquees. L'observation de ces normes permet de réduire le temps nécessaire pour écrire et maintenir les scripts de build car ils sont tous structurés de la même façon.

La description d'un projet est faite dans un fichier XML nommé POM (Project Object Model). Cette description contient notamment les dépendances, les spécificités de construction (compilation et packaging), éventuellement le déploiement, la génération de la documentation, l'exécution d'outils d'analyse statique du code, ...

Maven peut aussi assurer de nombreuses autres tâches car il est conçu pour utiliser des plugins : il est donc extensible. Maven est fourni avec un grand nombre de plugins standard mais il est aussi possible d'utiliser d'autres plugins qui sont stockés dans les dépôts voire même de développer ses propres plugins.

Maven permet une gestion des artefacts (dépendances, plugin-ins) qui sont stockées dans un ou plusieurs dépôts (repository).

93.2.1. L'installation

Il faut télécharger le fichier maven-1.0-rc2.exe sur le site de Maven et l'exécuter.

Un assistant permet de fournir les informations concernant l'installation :

- sur la page « Licence Agreement » : lire la licence et si vous l'acceptez cliquer sur le bouton « I Agree ».
- sur la page « Installations Options » : sélectionner les éléments à installer et cliquer sur le bouton « Next ».
- sur la page « Installation Folder » : sélectionner le répertoire dans lequel Maven va être installé et cliquer sur le bouton « Install ».
- une fois les fichiers copiés, il suffit de cliquer sur le bouton « Close ».

Sous Windows, un élément de menu nommé « Apache Software Foundation / Maven 1.0-rc2 » est ajouté dans le menu « Démarrer / Programmes ».

Pour utiliser Maven, la variable d'environnement système nommée MAVEN_HOME doit être définie avec comme valeur le chemin absolu du répertoire dans lequel Maven est installé. Par défaut, cette variable est configurée automatiquement lors de l'installation sous Windows.

Il est aussi particulièrement pratique d'ajouter le répertoire %MAVEN_HOME%/bin à la variable d'environnement PATH. Maven étant un outil en ligne de commande, cela évite d'avoir à saisir son chemin

complet lors de son exécution.

Enfin, il faut créer un repository local en utilisant la commande ci-dessous dans une boîte de commandes DOS :

Exemple :

```
C:\>install_repo.bat %HOMEDRIVE%%HOMEPATH%
```

Pour s'assurer de l'installation correcte de Maven, il suffit de saisir la commande :

Exemple :

```
C:\>maven -v

 __  __
|  \/  |__ _Apache__ ___
| |\/| / _` \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|  v. 1.0-rc2
C:\>
```

Lors de la première exécution de Maven, ce dernier va constituer le repository local (une connexion internet est nécessaire).

Exemple :

```
|  \/  |__ _Apache__ ___
| |\/| / _` \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|  v. 1.0-rc2
Le rÚpertoire C:\Documents and Settings\Administrateur\.maven\repository n'exist
e pas. Tentative de crÚation.
Tentative de tÚlÚchargement de commons-lang-1.0.1.jar.
.............................
.
Tentative de tÚlÚchargement de commons-net-1.1.0.jar.
.....................................................................
........
.
Tentative de tÚlÚchargement de dom4j-1.4-dev-8.jar.
.....................................................................
.....................................................................
.....................................................................
...........................................................
.
Tentative de tÚlÚchargement de xml-apis-1.0.b2.jar.
.....................................................
```

93.2.2. Les plugins

Toutes les fonctionnalités de Maven sont proposées sous la forme de plugins.

Le fichier maven.xml permet de configurer les plugins installés.

93.2.3. Le fichier project.xml

Maven est orienté projet, donc le projet est l'entité principale gérée par Maven. Il est nécessaire de fournir à Maven une description du projet (Project descriptor) sous la forme d'un document XML nommé

project.xml et situé à la racine du répertoire contenant le projet.

Exemple : un fichier minimaliste

```
<project>

  <id>P001</id>
  <name>TestMaven</name>
  <currentVersion>1.0</currentVersion>
  <shortDescription>Test avec Maven</shortDescription>

  <developers>
    <developer>
      <name>Jean Michel D.</name>
      <id>jmd</id>
      <email>jmd@test.fr</email>
    </developer>
  </developers>

  <organization>
    <name>Jean-Michel</name>
  </organization>

</project>
```

Il est possible d'inclure la valeur d'un tag défini dans le document dans un autre tag.

Exemple :

```
...
<shortDescription>${pom.name} est un test avec Maven</shortDescription>
...
```

Il est possible d'hériter d'un fichier project.xml existant dans lequel des caractéristiques communes à plusieurs projets sont définies. La déclaration dans le fichier du fichier père se fait avec le tag <extend>. Dans le fichier fils, il suffit de redéfinir ou de définir les tags nécessaires.

93.2.4. L'exécution de Maven

Maven s'utilise en ligne de commande sous la forme suivante :

Maven plugin:goal

Il faut exécuter Maven dans le répertoire qui contient le fichier project.xml.

Si les paramètres fournis ne sont pas corrects, une exception est levée :

Exemple :

```
C:\java\test\testmaven>maven compile

 __  __
|  \/  |__ _Apache__ ___
| |\/| / _` \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|  v. 1.0-rc2
com.werken.werkz.NoSuchGoalException: No goal [compile]
        at com.werken.werkz.WerkzProject.attainGoal(WerkzProject.java:190)
        at org.apache.maven.plugin.PluginManager.attainGoals(PluginManager.java:
531)
        at org.apache.maven.MavenSession.attainGoals(MavenSession.java:265)
```

```
        at org.apache.maven.cli.App.doMain(App.java:466)
        at org.apache.maven.cli.App.main(App.java:1117)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcces
sorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:324)
        at com.werken.forehead.Forehead.run(Forehead.java:551)
        at com.werken.forehead.Forehead.main(Forehead.java:581)
Total time: 4 seconds
Finished at: Tue May 18 14:17:18 CEST 2004
```

Pour obtenir une liste complète des plugins à disposition de Maven, il suffit d'utiliser la commande maven -g

Voici quelques-uns des nombreux plugins avec leurs goals principaux :

Plug in	Goal	Description
ear	ear	construire une archive de type ear
	deploy	déployer un fichier ear dans un serveur d'application
ejb	ejb	
	deploy	
jalopy	format	
java	compile	compiler des sources
	jar	créer une archive de type .jar
javadoc		
jnlp		
pdf		générer la documentation du projet au format PDF
site	generate	générer le site web du projet
	deploy	copier le site web sur un serveur web
test	match	exécuter des tests unitaires
war	init	
	war	
	deploy	

La commande maven clean permet d'effacer tous les fichiers générés par Maven.

Exemple :

```
C:\java\test\testmaven>maven clean

 __  __
|  \/  |__ _Apache__ ___
| |\/| / _` \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|  v. 1.0-rc2
build:start:
clean:clean:

```

```
    [delete] Deleting directory C:\java\test\testmaven\target
    [delete] Deleting: C:\java\test\testmaven\velocity.log
BUILD SUCCESSFUL
Total time: 2 minutes 7 seconds
Finished at: Tue May 25 14:19:03 CEST 2004
C:\java\test\testmaven>
```

93.2.5. La génération du site du projet

Maven propose une fonctionnalité qui permet de générer automatiquement un site web pour le projet regroupant un certain nombre d'informations utiles le concernant.

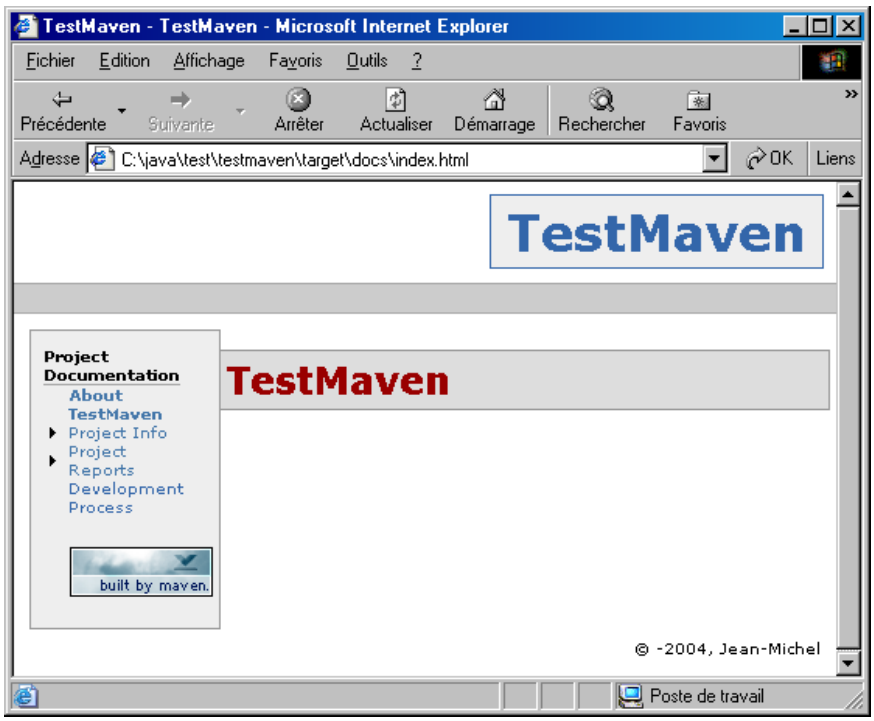
Pour demander la génération du site, il suffit de saisir la commande

maven site:generate

Lors de l'exécution de cette commande, un répertoire target/docs est créé contenant les différents éléments du site.

Exemple :

```
C:\java\test\testmaven\target\docs>dir
 Le volume dans le lecteur C s'appelle MACHINE
 Le numéro de série du volume est 3T78-19E4
 Répertoire de C:\java\test\testmaven\target\docs
25/05/2004  14:21       <DIR>          .
25/05/2004  14:21       <DIR>          ..
25/05/2004  14:21       <DIR>          apidocs
25/05/2004  14:21                5 961 checkstyle-report.html
25/05/2004  14:21                1 637 cvs-usage.html
25/05/2004  14:21                1 954 dependencies.html
25/05/2004  14:21       <DIR>          images
25/05/2004  14:21                1 625 index.html
25/05/2004  14:21                1 646 issue-tracking.html
25/05/2004  14:21                3 119 javadoc.html
25/05/2004  14:21                9 128 jdepend-report.html
25/05/2004  14:21                2 494 license.html
25/05/2004  14:21                5 259 linkcheck.html
25/05/2004  14:21                1 931 mail-lists.html
25/05/2004  14:21                4 092 maven-reports.html
25/05/2004  14:21                3 015 project-info.html
25/05/2004  14:21       <DIR>          style
25/05/2004  14:21                2 785 task-list.html
25/05/2004  14:21                3 932 team-list.html
25/05/2004  14:21       <DIR>          xref
              14 fichier(s)           48 578 octets
               6 Rép(s)     207 151 616 octets libres
```



Par défaut, le site généré contient un certain nombre de pages accessibles par le menu de gauche.

La partie « Project Info » regroupe trois pages : la mailing liste, la liste des développeurs et les dépendances du projet.

La partie « Project report » permet d'avoir accès à des comptes rendus d'exécution de certaines tâches : javadoc, tests unitaires, ... Certaines de ces pages ne sont générées qu'en fonction des différents éléments produits par Maven.

Le contenu du site pourra donc être réactualisé facilement en fonction des différents traitements réalisés par Maven sur le projet.

93.2.6. La compilation du projet

Dans le fichier project.xml, il faut rajouter un tag <build> qui va contenir les informations pour la compilation des éléments du projet.

Les sources doivent être contenues dans un répertoire dédié, par exemple src

Exemple :

```
...
  <build>
      <sourceDirectory>
        ${basedir}/src
      </sourceDirectory>
    </build>
...
```

Pour demander la compilation à Maven, il faut utiliser la commande Maven java :compile :

Exemple :

```
C:\java\test\testmaven>maven java:compile

 __  __
|  \/  |__ _Apache__ ___

```

```
| |\/| / _` \ V / -_) ' \  ~ intelligent projects ~
|_|  |_\__,_|\_/\___|_||_|  v. 1.0-rc2
Tentative de tÚlÚchargement de commons-jelly-tags-antlr-20030211.143720.jar.
.....
.
build:start:
java:prepare-filesystem:
    [mkdir] Created dir: C:\java\test\testmaven\target\classes
java:compile:
    [echo] Compiling to C:\java\test\testmaven/target/classes
    [javac] Compiling 1 source file to C:\java\test\testmaven\target\classes
BUILD SUCCESSFUL
Total time: 12 seconds
Finished at: Tue May 18 14:19:12 CEST 2004
```

Le répertoire « target/classes » est créé à la racine du répertoire du projet. Les fichiers .class issus de la compilation sont stockés dans ce répertoire.

La commande maven jar permet de demander la génération du packaging de l'application.

Exemple :

```
build:start:
java:prepare-filesystem:
java:compile:
    [echo] Compiling to C:\java\test\testmaven/target/classes
java:jar-resources:
test:prepare-filesystem:
    [mkdir] Created dir: C:\java\test\testmaven\target\test-classes
    [mkdir] Created dir: C:\java\test\testmaven\target\test-reports
test:test-resources:
test:compile:
    [echo] No test source files to compile.
test:test:
    [echo] No tests to run.
jar:jar:
    [jar] Building jar: C:\java\test\testmaven\target\P001-1.0.jar
BUILD SUCCESSFUL
Total time: 2 minutes 42 seconds
Finished at: Tue May 18 14:25:39 CEST 2004
```

Par défaut, l'appel à cette commande effectue une compilation des sources, un passage des tests unitaires s'il y en a et un appel à l'outil jar pour réaliser le packaging.

Le nom du fichier jar créé est composé de l'id du projet et du numéro de version. Il est stocké dans le répertoire racine du projet.

93.3. Maven 2

Maven 2 est une version différente de Maven 1 : ces deux versions ne sont d'ailleurs pas compatibles.

La version du fichier POM qui décrit un projet est passé de la version 3.0 à 4.0 : le nom par défaut est pom.xml avec Maven 2.

Le coeur de Maven 2 utilise un conteneur d'injection de dépendances (IoC) nommé Plexus.

93.3.1. L'installation et la configuration

Maven est un outil écrit en Java : Java doit donc être installé sur la machine. Généralement, surtout sur un ordinateur qui n'est pas connecté sur un réseau d'entreprise disposant d'un gestionnaire de dépôts, une connexion à internet est nécessaire pour permettre le téléchargement des plugins requis et des dépendances.

Pour installer Maven, il faut :

- Télécharger l'archive sur le site http://maven.apache.org/download.html
- Décompresser l'archive dans un répertoire du système
- Créer la variable d'environnement M2_HOME qui pointe sur le répertoire contenant Maven
- Ajouter le chemin M2_HOME/bin à la variable PATH du système

Pour vérifier l'installation, il faut lancer la commande mvn -version

Le répertoire de Maven contient plusieurs sous-répertoires :

- bin : des scripts dont la commande mvn
- conf : contient la configuration par défaut dans le fichier settings.xml
- lib : les bibliothèques contenant le noyau de Maven

La configuration du chemin du dépôt local se fait dans le fichier settings.xml du sous-répertoire .m2 contenu dans le répertoire home de l'utilisateur.

Le tag localRepository permet de préciser le chemin absolu du dépôt local.

Exemple :

```
1 <settings>
2   ...
3 <localRepository>C:\Documents and Settings\jm\.m2\repository</localRepository>
4   ...
5 </settings>
```

Pour tester l'installation, il est possible d'exécuter la commande mvn -clean

Cette commande échoue car elle ne trouve pas de fichier POM mais auparavant, elle télécharge des plugins du dépôt central vers le dépôt local.

93.3.2. Les concepts

Maven repose sur l'utilisation de plusieurs concepts :

- Les artéfacts : composants identifiés de manière unique
- Le principe de convention over configuration : utilisation de conventions par défaut pour standardiser les projets
- Le cycle de vie et les phases : les étapes de construction d'un projet sont standardisées
- Les dépôts (local et distant)

93.3.2.1. Les artéfacts

Un artéfact est un composant packagé possédant un identifiant unique composé de trois éléments : un groupId, un artifactId et un numéro de version.

La gestion des versions est importante pour identifier quel artefact doit être utilisé : la version est utilisée comme une partie de l'identifiant d'un artéfact.

Les versions standard correspondent à des releases de l'artéfact.

Les versions en cours de développement se terminent par -SNAPSHOT : ce sont des versions intermédiaires de travail en local.

Maven va systématiquement rechercher une version plus récente pour une dépendance dont le numéro de version est un SNAPSHOT.

Le numéro de version d'un artéfact Maven se compose généralement de plusieurs informations :

- majeur
- mineur
- bug fixe
- qualificateur : permet de préciser une version antérieure à une release (exemple alpha-1, beta-1, rc-1, ...). Une version avec qualificateur est plus récente qu'une version sans
- numéro de build : une version avec numéro de build est plus ancienne qu'une version sans

Exemple

1.2.10-20131112.2132121-1

Maven propose une syntaxe particulière pour désigner potentiellement plusieurs numéros de versions

Exemple :

[1.0,) : version 1.0 ou ultérieure

(,1.0] : version antérieure ou égale à 1.0

[1.0,1.2] : entre les versions 1.0 et 1.2 incluses

(,1.2),(1.2,) : toutes les versions sauf la 1.2

[1.0,2.0) : version supérieure ou égale à 1.0 et inférieure à 2.0

93.3.2.2. Convention plutôt que configuration

Maven met en oeuvre le principe de convention over configuration pour utiliser par défaut les mêmes conventions.

Par exemple, l'arborescence d'un projet Maven est par défaut imposée par Maven. Contrairement à d'autres outils comme Ant, l'arborescence de base de chaque projet Maven est toujours la même par défaut :

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

L'utilisation de ces conventions est un des points forts de Maven car elle permet aux développeurs de facilement être familiarisés avec la structure des projets qui est toujours la même.

Pour des besoins particuliers, il est possible de configurer une autre structure de répertoires mais cela n'est pas recommandé essentiellement car :

- La compréhension pour un nouveau développeur sur le projet est plus difficile
- Le fichier de configuration POM est plus complexe
- Il n'est généralement pas recommandé de sortir des standards

Maven propose aussi en standard d'autres conventions notamment :

- Chaque projet possède un artéfact principal par défaut
- Le nom des artéfacts est normalisé

93.3.2.3. Le cycle de vie et les phases

Maven 2 a standardisé le cycle de vie du projet en phases. Le cycle de vie par défaut de Maven contient plusieurs phases dont les principales sont : validate, compile, test, package, install et deploy.

Par défaut, aucun goal n'est associé aux phases du cycle de vie. En fonction du type de packaging du livrable du projet (jar, war, ejb, ejb3, ear, rar, par, pom, maven-plugin, ...) des goals différents sont associés aux différentes phases du cycle de vie Build de Maven. Le packaging par défaut est jar. Les goals exécutés pour chaque phase dépendent du type d'artéfact du projet : par exemple, la phase package exécute le goal jar:jar pour un artéfact de type jar, le goal war:war pour un artéfact de type war.

Il existe deux autres phases utiles :

- clean : effacer tous les éléments générés lors des exécutions précédentes
- site : générer un site web documentant le projet

Les phases et les goals peuvent être exécutés l'un après l'autre

Exemple :

```
1 mvn clean dependency:copy-dependencies package
```

Cette commande efface les fichiers générés, copie les dépendances et exécute les phases jusqu'à la phase package.

Le cycle de vie clean contient plusieurs phases :

- pre-clean
- clean
- post-clean

L'invocation d'une phase particulière du cycle de vie implique l'exécution de toutes les phases définies dans le cycle de vie qui la précède. Ainsi l'exécution de la commande mvn clean:clean exécutera les phases pre clean et clean.

Le goal clean:clean supprime tous les fichiers contenus dans le répertoire target où sont stockés les éléments produits par le build.

La phase pre clean peut permettre de réaliser des actions à exécuter avant la phase clean et la phase post clean peut permettre d'exécuter des actions après la phase clean.

93.3.2.4. Les archétypes

Un archétype est un modèle de projet. Maven propose en standard plusieurs archétypes dont les principaux sont :

Archétype	Description
maven-archetype-archetype	Un archétype pour un exemple d'archétype
maven-archetype-j2ee-simple	Un archétype pour un projet de type application J2EE simplifié
maven-archetype-mojo	Un archétype pour un exemple de plugin Maven
maven-archetype-plugin	Un archétype pour un projet de type plugin Maven
maven-archetype-plugin-site	Un archétype pour un exemple de site pour un plugin Maven
maven-archetype-portlet	Un archétype pour un projet utilisant les portlets
maven-archetype-quickstart	Un archétype pour un exemple de projet Maven
maven-archetype-simple	Un archétype pour un simple projet Maven
maven-archetype-site	Un archétype pour un site Maven
maven-archetype-site-simple	Un archétype pour un site Maven simplifié
maven-archetype-webapp	Un archétype pour un projet de type application web

D'autres achétypes peuvent être proposés par des tiers. Il est aussi possible de développer ses propres archétypes.

93.3.2.5. La gestion des dépendances

Généralement un artéfact a besoin d'autres artéfacts qui sont alors désignés comme des dépendances présentant elles-mêmes des dépendances.

Une dépendance peut être optionnelle : elle n'est requise que si une fonctionnalité particulière est utilisée. C'est par exemple le cas pour Hibernate avec le pool de connections c3p0 ou l'implémentation du cache de second niveau.

Certaines dépendances ne sont utiles que pour certaines phases, par exemple lors de la phase de tests qui devrait être la seule à avoir besoin d'un framework pour les tests unitaires ou d'un framework pour le mocking.

Certains artéfacts possèdent un support de plusieurs implémentations : c'est par exemple le cas de commons-logging qui a besoin d'une implémentation d'un moteur de logging. Il va dynamiquement utiliser celui qui sera trouvé.

La gestion des dépendances de Maven repose sur plusieurs concepts :

- les dépôts : permet de stocker les artéfacts
- la portée : permet de préciser dans quel contexte une dépendance est utilisée
- la transitivité : permet de gérer les dépendances de dépendances
- l'héritage

93.3.2.6. Les dépôts (repositories)

Maven utilise la notion de référentiel ou dépôt (repository) pour stocker les dépendances et les plugins requis pour générer les projets.

Un dépôt contient un ensemble d'artéfacts qui peuvent être des livrables, des dépendances, des plugins, ... Ceci permet de centraliser ces éléments qui sont généralement utilisés dans plusieurs projets : c'est notamment le cas pour les plugins et les dépendances.

Maven distingue deux types de dépôts : local et distant (remote). Ces dépôts peuvent être gérés à plusieurs niveaux :

- dépôt central : il stocke des dépendances et les plugins utilisables par tout le monde car disponible sur le web; ce sont généralement des artéfacts open source
- dépôt local : il stocke une copie des dépendances et plugins requis par les projets à générer en local. Ces artéfacts sont soit téléchargés des dépôts centraux soit créés avec Maven
- un référentiel au niveau entreprise ou domaine : ce référentiel permet de limiter les dépendances et les plugins ainsi que leurs versions à celles qui sont utilisables en local. Il permet de gérer et de restreindre les dépendances pour un certain niveau (entreprise, domaine, service, équipe, projet ...). Son utilisation est requise lorsque le nombre de projets et leur complexité sont importants.

Maven utilise une structure de répertoires particulière pour organiser le contenu d'un référentiel et lui permettre de retrouver les éléments requis :

Chemin_referentiel/groupId/artifactId/version

Par exemple avec Junit 3.8.2, dont les identifiants sont :

<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.2</version>

La structure de répertoires dans le dépôt est :

\junit\junit\3.8.2

Le répertoire de la version contient au moins l'artefact et son POM mais il peut aussi éventuellement contenir d'autres fichiers liés contenant une archive avec les sources, la Javadoc, la valeur du message digest calculée avec SHA-1, ...

Maven recherche un élément dans un ordre précis dans les différents référentiels :

- tout d'abord dans le dépôt local
- puis un des dépôts distant configuré s'il n'est pas trouvé

Si un élément n'est pas trouvé dans le répertoire local, il sera téléchargé dans ce dernier à partir du premier dépôt distant dans lequel il est trouvé.

Par défaut, le dépôt local est contenu dans le sous-répertoire .m2\repository du répertoire personnel de l'utilisateur.

Maven gère un référentiel central qui contient les artéfacts qui peuvent être diffusés. Il existe plusieurs miroirs du référentiel central.

Il existe aussi des référentiels proposés par des tiers notamment pour diffuser leurs propres artéfacts. Certains artéfacts ne peuvent pas être inclus dans des référentiels publics pour des raisons de licences.

Il est possible d'utiliser un ou plusieurs référentiels au niveau entreprise qui permettent de stocker ses propres artéfacts ou les artéfacts dont la licence interdit le diffusion publique.

L'accès aux référentiels peut se faire en utilisant plusieurs protocoles : http, https, ftp, sftp, WebDAV, SCP, ...

93.3.2.7. La portée des dépendances

Toutes les dépendances ne doivent pas forcément être utilisées de la même manière dans le processus de build ou lors de l'exécution de l'artéfact.

Maven utilise la notion de portée (scope) pour préciser comment la dépendance sera utilisée.

Maven définit quatre portées pour les dépendances :

- compile : la dépendance est utilisable par toutes les phases et à l'exécution. C'est le scope par défaut
- provided : la dépendance est utilisée pour la compilation mais elle ne sera pas déployée car elle est considérée comme étant fournie par l'environnement d'exécution. C'est par exemple le cas des API fournies par un serveur d'applications
- runtime : la dépendance n'est pas utile pour la compilation mais elle est nécessaire à l'exécution. C'est par exemple le cas des pilotes JDBC
- test : la dépendance n'est utilisée que lors de la compilation et de l'exécution des tests. C'est par exemple le cas pour la bibliothèque utilisée pour les tests unitaires (JUnit ou TestNG pas exemple) ou pour les doublures (Easymock, Mockito, ...)

93.3.2.8. La transitivité des dépendances

Maven 2 prend en charge la gestion des dépendances transitives. Les dépendances transitives sont des dépendances requises par un artéfact, les dépendances de ces dépendances et ainsi de suite.

Exemple :

Exemple :

```
1  <dependency>
2    <groupId>org.hibernate</groupId>
3    <artifactId>hibernate</artifactId>
4    <version>3.3.2</version>
5    <scope>compile</scope>
6  </dependency>
```

Cette déclaration de dépendances indique que l'artéfact a besoin d'Hibernate. Les dépendances d'Hibernate seront déterminées par Maven en analysant son POM puis les POM des dépendances et ainsi de suite.

Les dépendances transitives engendrent parfois de petits soucis qu'il convient de régler. C'est notamment le cas lors de l'inclusion d'une version différente de celle souhaitée car cette version est obtenue par une ou plusieurs dépendances transitives ou l'inclusion de dépendances non utilisées.

La qualité des dépendances transitives est grandement liée à la déclaration des dépendances dans les fichiers POM des différents artéfacts concernés.

Pour supprimer une dépendance transitive, il est possible de déclarer l'artéfact concerné avec le scope «provided» ou de définir une exclusion dans la définition de la dépendance.

93.3.2.9. La communication sur le projet

Maven propose de générer le contenu d'un site web dont le but est de fournir des informations sur le projet.

Il est facile possible d'inclure dans le site :

- les informations générales sur le projet
- le rapport sur l'exécution des tests unitaires et de la couverture de tests
- le rapport sur l'exécution des outils de qualité statique de code (PMD, Checkstyle, ...)
- les dépendances du projet
- la Javadoc
- ...

93.3.2.10. Les plugins

Maven en lui-même n'est composé que d'un noyau très léger.

Toutes les fonctionnalités pour générer un projet sont sous la forme de plugins qui doivent être présents dans le référentiel local ou téléchargés lors de la première utilisation.

La déclaration et la configuration des plugins à utiliser se fait dans le fichier POM.

Certains plugins utilisés dans le cycle de vie par défaut n'ont pas besoin d'être définit explicitement dans le fichier POM, sauf si la configuration par défaut doit être modifiée.

Il est aussi possible de développer ses propres plugins.

Un plugin est un artéfact Maven : il est donc identifié par un groupId, un artifactId et un numéro de version (par défaut, la version la plus récente).

La personnalisation d'un projet se fait en utilisant et en configurant des plugins.

Exemple pour préciser au compilateur d'utiliser la version 1.5 de Java

Exemple :

```
...
<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>
...
```

Les plugins sont des dépendances qui sont stockées dans le dépôt local après avoir été téléchargées.

Le tag <configuration> permet de fournir des paramètres qui seront utilisés par le plugin.

Le plugin Maven help possède un goal «describe» qui permet d'afficher des informations sur un plugin.

Le plugin dont on souhaite obtenir la description doit être précisé grâce à la propriété plugin fournie à la JVM.

La propriété medium fournie à la JVM permet de demander un niveau d'information supplémentaire qui offre une description de chaque goal.

Résultat :

```
C:\>mvn help:describe -Dplugin=help
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'help'.
[INFO]
------------------------------------------------------------------------
[INFO] Building Maven Default Project
[INFO]    task-segment: [help:describe] (aggregator-style)
[INFO]
------------------------------------------------------------------------
[INFO] [help:describe {execution: default-cli}]
[INFO] org.apache.maven.plugins:maven-help-plugin:2.1.1

Name: Maven Help Plugin
Description: The Maven Help plugin
provides goals aimed at helping to make
  sense out of the build environment. It includes the ability to view the
  effective POM and settings files, after inheritance and active profiles
  have been applied, as well as a describe a particular plugin goal to
  give usage information.
Group Id: org.apache.maven.plugins
Artifact Id: maven-help-plugin
Version: 2.1.1
Goal Prefix: help

This plugin has 9 goals:
help:active-profiles
  Description: Displays a list of the profiles which are currently active
  for this build.

help:all-profiles
  Description: Displays a list of available profiles under the current
  project.
  Note: it will list all profiles for a project. If a profile comes up
  with a status inactive then there might be a need to set profile activation
  switches/property.

help:describe
  Description: Displays a list of the attributes for a Maven Plugin and/or
  goals (aka Mojo - Maven plain Old Java Object).

help:effective-pom
  Description: Displays the effective POM as an XML for this build, with
  the active profiles factored in.

help:effective-settings
  Description: Displays the calculated settings as XML for this project,
  given any profile enhancement and the inheritance of the global settings
  into the user-level settings.

help:evaluate
  Description: Evaluates Maven expressions given by the user in an
  interactive mode.


```

```
help:expressions
  Description: Displays the supported Plugin expressions used by Maven.

help:help
  Description: Display help information on maven-help-plugin.
   Call
      mvn help:help -Ddetail=true -Dgoal=<goal-name>
   to display parameter details.

help:system
  Description: Displays a list of the platform details like system
  properties and environment variables.

For more information, run 'mvn help:describe [...] -Ddetail'
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO]
------------------------------------------------------------------------
[INFO] Total time: 2 seconds
[INFO] Finished at: Tue Nov 25 08:22:23 CET 2012
[INFO] Final Memory: 6M/510M
[INFO]
------------------------------------------------------------------------
```

La valeur par défaut de la propriété medium est true : elle affiche une description de chacun des goals du plugin. La valeur false affiche simplement une description du plugin.

Résultat :

```
C:\>mvn help:describe -Dplugin=help -Dmedium=false
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'help'.
[INFO]
------------------------------------------------------------------------
[INFO] Building Maven Default Project
[INFO]    task-segment: [help:describe] (aggregator-style)
[INFO]
------------------------------------------------------------------------
[INFO] [help:describe {execution: default-cli}]
[INFO] org.apache.maven.plugins:maven-help-plugin:2.1.1
Name: Maven Help Plugin
Description: The Maven Help plugin
provides goals aimed at helping to make
 sense out of the build environment. It includes the ability to view the
 effective POM and settings files, after inheritance and active profiles
 have been applied, as well as a describe a particular plugin goal to give
 usage information.

Group Id: org.apache.maven.plugins
Artifact Id: maven-help-plugin
Version: 2.1.1
Goal Prefix: help
For more information, run 'mvn
help:describe [...] -Ddetail'
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO]
------------------------------------------------------------------------
[INFO] Total time: < 1 second
[INFO] Finished at: Tue Nov 25 08:27:10 CET 2012
[INFO] Final Memory: 6M/510M
[INFO]
```

```
35 ------------------------------------------------------------------------
```

La propriété full fournie à la JVM permet de demander un niveau d'information complet qui offre en plus la liste des paramètres de chaque goal.

Résultat :

```
1 C:\>mvn help:describe -Dplugin=help -Dfull=true
```

Si le plugin n'est pas dans le dépôt local, il est nécessaire de fournir toutes les informations le concernant (nom, version)

Résultat :

```
1 C:\>mvn help:describe -Dplugin=org.apache.maven.plugins:maven-help-plugin:2.1.1
2 -Dmedium=false
```

La configuration d'un plugin se fait dans le fichier POM dans le tag <project><plugin><executions><configuration> pour toutes les exécutions ou <project><plugin><executions><execution><configuration> pour une exécution particulière.

93.3.3. La création d'un nouveau projet

La création d'un nouveau projet Maven repose sur l'utilisation d'un archétype et du goal archetype:generate en utilisant la commande :

mvn archetype:generate options

Résultat :

```
1 mvn archetype:generate -DarchetypeGroupId=org.apache.maven.archetypes
2 -DarchetypeArtifactId=maven-archetype-quickstart
3 -DgroupId=com.jmdoudoux.test.monapp -DartifactId=monApplication -DinteractiveMode=f
```

Cette commande va créer un répertoire pour contenir le projet et un ensemble de sous-répertoires et de fichiers selon l'archétype utilisé. Parmi les fichiers générés, il y a notamment le fichier pom.xml.

Les sous-répertoires créés suivent la structure standard des répertoires d'un projet qui est définie par convention.

Le sous-répertoire src/main/java contient les sources du projet.

Le sous-répertoire src/test/java contient les sources des tests unitaires du projet.

Un archétype est packagé dans une archive de type jar qui contient des métadonnées et un ensemble de template Velocity.

Lors de la création du premier archétype, Maven va télécharger dans son dépôt local tous les plugins qui lui sont nécessaires.

Un archétype est un template de projet qui est combiné avec certaines informations fournies pour créer un projet Maven d'un type particulier.

Les informations sont fournies sous la forme de propriétés fournies à la JVM -Dpropriete=valeur

Il est préférable d'utiliser le goal generate plutôt que l'ancien goal create.

Les archétypes sont eux même des artéfacts : ils seront téléchargés d'un référentiel distant.

93.3.4. Le fichier POM

Le fichier POM (Project Object Model) contient la description du projet Maven. C'est un fichier XML nommé pom.xml. Il contient les informations nécessaires à la génération du projet : identification de l'artéfact, déclaration des dépendances, définition d'informations relatives au projet, ...

Le fichier POM doit être à la racine du répertoire du projet.

Le tag racine est le tag <project>

Exemple :

```
<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.jmdoudoux.test</groupId>
  <artifactId>MaWebApp</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Mon application web</name>
  <dependencies>
     <dependency>
       <groupId>junit</groupId>
       <artifactId>junit</artifactId>
       <version>4.7</version>
       <scope>test</scope>
     </dependency>
   </dependencies>
</project>
```

Le tag <projet> peut avoir plusieurs tags fils dont les principaux sont :

Tag	Rôle
<modelVersion>	Préciser la version du modèle de POM utilisée
<groupId>	Préciser le groupe ou l'organisation qui développe le projet. C'est une des clés utilisée pour identifier de manière unique le projet et ainsi éviter les conflits de noms
<artifactId>	Préciser la base du nom de l'artéfact du projet
<packaging>	Préciser le type d'artéfact généré par le projet (jar, war, ear, pom, ...). Le packaging définit aussi les différents goals qui seront exécutés durant le cycle de vie par défaut du projet. La valeur par défaut est jar
<version>	Préciser la version de l'artéfact généré par le projet. Le suffixe -SNAPSHOT indique une version en cours de développement
<name>	Préciser le nom du projet utilisé pour l'affichage
<description>	Préciser une description du projet
<url>	Préciser une url qui permet d'obtenir des informations sur le projet
<dependencies>	Définir l'ensemble des dépendances du projet
<dependency>	Déclarer une dépendance en utilisant plusieurs tags fils : <groupId>, <artifactId>, <version> et <scope>

Les informations d'un POM sont héritées d'un POM parent sauf le POM racine.

93.3.4.1. L'utilisation et la configuration des plugins

Maven possède de très nombreux plugins fournis par Maven ou par des tiers. Il est aussi possible d'écrire ses propres plugins.

Certains plugins sont utilisés par défaut par un goal : il est alors possible de changer certains de leurs paramètres d'exécution dans la section plugins.

Exemple :

```
...

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

L'exemple ci-dessus permet de préciser au plugin de compilation d'utiliser la version 5 de Java.

Pour utiliser un plugin, il faut l'associer à un cycle de vie en précisant la phase concernée dans un tag executions/execution/phase et le goal dans un tag executions/execution/goals/goal

Exemple :

```
...
  <build>
    ...
    <plugins>
      <plugin>
      <artifactId>maven-antrun-plugin</artifactId>
      <executions>
            <execution>
              <phase>generate-sources</phase>
              <configuration>
                <tasks>
                  <!-tache Ant-->
                </tasks>
              </configuration>
              <goals>
                <goal>run</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
     </plugins>
  </build>
...
```

L'exemple ci-dessus exécute une tâche Ant en utilisant le plugin maven-antrun-plugin dans le goal run de la phase generate-sources.

93.3.4.2. Les métadonnées du projet

Il est possible d'ajouter des métadonnées dans le fichier POM : celles-ci sont essentiellement utilisées pour la génération du site de documentation du projet mais certains autres plugins peuvent aussi les utiliser pour des besoins spécifiques.

Exemple :

```
<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>
  [...]
  <name>Nom du projet</name>
  <description>Description du projet</description>
  <inceptionYear>2012</inceptionYear>
  <organization>
    <name>Nom de l'entreprise</name>
    <url>Url de l'entreprise</url>
  </organization>
  <licenses>
    <license>
      <name>Nom de la licence</name>
      <url>L'url de la licence</url>
      <comments>Commentaire concernant la licence</comments>
      <distribution>Politique de redistribution</distribution>
    </license>
  </licenses>
  <url>Url du projet</url>

```

```
  <version>1.3.0-1.2-MAVEN2TEST</version>

  <developers>
    <developer>
      <name>Jean-Michel Doudoux</name>
      <id>jmdoudoux</id>
      <email>jean-michel.doudoux@wanadoo.fr</email>
      <timezone>+1</timezone>
      <organization>Mon entreprise</organization>
      <organizationUrl>http://www.jmdoudoux.fr/</organizationUrl>
      <roles>
        <role>Tech lead</role>
        <role>développeur</role>
      </roles>
    </developer>
    ...
  </developers>
  <contributors>
    <contributor>
      <name>John Doe</name>
      <email>john_doe@inconnu.fr</email>
      <timezone>+1</timezone>
      <organization>Autre entreprise</organization>
      <organizationUrl>http://www.autreentreprise.fr/</organizationUrl>
      <roles>
        <role>Développeur</role>
      </roles>
    </contributor>
    [...]
  </contributors>
...
</project>
```

93.3.5. Le cycle de vie d'un projet

La construction d'un projet Maven repose sur la notion de cycle de vie qui définit de manière claire les différentes étapes nommées phases.

Trois cycles de vie sont définis :

- default : il permet de générer et déployer l'artéfact du projet
- clean : il permet de nettoyer les fichiers générés du projet
- site : il permet de générer un site web pour accéder à la documentation du projet

Un cycle de vie est composé de phases qui constituent les étapes de la génération de l'artéfact.

Le cycle de vie par défaut de Maven (build) propose plusieurs phases par défaut :

Phase	Rôle
validate	Valider les informations nécessaires à la génération du projet
initialize	Initialiser la génération du projet
generate-sources	Générer les sources qui doivent être incluses dans la compilation
process-sources	Traiter des sources (par exemple, application d'un filtre)
generate-resources	Générer les ressources qui doivent être incluses dans l'artéfact
process-resources	Copier les ressources dans le répertoire target
compile	Compiler le code source du projet
process-classes	Traiter les fichiers class résultant de la compilation (par exemple, pour faire du bytecode enhancement)
generate-test-sources	Générer des sources qui doivent être incluses dans les tests
process-test-sources	Traiter les sources pour les tests (par exemple, application d'un filtre)
generate-test-resources	Générer des ressources qui doivent être incluses dans les tests
process-test-resources	Copier les ressources dans le répertoire test
test-compile	Compiler le code source des tests
process-test-classes	Effectuer des actions sur les classes compilées (par exemple, pour faire du bytecode enhancement)
test	Compiler et exécuter les tests unitaires automatisés. Ces tests ne doivent pas avoir besoin de la forme diffusable de l'artéfact ni de son déploiement dans un environnement
Prepare-package (à partir de Maven 2.1)	Effectuer des actions pour préparer la génération du package
package	Générer l'artéfact sous sa forme diffusable (jar, war, ear, ...)
pre-integration-test	Effectuer des actions avant l'exécution des tests d'intégration (par exemple configurer l'environnement d'exécution)
integration-test	
(à partir de Maven 3)	Compiler et exécuter les tests d'intégration automatisés dans un environnement dédié au besoin en effectuant un déploiement
post-integration-test	Effectuer des actions après l'exécution des tests d'intégration (par exemple faire du nettoyage dans l'environnement)
verify	Exécuter des contrôles de validation et de qualité
install	Installer l'artéfact dans le dépôt local pour qu'il puisse être utilisé comme dépendance d'autres projets
deploy	Déployer l'artéfact dans un environnement dédié et copier de l'artéfact dans le référentiel distant

Ces différentes phases sont exécutées séquentiellement de la première étape jusqu'à la phase demandée à Maven. Toutes les phases jusqu'à la phase demandée seront exécutées.

Exemple :

mvn deploy

mvn integration-test

Maven définit la notion de multi-modules pour permettre de définir un projet qui possède un ou plusieurs sous-projets.

Une phase est composée d'un ou plusieurs goals. Un goal est une tâche spécifique à la génération ou la gestion du projet. Un goal peut être déclaré dans une ou plusieurs phases.

Il est possible d'invoquer directement un goal même s'il n'est pas associé à une phase.

mvn dependency:copy-dependencies

Il est possible d'invoquer des phases et des goals : leur ordre d'exécution sera celui fournit en paramètre de la commande mvn

Exemple :

mvn clean dependency:copy-dependencies package

Dans l'exemple ci-dessus, le cycle de vie clean est exécutée, puis le goal dependency :copy-dependencies puis la phase package avec toutes les phases précédentes du cycle de vie par défaut.

Si le goal est définit dans plusieurs phases alors le goal de chacune des phases sera invoqué.

Lors de l'invocation de la demande de l'exécution d'une phase à Maven, Maven va exécuter toutes les phases précédentes du cycle de vie.

Les goals exécutés par chaque phases dépendent du packaging de l'artéfact à générer. Le packaging est précisé grâce à l'élément <packaging> du POM.

Le cycle de vie clean de Maven contient plusieurs phases :

Phase	Rôle
pre-clean	Exécuter des traitements avant de nettoyer le projet
clean	Supprimer tous les fichiers par le build précédent
post-clean	Exécuter des traitements pour terminer le nettoyage du projet

Le cycle de vie site de Maven contient plusieurs phases :

Phase	Rôle
pre-site	Exécuter des traitements avant la génération du site
site	Générer le contenu du site de documentation du projet
post-site	Exécuter des traitements après la génération du site
site-deploy	Déployer le site sur un serveur web

Il est fréquemment reproché à Maven de télécharger beaucoup de fichiers à partir de dépôts distant mais Maven doit obtenir toutes les dépendances des projets, les plugins et les dépendances de ces plugins.

Tous ces éléments téléchargés sont stockés dans le dépôt local pour n'avoir à faire cette opération de téléchargement qu'une seule fois tant que l'élément est contenu dans le dépôt local.

93.3.5.1. L'ajout d'un goal à une phase

Il est possible d'ajouter de nouveaux goals à exécuter dans une phase, notamment pour permettre la configuration et l'utilisation de plugins.

Un plugin peut proposer un ou plusieurs goals. Généralement, la documentation d'un plugin doit préciser dans quelle phase un goal peut être utilisé.

La configuration d'un goal s'effectue avec un tag <execution> fils des tags <plugins>, <plugin> et <executions>. Les goals configurés viennent s'ajouter aux goals définis par défaut dans la phase concernée.

Il est possible de demander l'exécution du goal plusieurs fois avec des configurations différentes.

Pour chaque exécution, il est possible :

- de définir un identifiant avec le tag <id>.
- de préciser la phase d'exécution avec le tag <phase>
- de préciser les goals à exécuter avec le tag <goals> qui contient autant de tag fils <goal> que nécessaire
- le tag <configuration> permet de configurer le goal

Il est possible d'ajouter des goals à une phase du cycle de vie pour permettre de personnaliser les traitements exécutés lors de la génération du projet Maven.

L'exemple ci-dessous demande l'exécution du goal antrun:run qui devra s'exécuter dans la phase compile du cycle de vie du projet.

Exemple :

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-antrun-plugin</artifactId>
      <version>1.1</version>
      <executions>
        <execution>
          <id>id.compile</id>
          <phase>compile</phase>
          <goals>
            <goal>run</goal>
          </goals>
          <configuration>
            <tasks>
              <echo>Phase de compilation</echo>
            </tasks>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

93.3.6. La configuration générale de Maven

La configuration générale de Maven peut se faire à différents niveaux :

- global à Maven dans le fichier settings.xml du sous-répertoire conf de l'installation de Maven
- au niveau de l'utilisateur : dans le fichier settings.xml du sous-répertoire .m2 dans le répertoire home de l'utilisateur

Il est possible de préciser un autre fichier de configuration au niveau de l'utilisateur en utilisant l'option -s et au niveau global en utilisant l'option -gs.

Le goal help:effective-settings permet d'avoir un affichage des settings du projet. Les informations affichées sont une combinaison du contenu du fichier des settings.xml global et spécifique à l'utilisateur.

Résultat :

```
1 mvn help:effective-settings
```

93.3.6.1. Le fichier settings.xml

Le fichier settings.xml contient la configuration globale de Maven. C'est un document XML dont le tag racine est le tag <settings>.

Plusieurs tags fils permettent de préciser les éléments de la configuration :

Tag	Rôle
localRepository	Préciser le chemin du dépôt local. Par défaut, c'est le sous-répertoire .m2/repository du répertoire de l'utilisateur
interactiveMode	Utiliser Maven en mode interactif pour lui permettre d'obtenir des informations lorsqu'elles sont requises. Par défaut : true.
offline	Utiliser Maven en mode offline, donc déconnecter du réseau. Par défaut : false.
pluginGroups	
proxies	Définir un ou plusieurs proxy. Chaque proxy est défini dans un tag <proxy>
servers	Fournir des informations d'authentification sur une ressource. Chaque ressource est définie dans un tag <server>
mirrors	Définir des dépôts distants qui sont des miroirs. Chaque miroir est défini dans un tag <mirror>
profiles	Définir des profiles qui pourront être activés. Chaque profile est défini dans un tag <profile>
ActiveProfiles	Définir la liste des profiles qui sont activés par défaut pour tous les builds

Dans le fichier de configuration .m2/settings.xml, les informations d'authentification sur des serveurs sont définies dans le tags <servers>. Chaque serveur possède un tag fils <server> qui possède plusieurs tags fils :

Tag	Rôle
<id>	identifiant du serveur
<user>	nom de l'utilisateur
<password>	mot de passe de l'utilisateur

La configuration des dépôts utilisables se fait dans un tag <mirrors>. Chaque dépôt est défini dans un tag <mirror> qui possède plusieurs tags fils :

Tag	Rôle
<id>	identifiant du dépôt
<name>	nom du dépôt
<url>	url du dépôt
<mirrorOf>	

Exemple :

```
        <mirror>
            <id>artifactory</id>
            <name>enterprise repository</name>
            <url>http://dev-server/artifactory/libs-releases/</url>
            <mirrorOf>central</mirrorOf>
        </mirror>
```

La configuration de proxys se fait dans un tag <proxies>. Chaque proxy est défini dans un tag <proxy> qui possède plusieurs tags fils :

Tag	Rôle
<id>	L'identifiant du proxy
<protocol>	Le protocole utilisé pour accéder au proxy
<active>	
<username>	Le nom de l'utilisateur
<password>	Le mot de passe de l'utilisateur
<host>	Le host name du proxy
<port>	Le port du proxy
<nonProxyHosts>	énbsp;

Exemple :

```
<proxies>
  <proxy>
    <id>proxy</id>
    <active>yes</active>
    <protocol>http</protocol>
    <username>xxx</username>
    <password>yyy</password>
    <host>proxy-web</host>
    <port>8080</port>
    <nonProxyHosts>localhost</nonProxyHosts>
  </proxy>
</proxies>
```

Dans un dépôt, les artéfacts sont stockés en utilisant les identifiants de l'artéfact.

Une structure de répertoires est utilisée pour organiser les artéfacts en utilisant le groupId de l'artéfact de manière similaire à celle utilisée par les packages Java.

Il est donc intéressant d'utiliser un nom de domaine inversé comme groupId.

Exemple

com.jmdoudoux.monapp

L'arborescence de l'artéfact contient aussi un sous-répertoire ayant pour nom l'artifactId.

Enfin un sous-répertoire est aussi créé pour chaque version : ce sous-répertoire contient l'artéfact et son POM. D'autres fichiers peuvent aussi être présents selon la configuration du projet (jar contenant les sources, javadoc, ...)

Le goal install-file du plugin maven-install permet d'ajouter un artéfact dans le dépôt local.

Résultat :

```
mvn install:install-file -DgroupId=group-id \
                         -DartifactId=artifact-id \
                         -Dversion=version \
                         -Dpackaging=packaging \
                         -Dfile=fichierAinstaller
```

Cette commande ajoute un nouvel artéfact dont le fichier est précisé par la propriété file dans une arborescence utilisant la valeur des propriétés groupId, artifactId et version.

Cette solution est pratique pour faire un test ou s'il n'y qu'un seul développeur sur le projet. Dans le cas contraire, il est plus intéressant d'utiliser un gestionnaire de dépôts dans lequel il faut ajouter l'artéfact.

93.3.6.2. L'utilisation d'un miroir du dépôt central

Le dépôt central de Maven par défaut est à l'url http://repo.maven.apache.org/maven2/

Il peut être utile d'utiliser un autre dépôt pour plusieurs raisons : il est plus prêt géographiquement, il possède des artéfacts qui ne sont pas disponibles dans le dépôt par défaut, ...

Des miroirs publics du dépôt par défaut sont listés dans un lien de la page http://maven.apache.org/guides/mini/guide-mirror-settings.html.

Il est aussi possible de créer un dépôt intermédiaire gérer par un gestionnaire d'artéfacts qui va contenir une copie des artéfacts requis. L'utilisation d'un gestionnaire d'artéfacts possède plusieurs avantages :

- limiter l'utilisation de la bande passante sur internet et ainsi améliorer les performances
- restreindre les artéfacts utilisés uniquement à ceux présents dans le dépôt

Pour remplacer ou ajouter un autre dépôt, il faut modifier le fichier de configuration settings.xml

Exemple :

```
<settings>
  ...
  <mirrors>
    <mirror>
      <id>ibiblio.org</id>
      <url>http://mirrors.ibiblio.org/pub/mirrors/maven2</url>
    </mirror>
  </mirrors>
  ...
</settings>
```

Chaque dépôt est défini dans un tag <mirror>.

Exemple :

```
<settings>
  ...
  <mirrors>
    <mirror>
      <id>UK</id>
      <name>UK Central</name>
      <url>http://uk.maven.org/maven2</url>
      <mirrorOf>central</mirrorOf>

```

```
    </mirror>
  </mirrors>
  ...
</settings>
```

La valeur central du tag <mirrorOf> permet de préciser que le dépôt est un miroir du dépôt central. Maven va alors l'utiliser de préférence au dépôt central.

93.3.7. La configuration du projet

La configuration du projet se fait dans le fichier POM du projet.

Comme Maven utilise l'héritage de fichiers POM, le goal help:effective-pom permet d'avoir un affichage du POM effectif du projet. Les informations affichées sont une combinaison du contenu du fichier POM, des POM parents et des profiles qui sont actifs.

Résultat :

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

93.3.7.1. La déclaration des dépendances

Le tag <dependencies> du fichier POM permet de déclarer les dépendances externes du projet que ce soit pour la compilation, les tests ou l'exécution.

Pour chaque dépendance, il est nécessaire de préciser le groupId, l'artifactId, la version et le scope. Les trois premières informations doivent correspondre à celles définies dans le POM de la dépendance.

Exemple :

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

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

Le scope permet de préciser dans quel cadre la dépendance va être utilisée lors de la génération du projet :

- compile : utilisée pour la compilation et l'exécution (elle sera incluse dans le livrable)
- test : utilisée pour la compilation et l'exécution des tests uniquement (ne sera pas incluse dans le livrable)

- runtime : utilisée pour l'exécution uniquement (ce sont généralement des bibliothèques)
- provided : utilisée pour la compilation mais la dépendance sera fournie par l'environnement d'exécution (elle ne sera pas incluse dans le livrable)

Maven recherche les dépendances dans la hiérarchie des dépôts en commençant par le dépôt local. Si elle n'est pas trouvée, elle est téléchargée d'un autre dépôt.

Il est possible d'ajouter un nouvel artéfact directement dans le dépôt local : il est nécessaire de créer au préalable le fichier POM correspondant. Ceci est nécessaire notamment pour les artéfacts fournis par des tiers.

Pour ajouter une dépendance, il est nécessaire de connaître le groupId, l'artifactId et la version de l'artéfact correspondant. Il existe plusieurs sites pour permettre de rechercher ces informations, par exemple :

http://search.maven.org/

http://mvnrepository.com/

http://www.ibiblio.org

Une fois les informations identifiant l'artéfact connues, il faut l'ajouter en utilisant un tag <dependency> fils du tag <dependencies>

Exemple :

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

<modelVersion>4.0.0</modelVersion>

<groupId>com.jmdoudoux.app</groupId>
  <artifactId>monapp</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
        <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.17</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
</project>
```

La gestion des dépendances transitives est une fonctionnalité intéressante de Maven. Elle permet à Maven de gérer lui-même les dépendances d'une dépendance.

Maven assure aussi la gestion des dépendances transitives en double.

La prise en charge des dépendances transitives par Maven facilite grandement le travail de gestion des dépendances. Ce mécanisme fonctionne cependant bien si les dépendances sont correctement configurées dans les fichiers pom.xml des artéfacts dépendants.

93.3.7.2. L'utilisation de profiles

Les profiles permettent de définir différentes configurations.

Il est possible de définir le profile par défaut dans le fichier de configuration settings.xml en précisant le nom du profile dans un tag <activeProfile> fils du tag <activeProfiles>.

Exemple :

```
...
      <profiles>
...
            <profile>
                  <id>dev</id>
                  <properties>
                        <ma.propriete>Ma valeur de dev</ma.propriete>
                  </properties>
            </profile>
            <profile>
                  <id>prod</id>
                  <properties>
                        <ma.propriete>Ma valeur de prod</ma.propriete>
                  </properties>
            </profile>
...
      </profiles>
...
      <activeProfiles>
            <activeProfile>dev</activeProfile>
      </activeProfiles>
...
```

Pour afficher le profile actif, il faut utiliser le goal help:active-profiles

Maven permet d'activer un profile selon la version du JDK utilisé. Dans le fichier settings.xml, il faut ajouter un tag <jdk> fils des tags profile/activation qui précise la version de Java concernée.

Exemple :

```
...

<profile>

<id>profile.java.5</id>
      <properties>
        <ma.propriete>Ma valeur pour Java 5</ma.propriete>
      </properties>
      <activation>
        <jdk>1.5</jdk>
      </activation>
    </profile>
    <profile>
      <id>profile.java.6</id>
      <properties>
        <ma.propriete>Ma valeur pour Java 6</ma.propriete>
      </properties>
      <activation>
        <jdk>1.6</jdk>
      </activation>
    </profile>
...
```

Les profiles peuvent être définis dans trois fichiers :

- settings.xml de la configuration globale (dans le sous-répertoire conf du répertoire d'installation de Maven) ou de la configuration de l'utilisateur (dans le sous-répertoire .m2 du répertoire home de l'utilisateur)
- pom.xml
- profiles.xml à la racine du projet

Exemple extrait du fichier settings.xml

Exemple :

```
...
  <profiles>
...
    <profile>
      <id>dev</id>
      <properties>
        <ma.propriete>Ma valeur de dev</ma.propriete>
      </properties>
    </profile>
    <profile>
      <id>prod</id>
      <properties>
        <ma.propriete>Ma valeur de prod</ma.propriete>
      </properties>
    </profile>
...
```

Exemple dans un fichier pom.xml

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.jmdoudoux.test.monapp</groupId>
  <artifactId>monApplication</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>monApplication</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <profiles>
    <profile>
      <id>dev</id>
      <properties>
        <ma.propriete>Ma valeur de dev</ma.propriete>
      </properties>
    </profile>
    <profile>
      <id>prod</id>
      <properties>
        <ma.propriete>Ma valeur de prod</ma.propriete>
      </properties>
    </profile>

```

```
34    </profiles>
    </project>
```

Il est possible de préciser un profile par défaut en utilisant la valeur true pour le tag <activeByDefault> fils du tag <activation>.

Exemple :

```
1   <profiles>
2     <profile>
3       <id>dev</id>
4       <activation>
5         <activeByDefault>true</activeByDefault>
6       </activation>
7       <properties>
8         <ma.propriete>Ma valeur de dev</ma.propriete>
9       </properties>
10    </profile>
11  </profiles>
```

Pour connaitre le profile actif dans un projet, il faut invoquer le goal active-profiles du plugin help.

Résultat :

```
1   C:\JM\monApplication>mvn help:active-profiles
2    [INFO] Scanning for projects...
3   [INFO] Searching repository for plugin with prefix: 'help'.
4   [INFO]
5   ------------------------------------------------------------------------
6   [INFO] Building monApplication
7   [INFO]    task-segment: [help:active-profiles] (aggregator-style)
8   [INFO]
9   ------------------------------------------------------------------------
10  [INFO] [help:active-profiles {execution: default-cli}]
11  [INFO]
12  Active Profiles for Project 'com.jmdoudoux.test.monapp:monApplication:jar:1.0-SNAPSI
13  The following profiles are active:  - dev (source: pom)
14  [INFO]
15  ------------------------------------------------------------------------
16  [INFO] BUILD SUCCESSFUL
17  [INFO] ------------------------------------------------------------------------
18  [INFO] Total time: < 1 second
19  [INFO] Finished at: Thu Nov 25 08:35:26 CET 2012
20  [INFO] Final Memory: 6M/510M
21  [INFO]
22  ------------------------------------------------------------------------
```

Il est possible d'activer un profile selon la présence d'une variable d'environnement de la JVM.

Le nom de cette propriété doit être précisé comme valeur du tag <name> fils du tag <activation> <property>.

Exemple :

```
1   <project xmlns="http://maven.apache.org/POM/4.0.0"
2    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4    http://maven.apache.org/xsd/maven-4.0.0.xsd">
5
6
```

```xml
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.jmdoudoux.test.monapp</groupId>
  <artifactId>monApplication</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>monApplication</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <profiles>
    <profile>
      <id>dev</id>
      <activation>
        <activeByDefault>true</activeByDefault>
        <property>
          <name>profile.dev</name>
        </property>
      </activation>
      <properties>
        <ma.propriete>Ma valeur de dev</ma.propriete>
      </properties>
    </profile>
    <profile>
      <id>prod</id>
      <activation>
        <property>
          <name>profile.prod</name>
        </property>
      </activation>
      <properties>
        <ma.propriete>Ma valeur de prod</ma.propriete>
      </properties>
    </profile>
  </profiles>
</project>
```

Résultat :

```
C:\JM\monApplication>mvn -Dprofile.prod help:active-profiles
 [INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'help'.
[INFO]
------------------------------------------------------------------------
[INFO] Building monApplication
[INFO]    task-segment: [help:active-profiles] (aggregator-style)
[INFO]
------------------------------------------------------------------------
[INFO] [help:active-profiles {execution: default-cli}]
[INFO]
Active Profiles for Project 'com.jmdoudoux.test.monapp:monApplication:jar:1.0-SNAPSI
The following profiles are active:  - prod (source: pom)
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO]
------------------------------------------------------------------------
[INFO] Total time: < 1 second
```

```
[INFO] Finished at: Thu Nov 25 08:43:52 CET 2012
[INFO] Final Memory: 6M/510M
[INFO]
------------------------------------------------------------------------
```

Il est possible :

- d'activer plusieurs profiles en définissant chacune des variables concernées.
- d'activer un profile selon la valeur d'une variable d'environnement de la JVM.

Le nom de la variable est précisé dans un tag <name> et la valeur dans un tag <value>.

Exemple :

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

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.jmdoudoux.test.monapp</groupId>
  <artifactId>monApplication</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>monApplication</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <profiles>
    <profile>
      <id>dev</id>
      <activation>
        <activeByDefault>true</activeByDefault>
        <property>
          <name>profile</name>
          <value>dev</value>
        </property>
      </activation>
      <properties>
        <ma.propriete>Ma valeur de dev</ma.propriete>
      </properties>
    </profile>
    <profile>
      <id>prod</id>
      <activation>
        <property>
          <name>profile</name>
          <value>prod</value>
        </property>
      </activation>
      <properties>
        <ma.propriete>Ma valeur de prod</ma.propriete>
      </properties>
    </profile>
  </profiles>
</project>
```

Résultat :

```
C:\JM\monApplication>mvn -Dprofile=prod help:active-profiles
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'help'.
[INFO]
------------------------------------------------------------------------
[INFO] Building monApplication
[INFO]    task-segment: [help:active-profiles] (aggregator-style)
[INFO]
------------------------------------------------------------------------
[INFO] [help:active-profiles {execution: default-cli}]
[INFO]
Active Profiles for Project 'com.jmdoudoux.test.monapp:monApplication:jar:1.0-SNAPSH
The following profiles are active:  - prod (source: pom)
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO]
------------------------------------------------------------------------
[INFO] Total time: < 1 second
[INFO] Finished at: Thu Nov 25 08:51:08 CET 2012
[INFO] Final Memory: 6M/510M
[INFO]
------------------------------------------------------------------------
```

93.3.7.3. L'utilisation des propriétés

Dans le fichier POM, le tag <properties> permet de définir des propriétés. Chaque propriété est définie avec son propre tag. Le nom de la propriété correspond au nom du tag et la valeur est fournie dans le corps du tag.

Maven permet l'accès à une propriété en utilisant la syntaxe ${nom_de_la_propriete}.

Exemple :

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

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.jmdoudoux.test.monapp</groupId>
  <artifactId>monApplication</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>monApplication</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <mapropriete>Message de test</mapropriete>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

```

```
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-antrun-plugin</artifactId>
        <version>1.1</version>
        <executions>
          <execution>
            <phase>validate</phase>
            <goals>
              <goal>run</goal>
            </goals>
            <configuration>
              <tasks>
                <echo>mapropriete = ${mapropriete}</echo>
              </tasks>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

Maven considère les variables d'environnement comme des propriétés dont le nom est préfixé par «env.».

Par exemple, pour accéder à la variable M2_HOME dans le POM, il faut utiliser la syntaxe ${env.M2_HOME}

Les éléments du fichier POM peuvent être lus comme des propriétés. Le nom de ces propriétés commence par «project.»

Exemple : ${project.artifactId}

Les éléments du fichier settings peuvent être lus comme des propriétés. Le nom de ces propriétés commence par «settings.»

Exemple : ${settings.localRepository}

Les propriétés permettent aussi un accès aux variables d'environnement système et aux variables de la JVM.

Les propriétés peuvent être utilisées dans tout le POM.

Les propriétés peuvent aussi être utilisées dans les fichiers source (src/main/java) et les fichiers de ressources (src/main/resources). Les expressions correspondantes sont évaluées durant les phases process-sources et process-resources du cycle de vie par défaut de Maven.

93.3.7.4. L'ajout et l'exclusion de ressources dans l'artéfact

Il est fréquent de devoir ajouter des ressources dans un artéfact : fichiers de configuration, fichiers pour l'internationalisation, descripteurs, ...

Pour cela, Maven utilise sa structure standard de répertoires pour un projet : les ressources doivent être placées dans le sous-répertoire src/main/ressources de l'arborescence du projet. Tous les éléments (fichiers et sous-répertoires) contenus dans ce répertoire seront automatiquement ajoutés par Maven par le livrable de l'artéfact en conservant la sous-arborescence.

Résultat :

```
mon_app
|-- pom.xml
`-- src
```

```
    |-- main
        |-- java
        |   `-- com
        |       `-- jmdoudoux
        |           `-- app
        |               `-- MonApp.java
        `-- resources
            `-- META-INF
                `-- application.properties
```

Le livrable généré contient le sous-répertoire META-INF qui contient lui-même le fichier application.properties.

Résultat :

```
|-- META-INF
|   |-- MANIFEST.MF
|   |-- application.properties
|   `-- maven
|       `-- com.jmdoudoux.app
|           `-- mon_app
|               |-- pom.properties
|               `-- pom.xml
`-- com
    `-- jmdoudoux
        `-- app
            `-- MonApp.class
```

Par défaut, lors de la génération d'un artéfact de type jar, Maven rajoute par défaut deux fichiers dans le sous-répertoire META-INF :

- pom.xml
- pom.properties

Les méta-données contenues dans le fichier pom.properties peuvent par exemple permettre d'obtenir le numéro de version de l'artéfact.

Résultat :

```
#Generated by Maven
#Wed May 09 14:15:24 CEST 2012
version=1.0-SNAPSHOT
groupId=com.jmdoudoux.monapp
artifactId=mon_app
```

Maven créée un fichier META-INF/manifest par défaut si aucun n'est explicitement fourni dans les ressources du projet.

Résultat :

```
Manifest-Version: 1.0
Archiver-Version:
Plexus Archiver
Created-By: Apache Maven
Built-By: jm
Build-Jdk: 1.6.0_27
```

Il est aussi possible d'utiliser des ressources spécifiques aux tests unitaires. Pour ajouter ces ressources au

classpath pour l'exécution des tests unitaires, il faut les mettre dans le sous-répertoire /src/test/ressources du répertoire du projet. Dans les tests unitaires, il est alors possible de lire une ressource en demandant un flux au classloader.

Il est possible d'exclure certaines ressources en les définissant dans le tag <ressource>.

Le tag fils <directory> permet de préciser le répertoire concerné.

Le tag <excludes> permet de préciser des motifs pour sélectionner les fichiers à exclure. Chaque motif est précisé dans un tag <exclude>.

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.jmdoudoux.test.monapp</groupId>
  <artifactId>monApplication</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>monApplication</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
        <excludes>
          <exclude>**/*.txt</exclude>
        </excludes>
      </resource>
   </resources>
  </build>
</project>
```

93.3.7.5. La compilation d'un projet pour une version particulière de Java

Le maven-compiler-plugin permet de compiler le code source du projet et il est possible de le configurer pour utiliser une version particulière de Java.

Le tag <source> permet de préciser la version de Java avec laquelle le code source est compatible. Le tag <target> permet de préciser la version de Java qui sera utilisée pour générer le bytecode.

Exemple :

```
<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.jmdoudoux.test</groupId>
  <artifactId>MaWebApp</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Mon application web</name>
  <dependencies>
      <dependency>
         <groupId>junit</groupId>
         <artifactId>junit</artifactId>
         <version>4.7</version>
         <scope>test</scope>
      </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.6</source>
          <target>1.6</target>
        </configuration>
      </plugin>
   </plugins>
 </build>
</project>
```

93.3.7.6. La gestion des versions des dépendances dans un POM parent

Le tag <dependencyManagement> d'un POM parent permet de gérer les numéros de versions des dépendances des projets fils.

Si une dépendance est définie dans la partie dependencyManagement d'un POM parent, alors les POM des projets fils n'ont pas l'obligation d'utiliser cette dépendance mais si c'est le cas, il n'est pas obligatoire de préciser la version de cette dépendance. Dans ce cas, c'est la version du POM parent qui sera utilisée.

Exemple :

```
<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.jmdoudoux.test.maven</groupId>
  <artifactId>momapp</artifactId>
  <packaging>pom</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>monapp</name>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>commons-lang</groupId>

```

```
25          <artifactId>commons-lang</artifactId>
26          <version>2.3</version>
27        </dependency>
28      </dependencies>
29    </dependencyManagement>
30    <modules>
31      <module>../monapp_service</module>
32      <module>../monapp_persistance</module>
33    </modules>
  </project>
```

Dans le fichier POM d'un module, il n'est pas utile de préciser le numéro de version de la dépendance qui est déjà configuré dans la partie dependencyManagement du POM du projet parent.

Exemple :

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0"
2    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4    http://maven.apache.org/maven-v4_0_0.xsd">
5    <parent>
6      <groupId>com.jmdoudoux.test.maven</groupId>
7      <artifactId>monapp_service</artifactId>
8      <version>1.0-SNAPSHOT</version>
9      <relativePath>../monapp/pom.xml</relativePath>
10   </parent>
11
12   <modelVersion>4.0.0</modelVersion>
13   <groupId>com.jmdoudoux.test.maven</groupId>
14   <artifactId>monapp_service</artifactId>
15   <packaging>jar</packaging>
16   <version>1.0-SNAPSHOT</version>
17   <name>monapp_service</name>
18   <dependencies>
19     <dependency>
20       <groupId>commons-lang</groupId>
21       <artifactId>commons-lang</artifactId>
22       <scope>compile</scope>
23     </dependency>
24   </dependencies>
25 </project>
```

L'utilisation du dependencyManagement permet de centraliser les numéros de versions des dépendances à un endroit plutôt que d'avoir à répéter cette information dans plusieurs projets. Elle est particulièrement utile lors de l'utilisation de projets multi-modules.

93.3.7.7. La définition d'un fichier manifest particulier dans un jar

Par défaut, le fichier src/main/resources/META-INF/MANIFEST.MF n'est pas inclus dans l'artéfact de type jar généré.

Pour assurer sa prise en compte, il faut configurer le plugin maven-jar-plugin dans le fichier POM.

Exemple :

```
1  <project xmlns="http://maven.apache.org/POM/4.0.0"
2    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4
```

```
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.jmdoudoux.test.monapp</groupId>
    <artifactId>monApplication</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>monApplication</name>
    <url>http://maven.apache.org</url>
    <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      <mapropriete>Message de test</mapropriete>
    </properties>
    <dependencies>
      <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>3.8.1</version>
        <scope>test</scope>
      </dependency>
    </dependencies>
      <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-jar-plugin</artifactId>
          <configuration>
            <archive>
              <manifestFile>src/main/resources/META-INF/MANIFEST.MF</manifestFile>
            </archive>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </project>
```

Le fichier ne doit contenir que les propriétés spécifiques à l'artéfact.

Résultat :

```
MaPropriete: MaValeur
```

Le fichier META-INF/MANIFEST.MF dans le fichier jar généré contient la partie générée par défaut et le contenu du fichier MANIFEST.MF contenu dans le sous-répertoire resources.

Résultat :

```
Manifest-Version: 1.0
Archiver-Version:
Plexus Archiver
Created-By: Apache Maven
Built-By: JM
Build-Jdk: 1.6.0_34
MaPropriete: MaValeur
```

Pour définir la classe principale à exécuter dans le fichier manifest, il faut préciser la classe dans un tag fils <mainClass> dans la configuration du plugin maven-jar-plugin.

Exemple :

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <configuration>
          <archive>
            <manifest>
              <mainClass>com.jmdoudoux.test.monapp.App</mainClass>
            </manifest>
          </archive>
        </configuration>
      </plugin>
```

Lors de la génération de l'artéfact, la classe principale sera précisée dans le fichier manifest.

Résultat :

```
Manifest-Version: 1.0
Archiver-Version:
Plexus Archiver
Created-By: Apache Maven
Built-By: JM
Build-Jdk: 1.6.0_34
Main-Class: com.jmdoudoux.test.monapp.App
```

93.3.7.8. L'utilisation de valeurs de propriétés dans les ressources

Les fichiers de ressources peuvent contenir des références sur des propriétés en utilisant la syntaxe ${...}.

Exemple : le fichier /src/main/resources/test.txt

Résultat :

```
artifactId: ${project.artifactId}
```

Par défaut, lors de la génération Maven ne fait pas la résolution des valeurs de la propriété. Il est possible de demander l'application d'un filtre qui va réaliser cette résolution en utilisant le tag <filtering> avec la valeur true.

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.jmdoudoux.test.monapp</groupId>
  <artifactId>monApplication</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>monApplication</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <mapropriete>Message de test</mapropriete>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
```

```
19        <artifactId>junit</artifactId>
20        <version>3.8.1</version>
21        <scope>test</scope>
22      </dependency>
23    </dependencies>
24
25    <build>
26      <resources>
27        <resource>
28          <directory>src/main/resources</directory>
29          <filtering>true</filtering>
30        </resource>
31      </resources>
32    </build>
33  </project>
```

Dans l'artéfact généré, la propriété est remplacée par sa valeur lors de la phase process-resources du cycle de vie par défaut du projet.

Exemple :

```
1  artifactId: monApplication
```

Les valeurs des propriétés peuvent être extraites d'un fichier de properties qui va contenir la définition des valeurs des différentes propriétés à utiliser.

Exemple : le fichier src/main/filters/mesproprietes.properties

Exemple :

```
1  ma.propriete=Ma valeur
```

Les valeurs contenues dans le fichier de properties peuvent faire référence à des propriétés de la JVM en utilisant la syntaxe ${...}.

Résultat :

```
1  ma.propriete=${ma.propriete}
```

Les propriétés peuvent être soit celles standard de la JVM soit des propriétés définies au lancement de la JVM.

Résultat :

```
1  mvn clean package "-Dma.propriete.prop=Ma valeur"
```

Dans les fichiers de ressources, il faut utiliser la syntaxe ${...} pour représenter la valeur d'une propriété.

Exemple : le fichier src/main/resources/test.txt

Résultat :

```
1  valeur: ${ma.propriete}
```

Pour permettre de tenir compte du fichier contenant les propriétés, il faut appliquer une configuration particulière dans le POM:

- Affecter la valeur true dans le tag build/resources/resource/filtering
- Fournir le chemin du fichier properties dans le tag build/filters/filter
- Fournir le chemin du répertoire des ressources dans le tag build/resources/resource/directory : il faut repréciser le chemin puisque la configuration n'est pas celle par défaut

Exemple :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.jmdoudoux.test.monapp</groupId>
  <artifactId>monApplication</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>monApplication</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <mapropriete>Message de test</mapropriete>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <filters>
      <filter>src/main/filters/mesproprietes.properties</filter>
    </filters>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
        <filtering>true</filtering>
      </resource>
    </resources>
  </build>
</project>
```

Lors de la génération de l'artéfact,

Résultat :

```
valeur: Ma valeur
```

93.3.7.9. La génération d'un fichier jar contenant les sources du projet

Le plugin maven-source-plugin permet de générer un fichier jar contenant le code source du projet.

Le déploiement de ce fichier dans un dépôt distant peut permettre aux autres développeurs d'attacher le code source de l'artéfact dans leur IDE.

Le goal jar est par défaut exécuté dans la phase package du cycle de vie par défaut du projet.

Exemple :

```
<build>
  <plugins>
...

    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-source-plugin</artifactId>
      <executions>
        <execution>
          <id>attacher-sources</id>
          <goals>
            <goal>jar</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
...
  </plugins>
</build>
```

93.3.8. L'exécution de commandes

Une fois la configuration du projet effectuée, il faut demander à Maven d'effectuer certaines tâches selon les besoins en lui passant en paramètres des commandes.

Le nombre de ces commandes est important et dépend des plugins utilisés.

93.3.8.1. Un résumé des principales commandes

mvn package	Construire le projet pour générer l'artéfact
mvn site	Générer le site de documentation dans le répertoire target/site
mvn -Dtest=<unqualified-classname> test	Exécuter le test unitaire dont le nom de la classe est fourni
mvn -Dmaven.surefire.debug test	Exécuter des tests avec un débogueur distant en utilisant le port 5005. Surefire attend la connexion du débogueur
mvn help:effective-pom	Afficher le contenu du pom en incluant les POM hérités
mvn dependency:tree	Afficher une arborescence des dépendances, incluant les dépendances transitives
mvn clean	Supprimer les fichiers générés par les précédentes générations
mvn clean package	Supprimer les fichiers générés par les précédentes générations et construire le projet pour générer l'artéfact
mvn install	Générer l'artéfact et le déployer dans le dépôt local
mvn -Dmaven.test.failure.ignore=true package	Construire le projet pour générer l'artéfact même si les tests unitaires échouent
mvn eclipse:eclipse	Générer des fichiers de configuration Eclipse à partir du POM (notamment les dépendances)
mvn eclispe:clean eclipse:eclipse	Idem mais les fichiers précédemment générés sont supprimés
mvn -Dmaven.test.skip=true clean package	Supprimer les fichiers générés par les précédentes générations et construire le projet pour générer l'artéfact sans exécuter les tests unitaires
mvn javadoc:javadoc	Générer la Javadoc
mvn javadoc:jar	Générer la Javadoc dans un fichier jar
mvn --version	Afficher les informations sur la version de Maven
mvn test	Exécuter les tests unitaires
mvn compile	Compiler les sources du projet
mvn idea:idea	Générer des fichiers de configuration IntelliJ à partir du POM (notamment les dépendances)
mvn release:prepare release:perform	Livrer l'artéfact (créer un tag dans le gestionnaire de sources et supprimer SNAPSHOT dans la version)
mvn archetype:create -DgroupId=com.jmdoudoux.test -DartifactId=monAppli	Créer un nouveau projet en mode interactif

93.3.8.2. Les options de la commande mvn

Les principales options de la commande mvn sont :

Option	Rôle
-o,--offline	Mode offline
-h, --help, --usage	Afficher la liste des options utilisables
-X,--debug	Afficher les traces de debug
-v,--version	Afficher des informations sur la version de Maven
-V,--show-version	Afficher des informations sur la version de Maven sans arrêter le build
-s,--settings	Préciser un chemin alternatif pour le fichier de configuration settings.xml
-f,--file	Préciser un fichier POM alternatif
-D,--define	Définir une propriété
-B,--batch-mode	Demander l'exécution en mode batch (pas d'interaction avec l'utilisateur)
-fn,--fail-never	Demander à ce que le build n'échoue pas
-C,--strict-checksums	Demander un échec du build si un checksum ne correspond pas
-c,--lax-checksums	Demander une alerte si un checksum ne correspond pas
-P,--activate-profiles	Préciser la liste des profiles actifs
-up,--update-pluging	
-cpu,--check-plugin-updates	Demander de vérifier les mises à jour des plugins
-fae,--fail-at-end	Demander un échec du build le plus tardif possible : Maven tente d'exécuter tous les goals possibles
-npu,--no-plugin-updates	Demander de ne pas vérifier les mises à jour des plugins
--non-recursive	

93.3.8.3. Le nettoyage d'un projet

Le cycle de vie clean possède trois phases :

Phase	Rôle
pre-clean	Effectuer des tâches avant le nettoyage
clean	Supprimer les fichiers générés dans le répertoire target
post-clean	Effectuer des tâches après le nettoyage

Pour faire le ménage dans les fichiers générés, il faut invoquer la commande

Exemple :

```
1 mvn clean
```

C'est généralement une bonne pratique d'invoquer le cycle de vie clean avant d'invoquer la phase install

du cycle de vie par défaut car demander un nettoyage implique une construction complète du projet.

Exemple :

```
1 mvn clean install
```

93.3.8.4. La compilation du code source

Pour demander la compilation des sources du projet, il faut demander l'exécution de la phase compile du cycle de vie par défaut.

Exemple :

```
1 mvn compile
```

Lors de la première exécution d'une commande maven, cette dernière doit télécharger le plugin correspondant et ses dépendances dans le dépôt local. Les exécutions suivantes utiliseront la version stockée dans le dépôt local.

Par convention, le résultat de la compilation des classes est stocké dans le sous-répertoire target/classes du projet.

93.3.8.5. La compilation du code source des tests et leur exécution

Pour demander l'exécution des tests unitaire du projet, il faut demander l'exécution de la phase test du cycle de vie par défaut.

Exemple :

```
1 mvn test
```

Avant de compiler et d'exécuter les tests, Maven effectue une compilation du code source de l'artéfact.

L'exécution des tests par Maven se fait grâce au plugin Surefire.

Par défaut, le plugin Surefire recherche les tests unitaires dont les classes respectent une convention de nommage particulière :

**/*Test.java
**/Test*.java
**/*TestCase.java

Pour demander uniquement la compilation des tests mais pas leur exécution, il faut invoquer la commande Maven

Exemple :

```
1 mvn test-compile
```

Même si cela n'est pas une bonne pratique, il est parfois utile de désactiver l'exécution des tests durant la génération du projet.

Il suffit d'utiliser l'option -DskipTests=true dans les paramètres de lancement de la commande mvn.

Pour désactiver définitivement l'exécution des tests, il faut modifier la configuration du plug-in Surefire de

Maven dans le fichier pom.xml

Exemple :

```
...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-surefire-plugin</artifactId>
        <configuration>
          <skipTests>true</skipTests>
        </configuration>
      </plugin>
    </plugins>
  </build>
...
```

93.3.8.6. La création de l'artéfact

Pour demander la génération d'un artéfact, il faut demander l'exécution de la phase package du cycle de vie par défaut.

Résultat :

```
mvn package
```

L'artéfact généré est mis dans le sous-répertoire target du répertoire du projet.

93.3.8.7. L'installation de l'artéfact dans le dépôt local

Maven utilise la notion de dépôt pour stocker les artéfacts générés ou requis ainsi que les métadonnées qui leur sont associées.

Un artéfact Maven est un élément packagé livrable : par exemple, un fichier jar, war, ear, ...

Le dépôt local est, par défaut, dans le sous-répertoire .m2/repository du répertoire home de l'utilisateur.

Pour demander le déploiement d'un artéfact dans le dépôt local, il faut exécuter la commande :

Résultat :

```
mvn install
```

Maven travaille obligatoirement avec un dépôt local dans lequel il stocke les dépendances, les plugins, les artéfacts pour éviter d'avoir à toujours les télécharger d'internet.

L'installation d'un artéfact dans le dépôt local permet de l'utiliser comme dépendance dans d'autre projet.

Le dépôt Maven central est un dépôt distant qui contient un grand nombre d'artéfacts communs. Il est existe aussi des dépôts miroir.

Il est aussi possible d'utiliser des gestionnaires d'artéfacts, comme Archiva ou Artifactory, qui vont servir de dépôts intermédiaires pour limiter la quantité d'artéfacts téléchargés et restreindre l'utilisation d'artéfacts uniquement à ceux présents dans ces dépôts.

93.3.8.8. Le déploiement d'un artéfact dans un dépôt distant

Pour déployer un artéfact dans un dépôt distant, il faut configurer l'url du dépôt dans le fichier POM et les informations d'authentification dans le fichier settings.xml.

La configuration dans le fichier POM se fait dans un tag <distributionManagement>.

Chaque dépôt est configuré dans un tag <repository>

Exemple :

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.jmdoudoux.app</groupId>
  <artifactId>monapp</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <distributionManagement>
    <repository>
      <id>depot-entreprise</id>
      <name>Depot d'entreprise</name>
      <url>scp://depot.entreprise.com/repository/maven2</url>
    </repository>
  </distributionManagement>
</project>
```

Le tag <repository> possède plusieurs tags fils :

Tag	Rôle
<id>	permet de faire référence au serveur défini dans le fichier .m2/settings.xml
<name>	permet de fournir un nom au serveur
<url>	permet de fournir une url utilisable par le plugin Wagon (commence généralement par scp:)

Le tag <snapshotRepository> permet de définir un dépôt qui sera utilisé pour stocker les snapshots.

Si cette configuration est commune à plusieurs projets, il est possible de définir cette configuration dans un POM parent.

Pour demander le déploiement, il suffit d'invoquer la commande mvn deploy.

93.3.8.9. La génération de la Javadoc

Le plugin maven-javadoc-plugin permet de générer la javadoc d'un artéfact et de la déployer dans un dépôt.

Il faut l'activer en précisant le goal «jar».

Exemple :

```
...
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-javadoc-plugin</artifactId>
      <executions>
        <execution>
          <id>attach-javadocs</id>
          <goals>
            <goal>jar</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
...
```

Il suffit alors d'invoquer la commande mvn clean package pour générer l'artéfact (xxx-1.0.jar), le source (xxx-1.0-sources.jar) et la javadoc(xxx-1.0-javadoc.jar). Pour générer ces éléments et les installer dans le dépôt, il suffit d'invoquer la commande mvn clean deploy.

93.3.9. L'utilisation de projets multi-modules

Maven 2.0 propose la possibilité d'utiliser des projets multi-modules.

Un projet multi-modules est un projet qui contient d'autres projets fils : c'est un regroupement logique de sous-projets.

Il contient un fichier pom.xml mais le projet ne crée pas lui-même d'artéfact.

Il est possible d'imbriquer des projets multi-modules.

Il est recommandé d'organiser les projets de manière hiérarchique : les sous-projets sont dans le projet multi-modules. Ce n'est pas une obligation mais cela facilite la compréhension et la mise en oeuvre car la hiérarchie des projets est directement reflétée dans la structure des répertoires. Cependant, ce n'est pas une obligation et une organisation des projets à plat est possible.

Résultat :

```
+-monAppli
  +- pom.xml
  +- monAppliIHM
  | +- pom.xml
  | +- src
  |   +- main
  |
+- java
  +- monAppliUtil
  | +- pom.xml
  | +- src
  |   +- main
  |
+- java
```

Le fichier POM du projet multi-module doit avoir le type de packaging pom

Exemple :

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.jmdoudoux.test</groupId>
  <artifactId>monAppli</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  <name>monAppli</name>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <modules>
    <module>monAppliIHM</module>
    <module>monAppliUtil</module>
  </modules>
</project>
```

Le chemin de chacun des modules est précisé dans un tag <module> fils du tag <modules>. Cela indique à Maven que les opérations ne vont pas être réalisées sur le projet mais sur les modules du projet.

Le POM de chacun des modules doit contenir un tag <parent> qui permet d'identifier le module parent.

Le POM du module monAppliIHM

Exemple :

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.jmdoudoux.test</groupId>
    <artifactId>monAppli</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <groupId>com.jmdoudoux.test.monappli.ihm</groupId>
  <artifactId>monAppliIHM</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>monAppliIHM</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>com.jmdoudoux.test.monappli.util</groupId>
      <artifactId>monAppliUtil</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
  </dependencies>
</project>
```

Le POM du projet monAppliUtil

Exemple :

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.jmdoudoux.test</groupId>
    <artifactId>monAppli</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <groupId>com.jmdoudoux.test.monappli.util</groupId>
  <artifactId>monAppliUtil</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>monAppliUtil</name>
  <url>http://maven.apache.org</url>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Exemple :

```java
package com.jmdoudoux.test.monappli.util;

public class AppUtil {
    public static String getMessage() {
        return "Hello World!";
    }
}
```

Si les modules ne sont pas dans des sous-répertoire mais dans des répertoires de même niveau que le projet parent, il est nécessaire de mettre «../» dans le chemin du module. C'est notamment le cas si les projets sont utilisés dans Eclipse. Il est aussi nécessaire dans le POM des modules d'utiliser le tag <relativePath> fils du tag <parent> pour préciser le chemin du POM parent qui devra lui aussi commencer par «../».

Résultat :

```
C:\TEMP\monAppli>mvn clean install

[INFO] Scanning for projects...
[INFO] Reactor build order:
[INFO]   monAppli
[INFO]   monAppliUtil
[INFO]   monAppliIHM
[INFO] ------------------------------------------------------------------------
[INFO] Building monAppli
```

```
[INFO]    task-segment: [clean, install]
[INFO] ------------------------------------------------------------------------
[INFO] [clean:clean {execution: default-clean}]
[INFO] [site:attach-descriptor {execution: default-attach-descriptor}]
[INFO] [install:install {execution: default-install}]
[INFO] Installing C:\TEMP\monAppli\pom.xml to
C:\java\maven_repository\com\jmdoudoux\test\monAppli\1.0-SNAPSHOT\monAppli-1.0-SNAP
[INFO] ------------------------------------------------------------------------
[INFO] Building monAppliUtil
[INFO]    task-segment: [clean, install]
[INFO] ------------------------------------------------------------------------
[INFO] [clean:clean {execution: default-clean}]
[INFO] Deleting directory C:\TEMP\monAppli\monAppliUtil\target
[INFO] [resources:resources {execution: default-resources}]
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory C:\TEMP\monAppli\monAppliUtil\src\main\r
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Compiling 1 source file to C:\TEMP\monAppli\monAppliUtil\target\classes
[INFO] [resources:testResources {execution: default-testResources}]
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory C:\TEMP\monAppli\monAppliUtil\src\test\r
[INFO] [compiler:testCompile {execution: default-testCompile}]
[INFO] Compiling 1 source file to C:\TEMP\monAppli\monAppliUtil\target\test-classes
[INFO] [surefire:test {execution: default-test}]
[INFO] Surefire report directory: C:\TEMP\monAppli\monAppliUtil\target\surefire-rep

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.jmdoudoux.test.monappli.util.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.016 sec

Results :

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

[INFO] [jar:jar {execution: default-jar}]
[INFO] Building jar: C:\TEMP\monAppli\monAppliUtil\target\monAppliUtil-1.0-SNAPSHOT
[INFO] [install:install {execution: default-install}]
[INFO] Installing C:\TEMP\monAppli\monAppliUtil\target\monAppliUtil-1.0-SNAPSHOT.ja
to C:\java\maven_repository\com\jmdoudoux\test\monappli\util\monAppliUtil\1.0-SNAPS
OT\monAppliUtil-1.0-SNAPSHOT.jar
[INFO] ------------------------------------------------------------------------
[INFO] Building monAppliIHM
[INFO]    task-segment: [clean, install]
[INFO] ------------------------------------------------------------------------
[INFO] [clean:clean {execution: default-clean}]
[INFO] Deleting directory C:\TEMP\monAppli\monAppliIHM\target
[INFO] [resources:resources {execution: default-resources}]
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory C:\TEMP\monAppli\monAppliIHM\src\main\re
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Compiling 1 source file to C:\TEMP\monAppli\monAppliIHM\target\classes
[INFO] [resources:testResources {execution: default-testResources}]
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory C:\TEMP\monAppli\monAppliIHM\src\test\re
[INFO] [compiler:testCompile {execution: default-testCompile}]
[INFO] Compiling 1 source file to C:\TEMP\monAppli\monAppliIHM\target\test-classes
[INFO] [surefire:test {execution: default-test}]
[INFO] Surefire report directory: C:\TEMP\monAppli\monAppliIHM\target\surefire-repo

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.jmdoudoux.test.monappli.ihm.AppTest
```

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.015 sec

Results :

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

[INFO] [jar:jar {execution: default-jar}]
[INFO] Building jar: C:\TEMP\monAppli\monAppliIHM\target\monAppliIHM-1.0-SNAPSHOT.ja
[INFO] [install:install {execution: default-install}]
[INFO] Installing C:\TEMP\monAppli\monAppliIHM\target\monAppliIHM-1.0-SNAPSHOT.jar
to C:\java\maven_repository\com\jmdoudoux\test\monappli\ihm\monAppliIHM\1.0-SNAPSHOT
onAppliIHM-1.0-SNAPSHOT.jar
[INFO]
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO] ------------------------------------------------------------------------
[INFO] monAppli ........................................... SUCCESS [1.344s]
[INFO] monAppliUtil ....................................... SUCCESS [1.765s]
[INFO] monAppliIHM ........................................ SUCCESS [0.859s]
[INFO] ------------------------------------------------------------------------
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 4 seconds
[INFO] Finished at: Tue Nov 25 09:25:51 CET 2012
[INFO] Final Memory: 15M/510M
[INFO] ------------------------------------------------------------------------
```

L'application contient une simple classe avec une méthode main() pour permettre son exécution.

Exemple :

```
package com.jmdoudoux.test.monappli.ihm;

import com.jmdoudoux.test.monappli.util.AppUtil;

public class App  {
    public static void main( String[] args ) {
        System.out.println( "Lancement MonAppli" );
        System.out.println( AppUtil.getMessage() );
    }
}
```

Tous les artéfacts doivent être dans le classpath.

Résultat :

```
C:\TEMP\monAppli>java -cp C:\TEMP\monAppli\monAppliIHM\target\monAppliIHM-1.0-SNAPSH
C:\TEMP\monAppli\monAppliUtil\target\monAppliUtil-1.0-SNAPSHOT.jar
com.jmdoudoux.test.monappli.ihm.App
Lancement MonAppli
Hello World!
```