Maven

(v2,v3)

Table des matières

I - Maven (présentation, positionnement)

1. Présentation de Maven

Apache Maven est un outil logiciel open source pour la gestion et l'automatisation de production des projets logiciels Java/JEE .

Maven utilise un paradigme connu sous le nom de ***Project Object Model*** **(POM)** afin de décrire un projet logiciel, ses dépendances avec des modules externes et l'ordre à suivre pour sa production. Il est livré avec un grand nombre de tâches pré-définies, comme la compilation de code Java .

Chaque projet ou sous-projet est configuré par un fichier ***`pom.xml`*** à la racine du projet qui contient les informations nécessaires à Maven pour traiter le projet (nom du projet, numéro de version, dépendances vers d'autres projets, bibliothèques nécessaires à la compilation, noms des contributeurs ...).

Maven impose une arborescence et un nommage des fichiers du projet selon le concept de ***Convention plutôt que configuration***. Ces conventions permettent de réduire la configuration des projets, tant qu'un projet suit les conventions. Si un projet a besoin de s'écarter de la convention, le développeur le précise dans la configuration du projet.

Voici une liste non-exhaustive des répertoires d'un projet Maven :

- `/src` : les sources du projet
- `/src/main` : code source et fichiers source principaux
- **`/src/main/java`** : code source
- **`/src/main/resources`** : fichiers de ressources (images, fichiers de configurations pour Spring , hibernate , log4j, ...)
- `/src/test` : fichiers de test
- `/src/test/java` : code source de test
- `/src/test/resources` : fichiers de ressources de test
- `/src/site` : informations sur le projet et/ou les rapports générés (checkstyle,javadoc)
- **`/src/webapp`** : webapp du projet
- `/target` : fichiers résultat, les binaires (du code et des tests), les packages générés et les résultats des tests

La gestion des dépendances au sein de Maven est simplifiée par les notions d'héritage et de **transitivité**. La déclaration de ces dépendances est alors limitée.

Les **buts** (***goals*** en anglais) principaux du cycle de vie d'un projet Maven sont:

- `compile` (compiler ".java" --> ".class")
- **`test`** (lancer tous les tests unitaires (JUnit) de la branche src/test/java)
- **`package`** (construire le ".jar" ou le ".war")
- `install` (installer la chose construite sur le référentiel local)
- **`deploy`** (déployer vers un référentiel distant ou sur un serveur)

NB: lancer "package" déclenche *compile* puis *test* puis *package* .
+

- clean (supprimer les ".jar" et .... générés)
- site (générer la documentation)

2. Principales fonctionnalités de maven

Maven (principales fonctionnalités)

- ***Centré sur la notion de projet*** (api java, module applicatif) --> toutes les caractéristiques d'un projet sont déclarées dans un fichier "***pom.xml***" (**P**roject **O**bject **M**odel).

- ***Configuration "déclarative"*** (basée sur des conventions) plutôt qu'explicite. Pas de commandes et chemins précis à renseigner (contrairement à ANT).

- ***Gestion distribuée*** (sur le web) des ***dépendances*** inter-projets. ---> identification et ***téléchargement automatique des ".jar"*** nécessaires selon les API déclarées en dépendances .

- Gère toutes les phases (compile/build , tests unitaires , packaging , stockage dans le référentiel , éventuel lien avec SVN, ...) .

3. Intérêts de Maven

Avec la technologie ***ANT*** , il faut tout **expliciter** dans les scripts (répertoires , listes des ".jar" du classpath , ...). A l'inverse , la technologie ***Maven*** automatise presque tout de manière **implicite** (en se basant sur des conventions de structuration des projets) .

Dans de nombreux projet Java/JEE on a besoin de faire collaborer/cohabiter tout un tas de technologies "open source" qui:

- proviennent de différents éditeurs (jakarta Apache , Jboss Group , SUN , Spring , ...)
- utilisent en interne différents modules d'intérêt général (commons-logging , ...)
- évoluent régulièrement (nouvelles versions pas toujours compatibles avec tout le reste)

La technologie Maven permet de gérer presque automatiquement tous ces problèmes de dépendances car elle s'appuie en interne sur une sorte de référentiel de produits/technologies qui est régulièrement actualisé sur le site web de référence de maven. [*Remarque*: Après une installation de maven sur un poste de développement, un accès internet est ainsi indispensable pour ré-actualiser certains modules]

Il est possible de configurer un nouveau référentiel maven au sein d'une entreprise pour ne plus dépendre des référentiels externes. Ce référentiel se créer comme une copie partielle du référentiel de référence (avec des actualisations régulières conseillées).

NB: il existe maintenant une extension pour ANT appelée "IVY" qui gère également les dépendances de librairies (".jar") , la syntaxe des fichiers de configuration d'IVY est cependant légèrement différente de celle de "maven". Maven est plus orienté "projet" et va plus loin.

4. Evolutions, versions

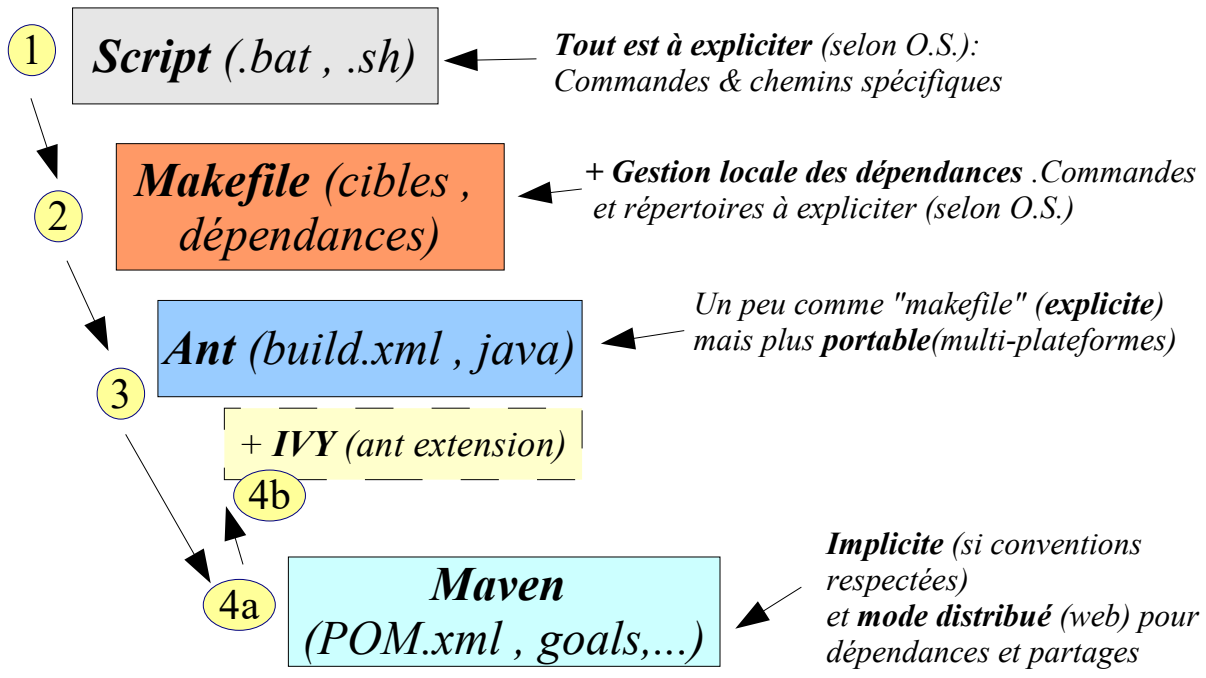
Versions de maven

- *Maven 1* (version assez ancienne , aujourd'hui obsolète)

- **Maven 2** (beaucoup d'améliorations , changements en profondeur)
  --> version mature (avec pleins de plugins disponibles)
  --> plugin eclipse "m2e" maintenant au point.

- ***Maven 3*** (depuis début 2011 , dans la continuité de la V2)
  --> même configuration que la V2 (syntaxe inchangée)
  --> restructuration interne permettant d'obtenir une ***nette amélioration des performances***.
  --> quelques ajouts (parallélisme , ...)

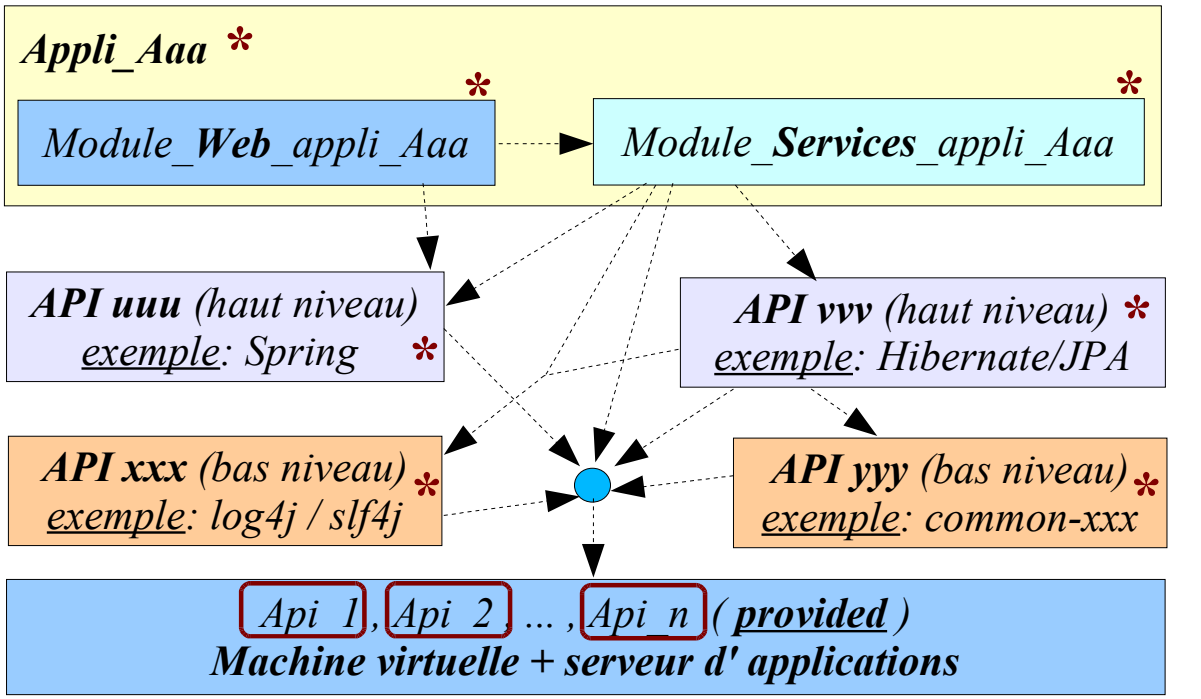
évolutions (du script à maven)



5. Fonctionnement de maven

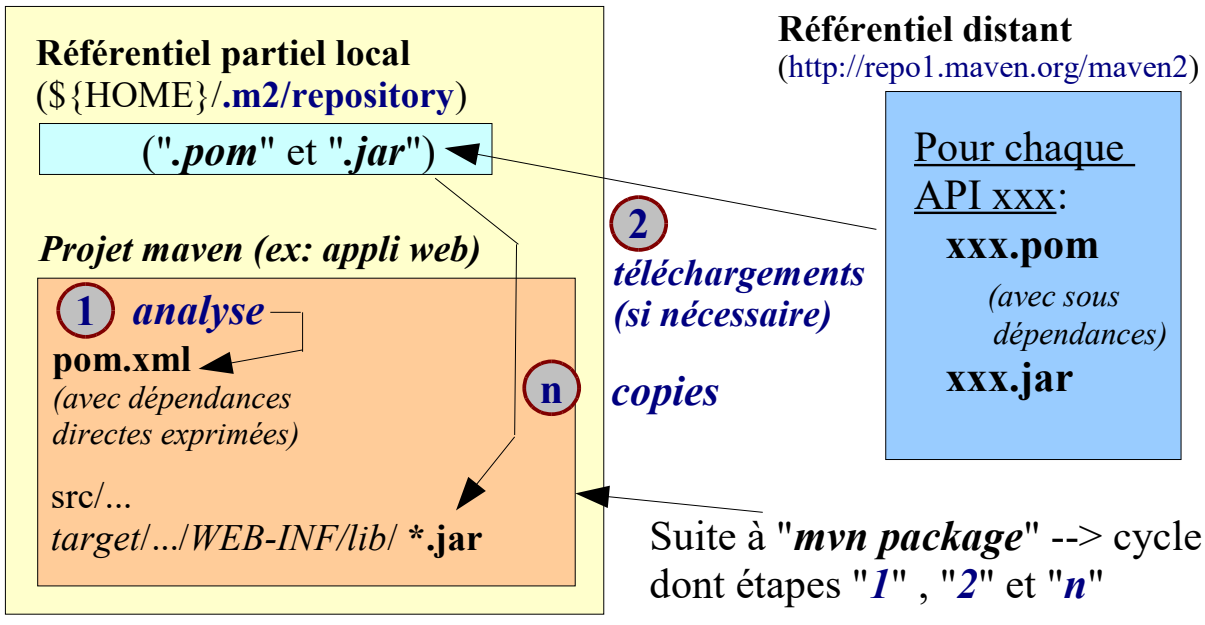
Exemple (classique) de dépendances

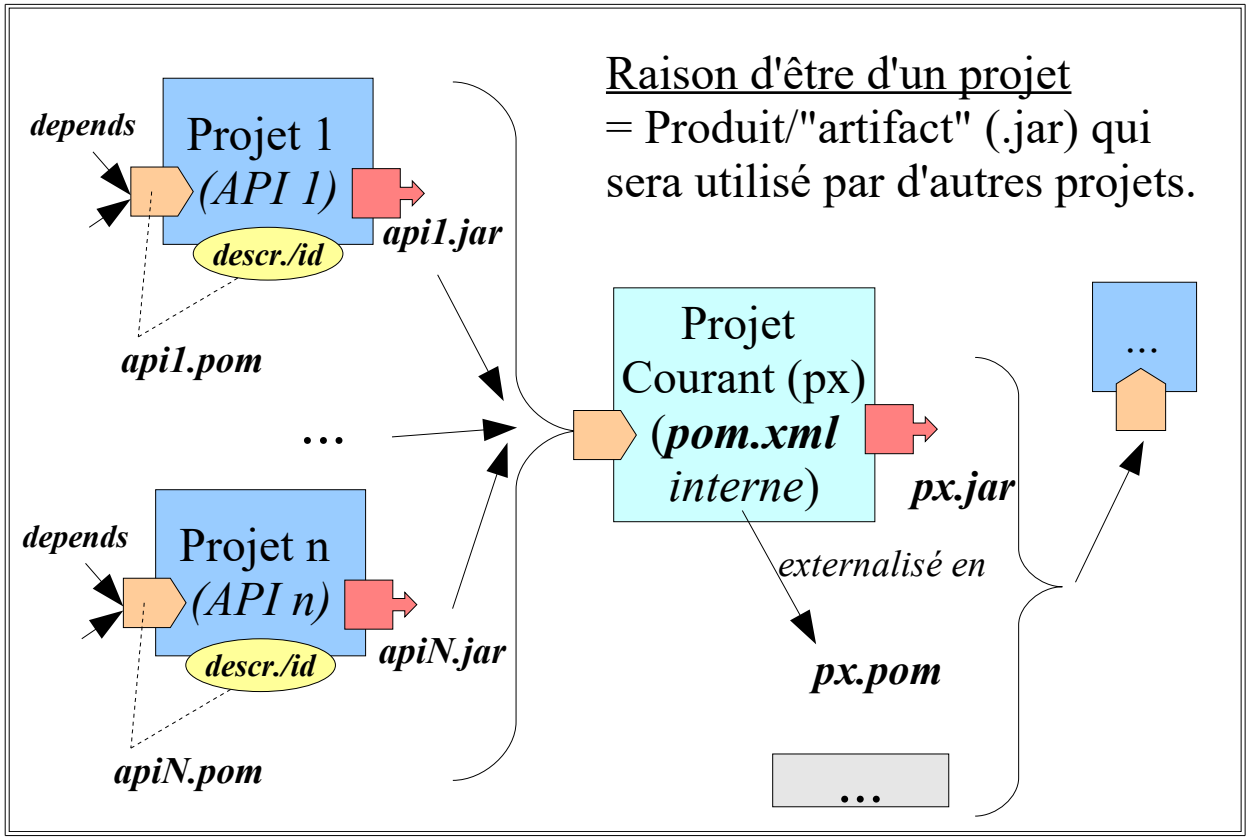
* *Un fichier "pom" à chaque niveau (API,...)*



Téléchargement automatique des librairies nécessaires

(avec prise en compte des dépendances indirectes)

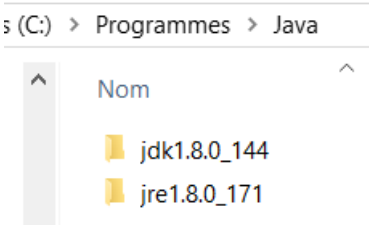



Raison d'être d'un projet
= Produit/"artifact" (.jar) qui sera utilisé par d'autres projets.
depends
Projet 1
(API 1)
descr./id
api1.jar
api1.pom
...
depends
Projet n
(API n)
descr./id
apiN.jar
apiN.pom
Projet Courant (px)
(pom.xml interne)
px.jar
externalisé en
px.pom
...
...

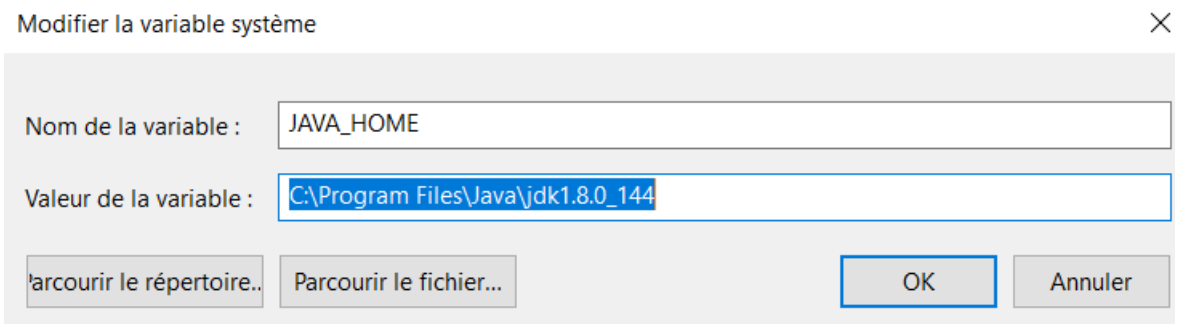
II - Maven (installation, utilisation)

1. Variables d'environnement à bien régler

Maven est une technologie qui s'appuie en interne sur java pour fonctionner. il est donc important de vérifier le bon paramétrage des variables d'environnement **PATH** et **JAVA_HOME** .

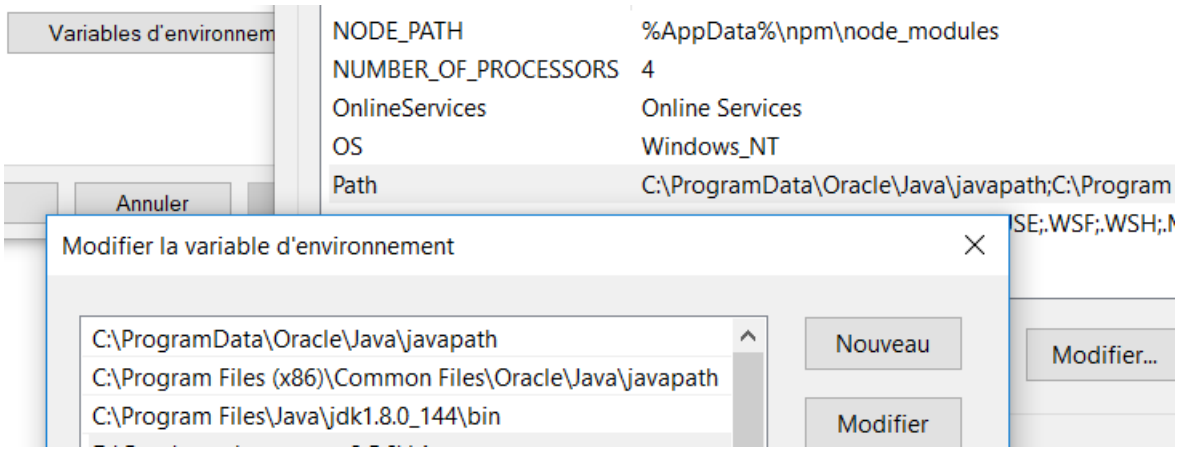


Exemple : **JAVA_HOME**=***C:\Program Files\Java\jdk1.8.0_144***
PATH=.... ;*C:\Program Files\Java\jdk1.8.0_144**bin*** ;....



D'autre part , une fois que maven.3.5....zip aura été recopié et "dezippé/décompressé" sur le poste du développeur, il faudra que le sous répertoire **bin** de ***apache-maven-3.5...*** soit **présent** dans le **PATH** pour que la commande "**mvn**" soit trouvée dans le cadre d'un déclenchement en ligne de commande (depuis une fenêtre "CMD" par exemple sous windows") .

et donc **PATH**=.... ;*C:\Program Files\Java\jdk1.8.0_144**bin*** ;***D:\Prog\apache-maven-3.5.0\bin***;... par exemple .



NB : Selon la version de windows (ex : 10 ou autre) , le paramétrage des variables d'environnement pourra être effectué depuis une partie du panneau de configuration (que l'on peut souvent rechercher via "**env**") .

2. Mise en oeuvre de Maven

Utilisation pratique de maven

Bien que l'on puisse indirectement utiliser maven via le plugin eclipse "m2e", l'utilisation directe de "maven" via des lignes de commandes est conseillée pour bien appréhender et comprendre maven.

1. télécharger (*http://maven.apache.org/download.html*) et installer le produit "maven" en "dézippant" le contenu de "***apache-maven-2-ou-3.zip***" et en ajoutant c:\....\maven...\bin dans le path du système (os).
Tester l'installation via la commande "***mvn --version***"

2. créer un nouveau projet maven "*my-app*" via la commande
mvn archetype:**create** -D**groupId**=*com.mycompany.app* -D**artifactId**=*my-app*
--> ceci permet de créer toute l'arborescence de l'application "my-app" .

3. lancer les phases "compile + ... + package (.jar)" via la commande
mvn package

4. tester éventuellement l'application "*Hello world*" via
java -cp *target/my-app-1.0-SNAPSHOT.jar* com.mycompany.app.*App*

NB : avec ancienne version de maven --> mvn archetype:create
avec version récente de maven --> mvn archetype:generate

Ligne de commande pour créer un nouveau projet "maven"

```
mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=my-app
```

Arborescence de répertoires et fichiers créée :

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- mycompany
    |               `-- app
    |                   `-- App.java
    `-- test
        `-- java
            `-- com
                `-- mycompany
                    `-- app
                        `-- AppTest.java
```

pom.xml

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

  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>Maven Quick Start Archetype</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>
</project>
```

NB: avec Maven , les dépendances sont simplement exprimées en termes de produit (ex: Junit) et de version (ex: 3.8.1) . Il n'est plus nécessaire d'indiquer explicitement la liste des ".jar" qui constituera le classpath .

Ligne de commande pour construire le projet:

```
mvn package
```

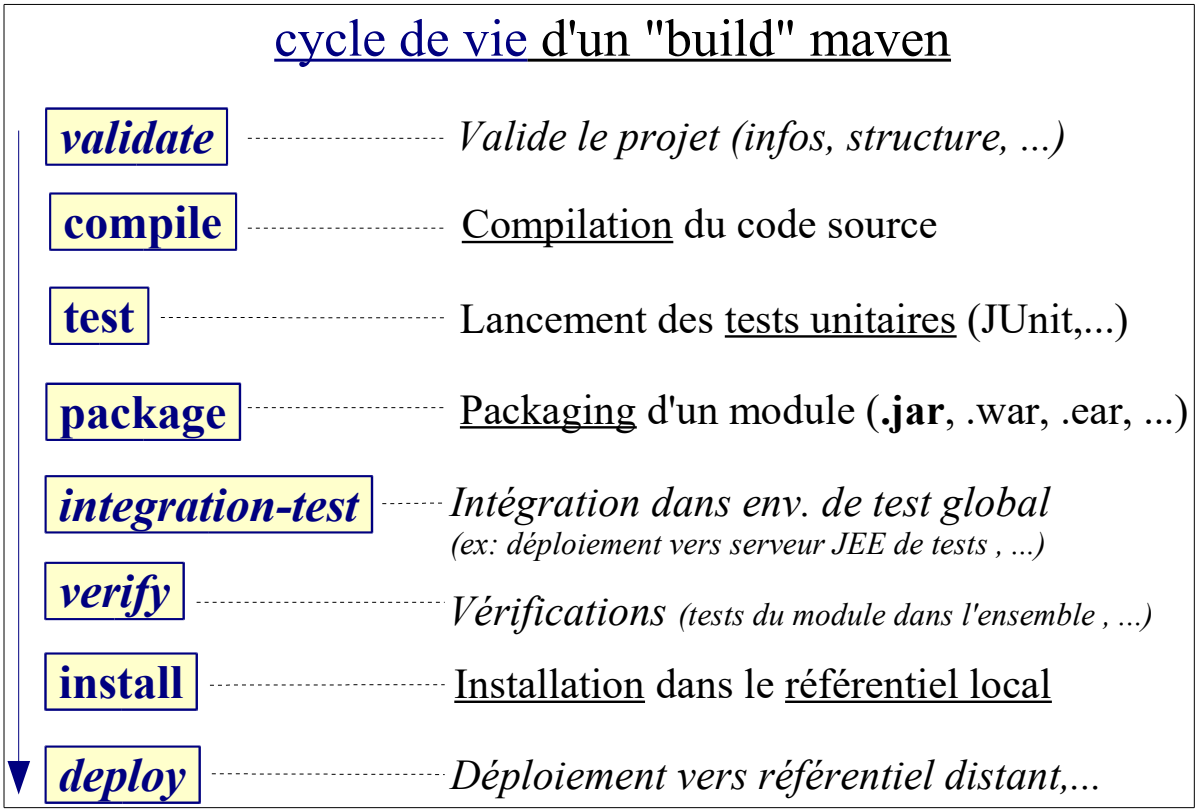
--> résultat dans "target" (.class et .jar) .

III - Maven "Goals"

Buts (goals) et phases

1.1. "buts/goals" liés au cycle de construction d'un projet maven

Principaux buts (goals):



Phases du cycle de construction

Lorsque l'on déclenche une *ligne de commande* "**mvn <goal>**" (ex: **mvn install**) ,
on ***demande explicitement à atteindre un but*** .

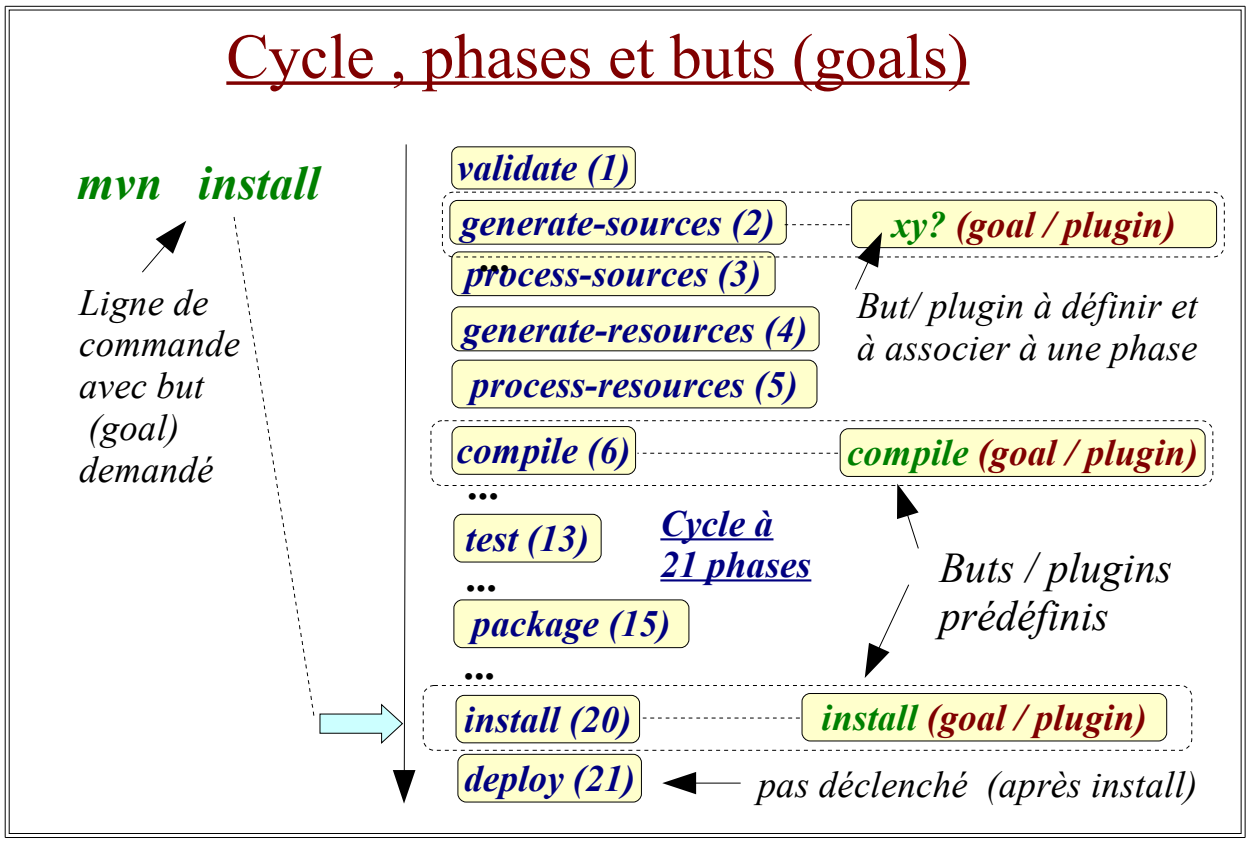
Pour atteindre ce but , maven va déclencher un processus de construction qui comporte au maximum 21 phases (dans la version actuelle).

Ces phases ont des "ordres" et "noms" bien déterminés (ex: validate(1) , generate-sources (2) , ...).

Selon le paramétrage du projet (pom.xml) , les phases de 1 à n-1 seront (ou pas) associées à des plugins (actions / buts préalables) à déclencher.

La demande d'atteinte du but associé à la phase n , déclenche dans l'ordre l'exécution de tous les plugins associés aux phases 1 , ..., n-1 puis n .

lien entre phases et buts (goals):



Phases du cycle de construction par défaut:
Plugins (et goals) prédéfinis et activés lors du cycle par défaut (pour packaging "jar").

Nom de la phase	*plugin:goal*
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar (*)
install	install:install
deploy	deploy:deploy

(*) ou war:war , ejb:ejb3 , ear:ear ... selon autre type de packaging du projet .

NB:
- Chaque but (goal) est codé dans un plugin maven (packagé comme un ".jar") .
- Un plugin maven peut contenir plusieurs buts (goals)

[ex: deploy:deploy , deploy:deploy-file , deploy:....]

Liste des 20 à 23 (*) phases du principal cycle de construction "maven" (par défaut):

(*) selon version de maven

validate	Valide si le projet est correcte et que toutes les informations nécessaires sont disponibles.
initialize	initialise la construction (set properties, create directories).
generate-sources	Génération éventuelle de code source à inclure dans la future compilation (ex: xdoclet , apt , ...)
process-sources	Traite le code source code, (ex: filtrage: remplacement de variables par des valeurs selon paramétrage de pom.xml).
generate-resources	génération de ressources (fichiers de configuration ou de données) pour future inclusion dans un package.
process-resources	Copie et traitement des ressources au sein du répertoire destination , prêt pour le packaging.
compile	compile le code source du projet.
process-classes	Traite après coup les fichiers créés par la compilation,par exemple pour enrichir le "bytecode" des classes Java.
generate-test-sources	éventuelle génération de code source spécifique au test .
process-test-sources	Traitement du code source de test, (ex: filtrage).
generate-test-resources	Éventuelle création de ressources pour les tests.
process-test-resources	copie et traite les ressources dans le répertoire de destination pour les tests.
test-compile	compile le code source des tests et place le résultat dans "target/test/...."
process-test-classes	Traite éventuellement après coup les fichiers créés par la compilation des tests,par exemple pour enrichir le "bytecode" des classes Java.
test	Lancement des tests unitaires (via Junit ou autre). Ces tests ne nécessite pas un packaging ni un déploiement du code des tests.
prepare-package	éventuelle préparation du packaging (ajustement de ... , ...)
package	Packaging du code compilé et des ressources (.jar , .war , ...).

pre-integration-test	éventuelle préparation des tests d'intégration (ex: préparer l'environnement d'exécution).
integration-test	Traite et déploie si nécessaire le package dans un environnement d'exécution (ex: serveur JEE , ...) au sein duquel les tests d'intégration peuvent s'exécuter.
post-integration-test	Éventuels post-traitements pour les test d'intégration (ex: "clean" , ...)
verify	Lance d'éventuelles vérifications du package pour vérifier sont intégrité et sa qualité.
install	Installe le package dans le référentiel local pour qu'il puisse être réutilisé depuis d'autre projets maven (du même poste de développement).
deploy	Copie en plus le package créé dans un référentiel partagé de l'entreprise (ex: référentiel géré par archiva).

Les 4 phases du cycle générant la documentation (Site Lifecycle):

pre-site	préparation
site	génération de la documentation du projet (site)
post-site	Finalisation et éventuelle préparation au déploiement
site-deploy	déploiement de " site documentation " vers le serveur web spécifié

A déclencher via ***mvn site*** ou ***mvn site-deploy*** .

1.2. Autres Buts (goals) fondamentaux (clean , ...)

mvn clean ..... pour supprimer ce qui a été (anciennement) généré dans "target" .

Cycle spécifique au "clean":

Clean Lifecycle

pre-clean	Pre....
clean	Supprime tous les fichiers de target (générés via anciens builds)
post-clean	Post....

Autres buts:
selon plugins (et documentation associée)

2. Quelques options importantes

L'option **-U** (alias ***--update-snapshots***) de maven signifie "***force update of snapshots***"
et demande à télécharger une éventuelle "nouvelle version plus récente" d'une dépendance du projet dont la version se termine par "-SNAPSHOT" .

NB : Une version ordinaire (ex : 1.3 ou 1.3-RELEASE) qui ne se termine pas par -SNAPSHOT est considérée par maven comme "définitive" et n'est pas "re-téléchargée" pour rien .

Un artifact (ex : .jar) dont la version se termine par "0.0.1-SNAPSHOT" peut être re-généré (ré-écrasé) en cours de développement par une version améliorée (code différent) ayant pourtant le même numéro de version " 0.0.1-SNAPSHOT" et l'option -U peut alors être utile .

L'option **-C** (ou bien ***--strict-checksums***) permet de demander une vérification des cohérences de "checksums" (entre .jar et .sha1 par exemple) .

Sans cela simple warning (passant souvent inaperçu) , grâce à cela , échec du build .

Dans certains cas (heureusement assez rares) , le réseau informatique est lent (ou un peu défectueux) et les téléchargements des librairies peuvent être interrompus avant leurs fins normales : il manque alors quelques octets dans un ".jar" mal téléchargé d'une partie du référentiel local (dans un des répertoires de ${HOME}/.m2/repository/ ...).

La technologie maven croit alors que le ".jar" est correct (alors qu'il ne l'est pas) . comme le ".jar" est tout de même présent , un re-téléchargement n'est pas retenté en mode "release" dans que le répertoire contenant le ".jar" défectueux n'est pas supprimé .

La commande **mvn dependency:purge-local-repository** (à lancer comme d'habitude depuis le répertoire du projet courant) est prévue pour supprimer et re-télécharger (dans le référentiel local ${HOME}/.m2/repository/) les dépendances (directes et indirectes) du projet courant paramétré par pom.xml .

IV - Fichiers "P.O.M." (structures , ...)

1. POM (Project Object Model)

1.1. GroupId & artifactId

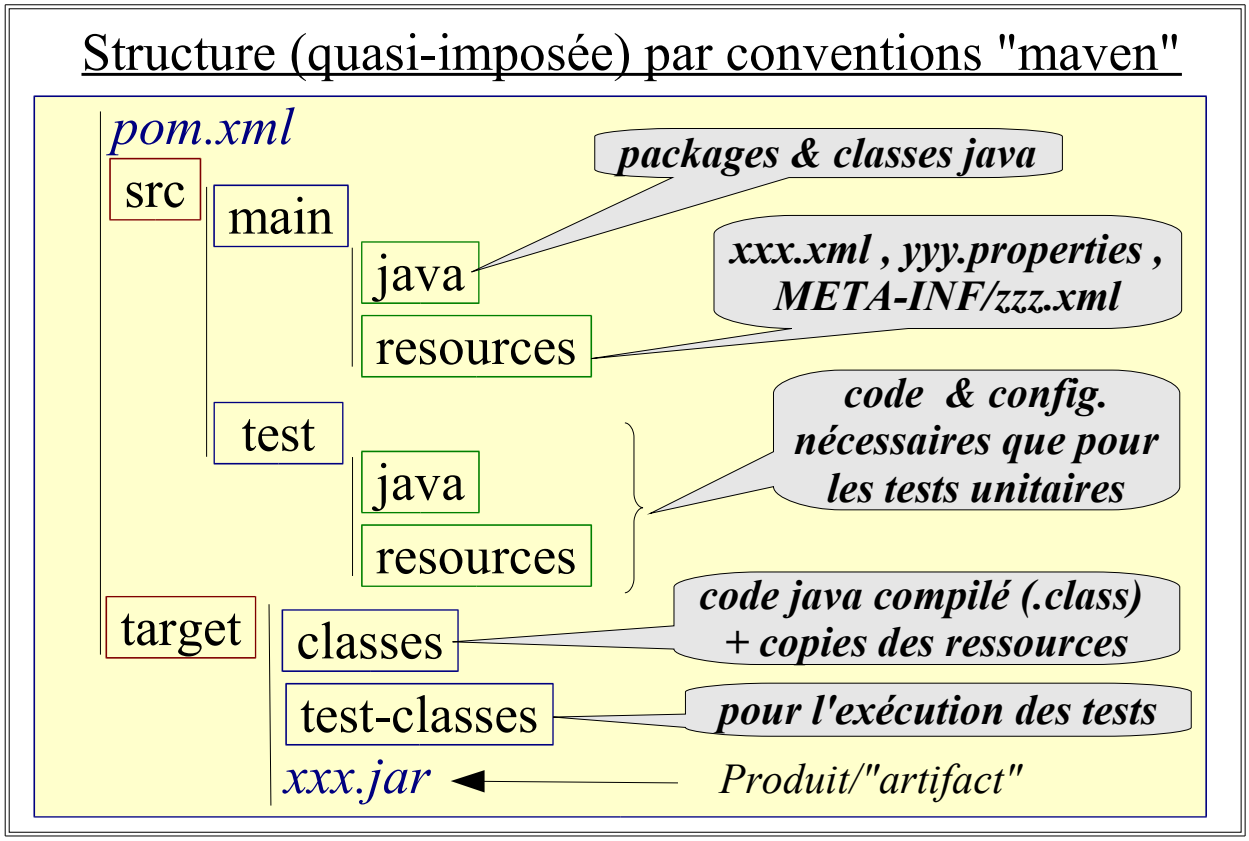
groupId & artifactId (quelques exemples)

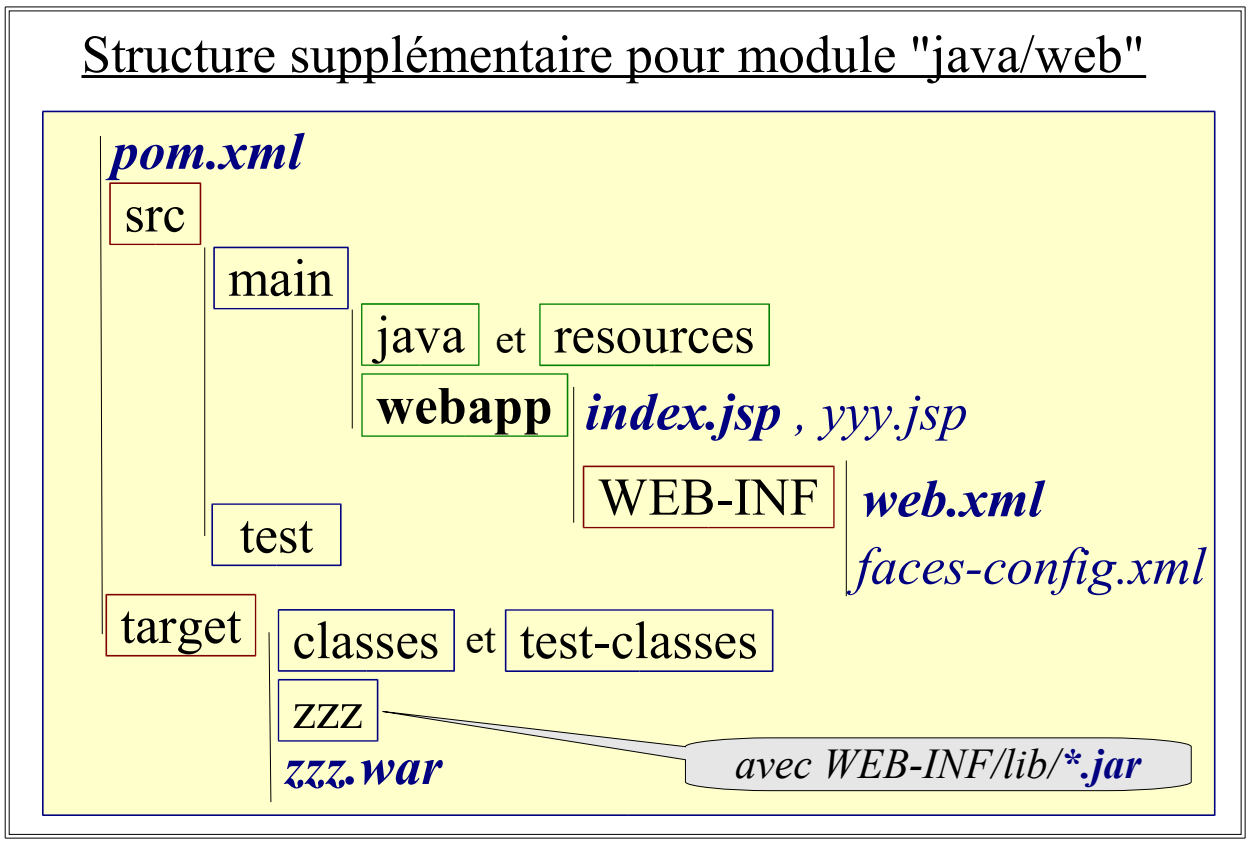
Éditeur, organisation/entreprise (+éventuelle sous branche)

Produit (application, sous module,Api)

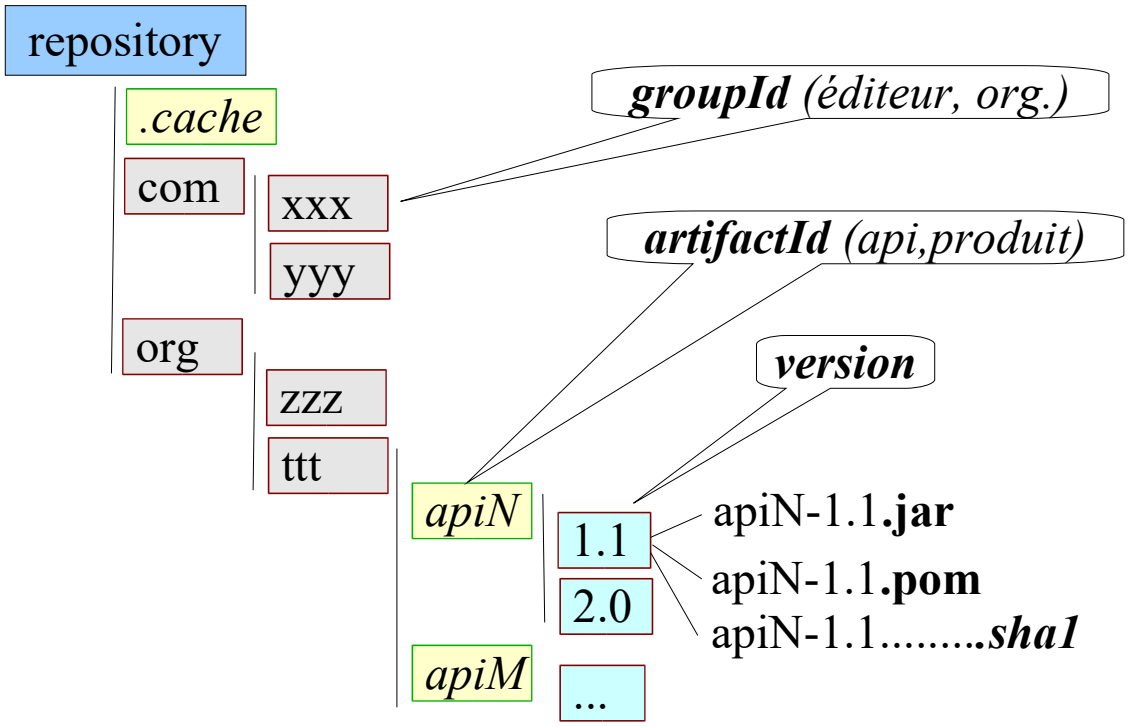
org.apache.cxf	cxf-api , cxf-rt-core , ...
org.springframework	spring-core , spring-orm , ...
org.hibernate	hibernate-core, hib...-annotations, ...
javax.persistence	persistence-api (JPA)
javax.servlet	servlet-api

1.2. Arborescences conventionnelles





Structure d'un référentiel "maven" *(local ".m2" ou distant)*



1.3. portées ("scope") des dépendances:

Principaux types de dépendances "maven" (*scope*)

. **compile** *(par défaut)*
--> ***nécessaire pour l'exécution et la compilation*** *(dépendance directe puis transitive)* [diffusé dans tous les "classpath"].

. **runtime**
--> ***nécessaire à l'exécution*** *(dépendance indirecte* ***transitive****)*

.**provided**
--> ***nécessaire à la compilation*** mais ***fourni par l'environnement d'exécution (JVM + Serveur JEE)*** *[diffusé uniquement dans les "classpath" de compilation et de test, dépendance non transitive]*

. **test**
--> uniquement nécessaire pour les ***tests*** *(ex: spring-test.jar , junit4.jar)*

Diffusé dans quel(s) "classpath" ?

Type de dépendances	compilation	Tests unitaires	exécution	Transitivité (dans futur projet utilisateur / propagation)
compile (C)	x	x	x	C(C) -->C(*), P(C) -->P T(C) -->T, R(C) -->R
provided (P)	x	x	x *(provided)*	--> pas propagé , à ré-expliciter si besoin
test (T)	x	x		--> pas propagé
runtime (R)		x	x	R(R) --> R, C(R)-->R T(R)-->T, P(R)-->P

(*) bizarrement quelquefois "compile" plutôt que "runtime" dans le cas où l'on souhaite ultérieurement étendre une classe par héritage.

2. Structure & syntaxes (pom.xml)

Fichier "POM" (éléments essentiels)

servlet-api-2.3.pom (exemple)

```
<project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>javax.servlet</groupId>
 <artifactId>servlet-api</artifactId>
 <version>2.3</version>
</project>
```

Version du modèle interne de maven

biblio-web-....pom

```
<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> ...  </parent>
 <groupId>tp</groupId>   <artifactId>biblio-web</artifactId>
 <packaging>war</packaging>  <version>0.0.1-SNAPSHOT</version>
 <name>biblio-web JEE5 Webapp</name>
 .... <dependencies> ... </dependencies>  <build>....</build>
</project>
```

Fichier "POM" (déclaration des dépendances / partie 1)

```
<project ...>
 <modelVersion>4.0.0</modelVersion> <parent> ... </parent>
 <groupId>tp</groupId>  <artifactId>biblio-web</artifactId>
 <packaging>war</packaging> <version>0.0.1-SNAPSHOT</version>
 <name>biblio-web JEE5 Webapp</name>....
<dependencies>
      <dependency>
                  <groupId>javax.servlet</groupId>
                  <artifactId>servlet-api</artifactId>
                  <version>2.5</version>      <scope>provided</scope>
       </dependency>
       <dependency>
                  <groupId>org.slf4j</groupId>
                  <artifactId>slf4j-api</artifactId>
                  <version>1.5.6</version>    <scope>compile</scope>
       </dependency> ...
</dependencies>
 <build>....</build>
</project>
```

Fichier "POM" (déclaration des dépendances / partie 2)

```
...
<dependency>
      <groupId>org.hibernate</groupId>  <artifactId>hibernate-core</artifactId>
      <version>3.5.1-Final</version> <scope>compile</scope>
      <exclusions>
            <exclusion>
                  <groupId>javax.transaction</groupId>
                  <artifactId>jta</artifactId>
            </exclusion>
            <exclusion>
                  <groupId>asm</groupId>  <artifactId>asm</artifactId>
            </exclusion>
      </exclusions>
</dependency>
<dependency>
      <groupId>javax.transaction</groupId>
      <artifactId>jta</artifactId>  <version>1.1</version>
</dependency>
...
```

exclusion(s) explicite(s) de dépendance(s) indirecte(s) transitive(s)

Contrôle direct de la version souhaitée pour éviter des conflits ou des doublons

Mise au point des dépendances (partie 3)

- La ***mise au point des dépendances*** peut éventuellement être délicate en fonction des différents points suivants:
  - * potentiel ***doublon*** (2 versions différentes d'une même librairie) à partir de plusieurs dépendances transitives indirectes.
  - * potentiel ***conflit*** de librairie à l'exécution (incompatibilité entre une librairie "A" en version "runtime" et une librairie complémentaire "B" en version "provided" imposée par le serveur JEE)
  - * autres mauvaises surprises de "murphy" .
- Eléments de solutions:
  - * ***étudier finement les compatibilités/incompatibilités*** et re-paramétrer les "***version***" et "***exclusion***"
  - * (tester , ré-essayer , re-tester) de façon itérative

Fichier "POM" (partie "build")

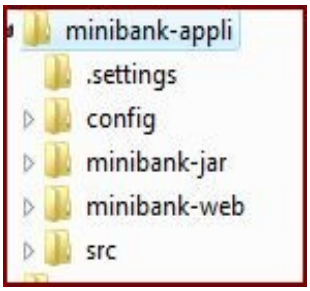
```
<project ...>
 <modelVersion>4.0.0</modelVersion>  <parent> ...  </parent>
 <groupId>tp</groupId>   <artifactId>biblio-web</artifactId>
 <packaging>war</packaging>  <version>0.0.1-SNAPSHOT</version>
 <name>biblio-web JEE5 Webapp</name>....
<dependencies>
        <dependency>...</dependency> ...
</dependencies>
 <build>
    <plugins>  <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId><ersion>2.0.2</version>
                <configuration>
                        <source>1.6</source> <target>1.6</target>
                </configuration>
        </plugin>...   </plugins>
   <finalName>biblio-web</finalName>
</build>
</project>
```

Fichier "POM" (parties "repositories" et "properties")

```
<project  ...>
  <modelVersion>4.0.0</modelVersion>  <parent> ...  </parent>
  <groupId>tp</groupId>   <artifactId>biblio-web</artifactId>
  <packaging>war</packaging>  <version>0.0.1-SNAPSHOT</version>
  <name>biblio-web JEE5 Webapp</name>....
<repositories> <!-- en plus de  http://repo1.maven.org/maven2 -->
        <repository> <!-- specific repository needed for richfaces -->
            <id>jboss.org</id>
            <url>http://repository.jboss.org/maven2/</url>
         </repository>     ...
</repositories>
<properties>
    <org.springframework.version>3.0.5.RELEASE</org.springframework.version>
    <org.apache.myfaces.version>2.0.3</org.apache.myfaces.version>
</properties>
<dependencies>  <dependency>...
      <version>${org.springframework.version}</version>
</dependency> ...</dependencies>   <build>....</build>
</project>
```

3. Configuration multi-modules (avec sous projet(s))

Mode "multi-projets" [parent/enfants , ear(war,jar)]



pom.xml (parent)

```
<project ....>...
 <artifactId>minibank-appli</artifactId>
 <packaging>pom</packaging>
 <modules>
    <module>minibank-jar</module>
    <module>minibank-web</module>
 </modules>
</project>
```

pom.xml (mod. web)

```
<project ....>...
 <parent> ...
  <artifactId>minibank-appli</artifactId>
 </parent>  <groupId>tp</groupId>
 <artifactId>minibank-web</artifactId>
 <packaging>war</packaging>
 <dependencies>    <dependency>
        <groupId>tp</groupId> ...
        <artifactId>minibank-jar</artifactId>
        <scope>runtime ou compile</scope>
   </dependency> ...<dependencies>...
</project>
```

pom.xml (sous module de services)

```
<project ....>...
 <parent>
  <artifactId>minibank-appli</artifactId>
  <groupId>tp</groupId> ...
 </parent>
 <groupId>tp</groupId>
 <artifactId>minibank-jar</artifactId>
</project>
```

Mode "multi-modules" (suite)

Arborescence globale conseillée:

my-global-app
- pom.xml
- minibank-jar (module de services)
  - ...
  - pom.xml
- mywebapp
  - pom.xml
  - ...

Un module retrouve son projet parent dans le répertoire parent « .. »
tandis qu'un projet retrouve son éventuel projet parent dans un référentiel maven (local ou distant).

Principal intérêt du mode multi-module :
* ne pas devoir construire de multiples projets séparés un par un (dans l'ordre attendu en fonction des dépendances)
* simplement lancer la construction du projet principal pour que tous les sous-modules soient automatiquement (re-)construits dans le bon ordre (selon inter-dépendances)

pom.xml (de niveau projet global)

```
<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.mycompany.app</groupId>
  <artifactId>my-global-app</artifactId>
  <packaging>pom</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>....</name>
  <modules>
    <module>services</module>  <!-- ou <module>../services</module>  -->
    <module>mywebapp</module>  <!-- ou <module>../mywebapp</module>  -->
  </modules>
</project>
```

....

pom.xml (de niveau sous projet / module)

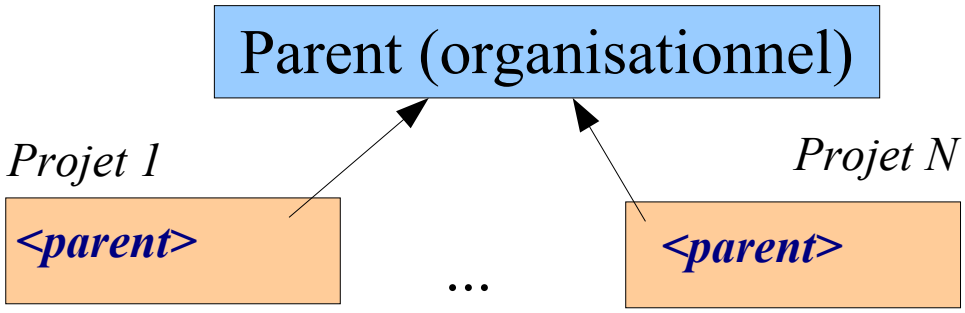
```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>mywebapp</artifactId> <!-- même nom que sous module courant -->
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging> <!-- ou jar -->
  ...
  <parent>
    <groupId>com.mycompany.app</groupId>
    <artifactId>my-global-app</artifactId>
  </parent>
  ...
  <dependencies>
    <!-- ici le module de présentation (ihm web) utilise le module frère "services"
         et la dépendance sera alors interprétée comme une dépendance directe (source)-->
    <dependency>
      <groupId>${pom.groupId}</groupId>
      <artifactId>services</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
    ...
  </dependencies>

</project>
```

4. Héritage entre projets "maven" (<parent>)

Héritage "organisationnel" au niveau de "maven"

Dans l'absence d'une organisation multi-modules, la balise **<parent>** d'un fichier "**pom.xml**" permet de définir un lien d'héritage entre le projet courant et le projet parent :
Une certaine partie de la configuration du projet parent est ainsi héritée (sans devoir être répétée).
--> intérêt du projet parent: **factoriser** une **configuration commune** entre différents projets "fils".

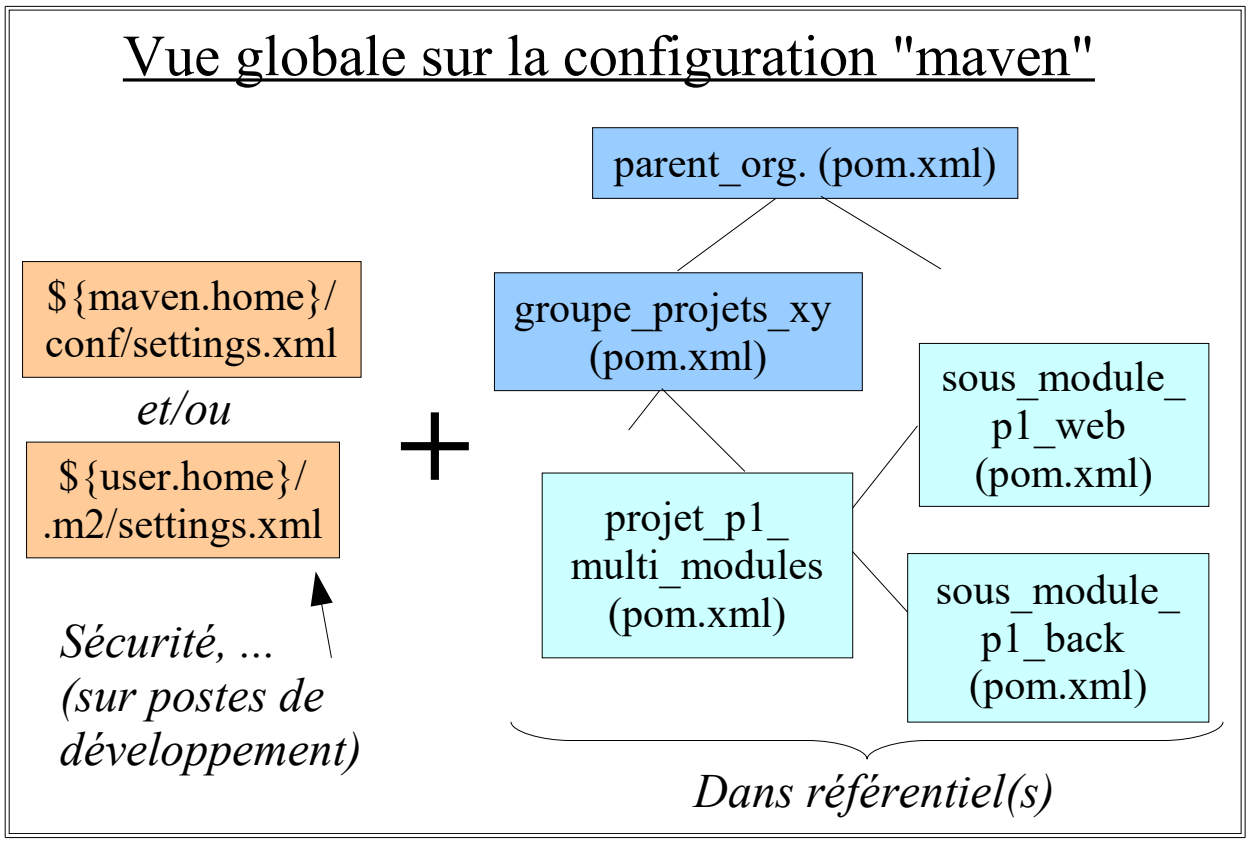


Eléments couramment hérités (pom.xml)

Un projet parent "organisationnel" *(sans obligation d'être placé dans un répertoire parent)* sert généralement à factoriser les points suivants:

- *des informations générales sur l'entreprise* (organisation, e-mails).
- *liste (avec URL) des référentiels "maven" et "svn/git" de l'entreprise et/ou des référentiels externes* (***<repositories>*** , ***<distributionManagement>*** , ...) .
- *section <**dependencyManagement**> servant à préciser des versions et des exclusions au sein des futures dépendances qui seront exprimées au cas par cas dans les projets fils*.
- ....

Conseil: plusieurs niveaux de parents (parent intermédiaire).

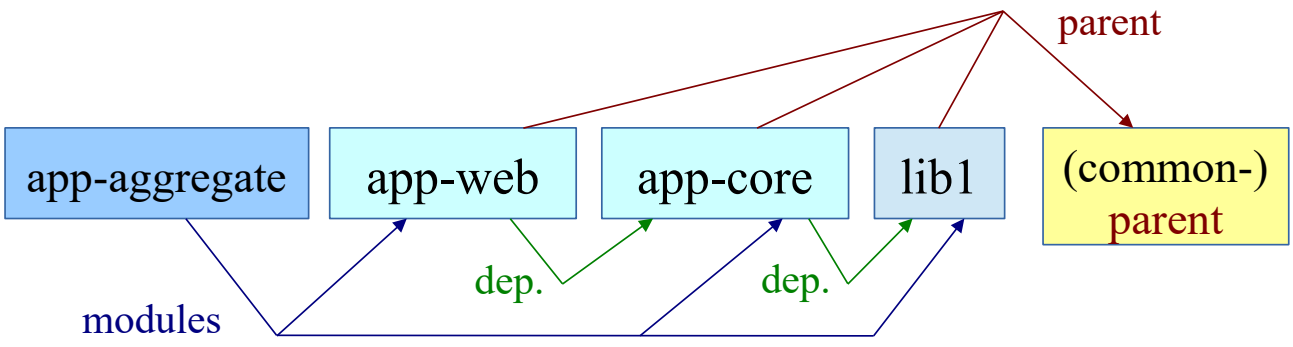


Notion de projet "aggregate" (depuis maven 3)

Un projet "**aggregate**" de type "**pom**" ***sert simplement à tout reconstruire d'un coup*** via "**mvn install**" .

Les **mécanismes "reactor" de maven** vont **tenir compte des inter-dépendances entre les projets référencés comme des modules** et **tout construire DANS LE BON ORDRE**.

Cependant, contrairement à un projet multi-modules classique **les projets référencés ne sont pas des sous-répertoires mais des répertoires de même niveau** et **ces différents projets ont un autre parent que le projet courant "app-aggregate"**



app-aggregate/pom.xml

```
<project ...>...  <packaging>pom</packaging>
 <artifactId>app-aggregate</artifactId>  <version>0.0.1-SNAPSHOT</version>
 <modules>
      <module>../common-parent</module>
       <module>../lib1</module>
       <module>../app-core</module>
       <module>../app-web</module>
 </modules> </project>
```

app-core/pom.xml

```
… <parent>...
<artifactId>common-parent</artifactId>
<version>0.0.2-SNAPSHOT</version>
<relativePath>../common-parent
</relativePath>
</parent>
<groupId>tp</groupId>
<version>0.0.1-SNAPSHOT</version>
<artifactId>app-core</artifactId>

 ... <dependency>
      <groupId>tp</groupId>
      <artifactId>lib1</artifactId>
      <version>0.0.1-SNAPSHOT</version>
  </dependency> ...
```

app-web/pom.xml

```
<parent> ...
 <artifactId>common-parent</artifactId>
 <version>0.0.2-SNAPSHOT</version>
<relativePath>../common-parent
</relativePath>
 </parent>
 <groupId>tp</groupId>
 <version>0.0.1-SNAPSHOT</version>
 <artifactId>app-web</artifactId>
 <packaging>war</packaging>
...    <dependency>
        <groupId>tp</groupId>
        <artifactId>app-core</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </dependency> ...
```

5. Archetypes

Archetype "maven" *(prédéfini ou à définir)*

- Un "***archetype***" est un ***modèle (template) de configuration de projet*** qui est accessible depuis un référentiel "maven" et qui permet de rapidement définir un nouveau fichier "pom.xml" sans devoir tout préciser en partant de "zéro".
  --> ce "*point de départ*" est évidemment à personnaliser au cas par cas selon les spécificités de chaque projet.

- Quelques exemples ("archetypes" / "projet type"):
  - "j2ee-web"
  - "jee5-..."
  - "java5"
  - "spring3+jpa2+jsf2+cxf_pour_tc6" *à mettre au point*
  - ...

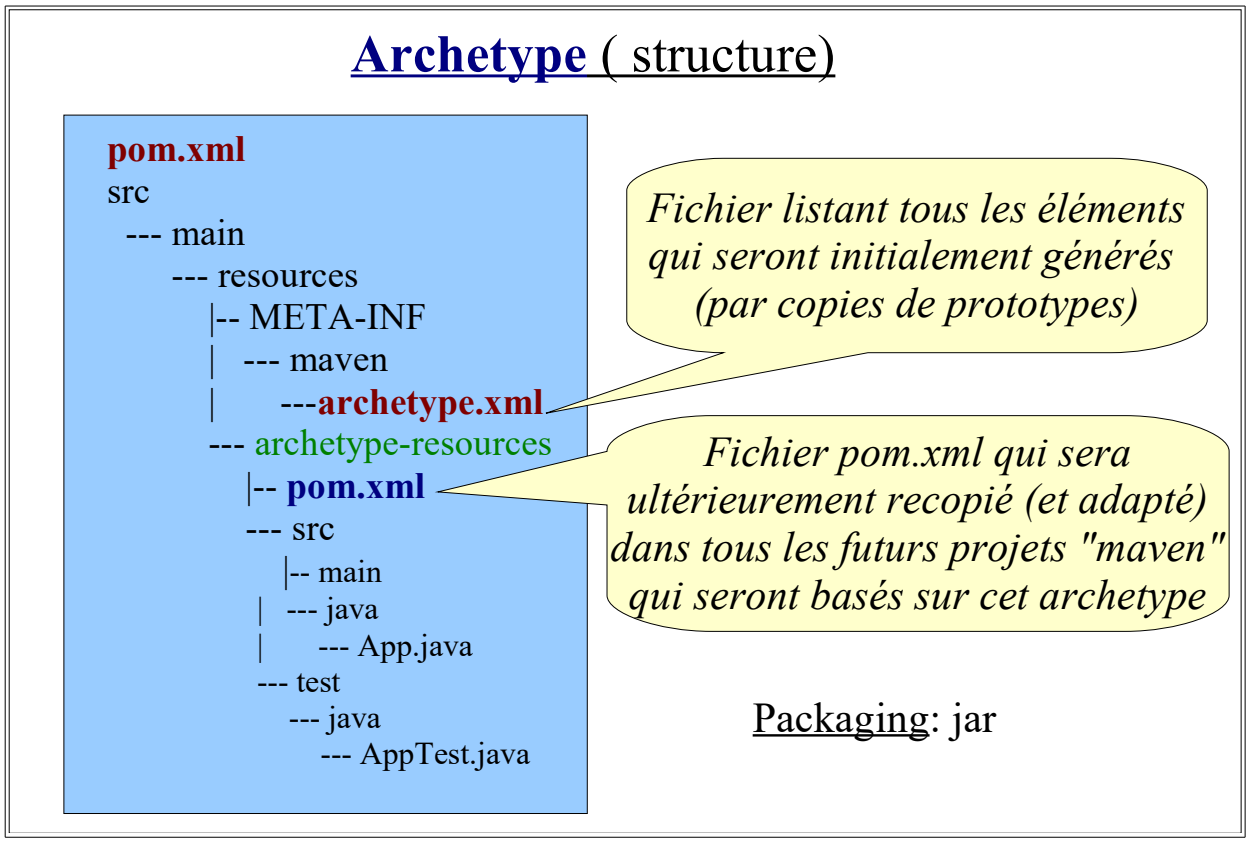
6. Utilisation d'un (nouvel) archetype maven

```
mvn archetype:generate
      -DarchetypeGroupId=com.mycompany.app1 \
      -DarchetypeArtifactId=my-java-archetype1 \
      -DarchetypeVersion=1.0-SNAPSHOT \
      -DgroupId=com.mycompany.app.via.archetype1 \
      -DartifactId=my-java-via-archetype1 \
```

NB: version attendue="RELEASE" si archetypeVersion n'est pas précisé .

+ suite habituelle (édition de code , mvn package , ...) .

7. Création d'un nouvel archetype maven



7.1. Construction d'un archetype maven depuis une application exemple/modèle .

Pour ***construire un nouvel archetype maven*** , le mode opératoire conseillé est le suivant :

1) Développer (et tester) une application exemple/modèle "*appli-exemple*" très simple (*de type point de départ avec structure conseillée et un petit exemple*) .
   Bien veiller à ce que les débuts des noms de packages coïncident avec le groupId .

2) Effectuer un "***mvn clean***" pour supprimer les parties "target" et si le développement se fait sous eclipse alors générer une copie du projet en dehors d'eclipse et en supprimant tous les fichiers "eclipse" inutiles à maven (ex : .eclipse , .project , .settings , …)

3) lancer la commande "**mvn archetype:create-from-project**" (depuis un projet mono module ou bien depuis le projet principal parent des autres modules) .
   ==> ceci permet de construire tout les fichiers sources d'un nouvel archetype.
   Résultat *(pom.xml + src)* dans **target/generated-sources/archetype** .

4) recopier le code généré dans un projet "*appli-exemple*-**archetype**" puis lancer la commande
   "mvn ***install***" ou "mvn ***deploy***" pour que l'archetype soit enregistré et utilisable .

V - Test unitaire via maven

1. Tests unitaires avec maven

1.1. Rappels sur la structure d'un projet (partie "test")

src/test/java

Test Junit 3:

```
package com.mycompany.app1;

import junit.framework.Test;
import junit.framework.TestCase;

/**
 * Unit test for simple Calculateur.
 */
public class CalculateurTest
    extends TestCase
{
    private Calculateur c;

    protected void setUp(){
      c = new Calculateur();
    }

    public void testAdd()
    {
        assertEquals( c.add(5,6) , 11 , 0.000001 );
    }

    public void testMult()
    {
        assertEquals( c.mult(5,6) , 30 , 0.000001 );
    }
}
```

Test Junit 4:

```
package package com.mycompany.app1;

import org.junit.Assert;

import org.junit.Test;
import org.junit.Before;

/**
 * Unit test for simple Calculateur. (JUnit 4 with annotations)
 */
public class CalculateurTest
{
    private Calculateur c;

   /* @BeforeClass : pour initialiser une seule fois des choses statiques
      @Before ou bien "default constructor": (re)déclenchement avant chaque test
      pour initialiser des choses "non static"  [ une instance de la classe par @Test !!! ] */
    @Before
    public void mySetUp(){
      c = new Calculateur();
    }

    @Test
    public void myTestAdd(){
       Assert.assertEquals( c.add(5,6) , 11 , 0.000001 );
    }

    @Test
    public void myTestMult(){
       Assert.assertEquals( c.mult(5,6) , 30 , 0.000001 );
    }
}
```

1.2. Lancement des tests unitaires

mvn test **-> lance tous tests dont les noms de classes commencent ou se terminent par "Test"**

mvn test -Dtest=XxxTest -> lance que le test "XxxTest"

mvn test -Dtest=*xxTest -> lance tous les tests finissants par "xxTest"

NB: par défaut , les tests unitaires sont systématiquement déclenchés lors d'un build ordinaire.
L'option "**-DskipTests=true** " de mvn permet d'annuler/sauter l'exécution des tests.

VI - Liaison avec eclipse (m2e)

1. Lien entre maven et eclipse (m2e)

Etant donné que l'IDE eclipse gère lui aussi les projets "Java/JEE" avec sa propre structure de répertoires (différente de Maven) , **il faut installer au sein d'eclipse un plugin "Maven" spécifique de façon à ce qu'eclipse puisse déléguer à Maven certaines tâches** (compilations , gestion des dépendances , ...) .

Attention: Ce plugin s'appelle m2e ("maven2eclipse") , il n'est vraiment au point que dans ses versions les plus récentes (pour eclipse 3.5 et 3.6 , 3.7 , >=4.2) .

Plugin eclipse "m2e" pour maven (partie 1)

Depuis eclipse , on peut installer le plugin "m2e" (maven to eclipse) via l'update site suivant: http://m2eclipse.sonatype.org/sites/m2e .

On peut ensuite créer de nouveaux projets eclipse de type "maven" via le menu habituel. Le choix d'un archetype peut être effectué dès la création du projet (mais n'est pas obligatoire).

Lorsque l'on restructure en profondeur le fichier pom.xml , il est conseillé de déclencher le menu contextuel "***Maven/Update Project Configuration***" d'eclipse pour que la configuration du projet eclipse s'adapte à la configuration "maven".

Le menu contextuel "***Run as ...***" d'éclipse permet de déclencher les "build" ordinaires de maven (***clean*** , ***test***, ***package*** , ***install***, ...).

NB : URL exacte à ajuster selon version d'eclipse

1.1. Utilisation du plugin eclipse "m2e"

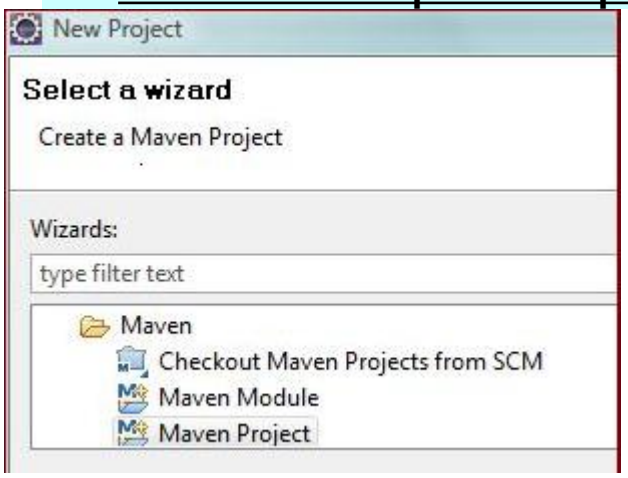
Installer si nécessaire dans eclipse 3.x ou 4.2 le plugin eclipse "m2e" via le "update site" suivant:

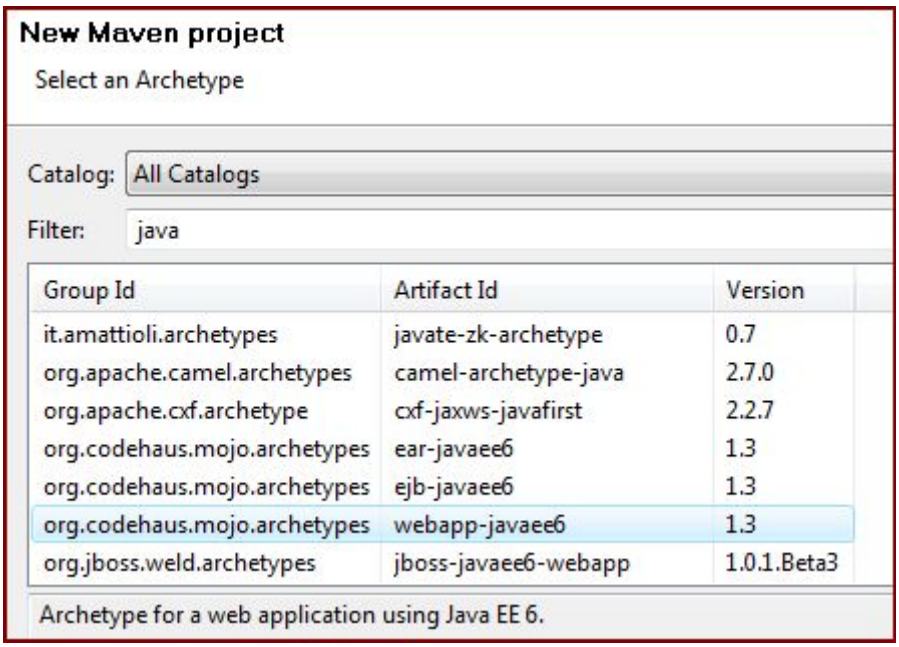
- http://...../**m2e**/**...**
- http://...../**m2e-wtp**/**...** ***[NB : m2e-wtp permet le "Run as / run on server" classique depuis un projet web sans trop d'erreur]***

\+ nouveau projet "maven"
\+ éditer manuellement le fichier "pom.xml"
\+ éditer le code au bon endroit (dans src/main/java, ...)

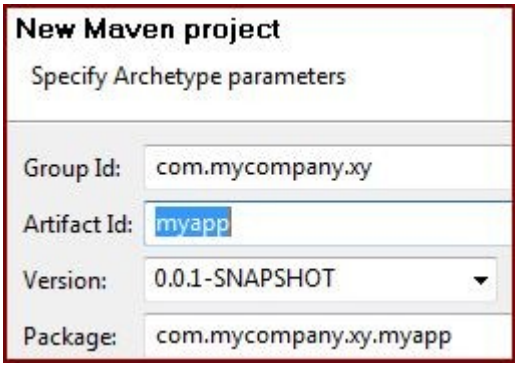
- **run as / maven .... sur le projet** (ou run as / ... sur des sous parties (test junit))

1.2. Assistants "eclipse/m2e" pour créer un nouveau projet maven

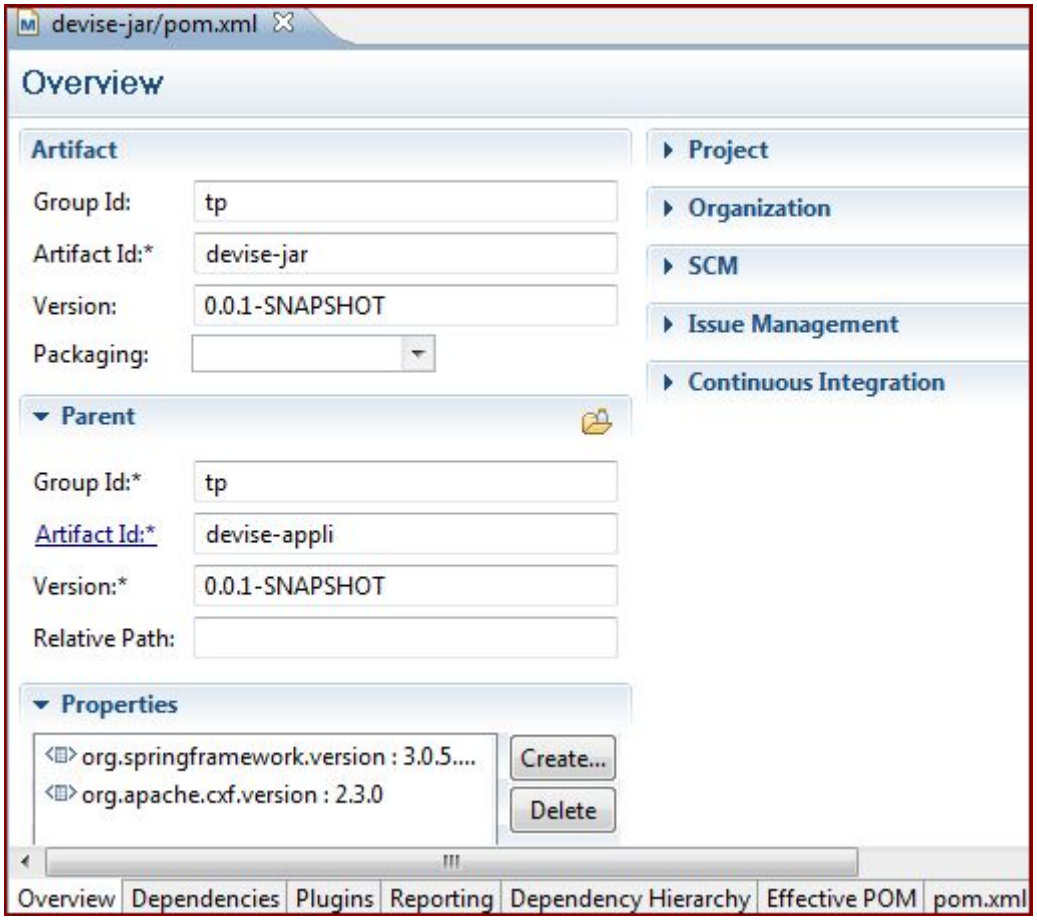


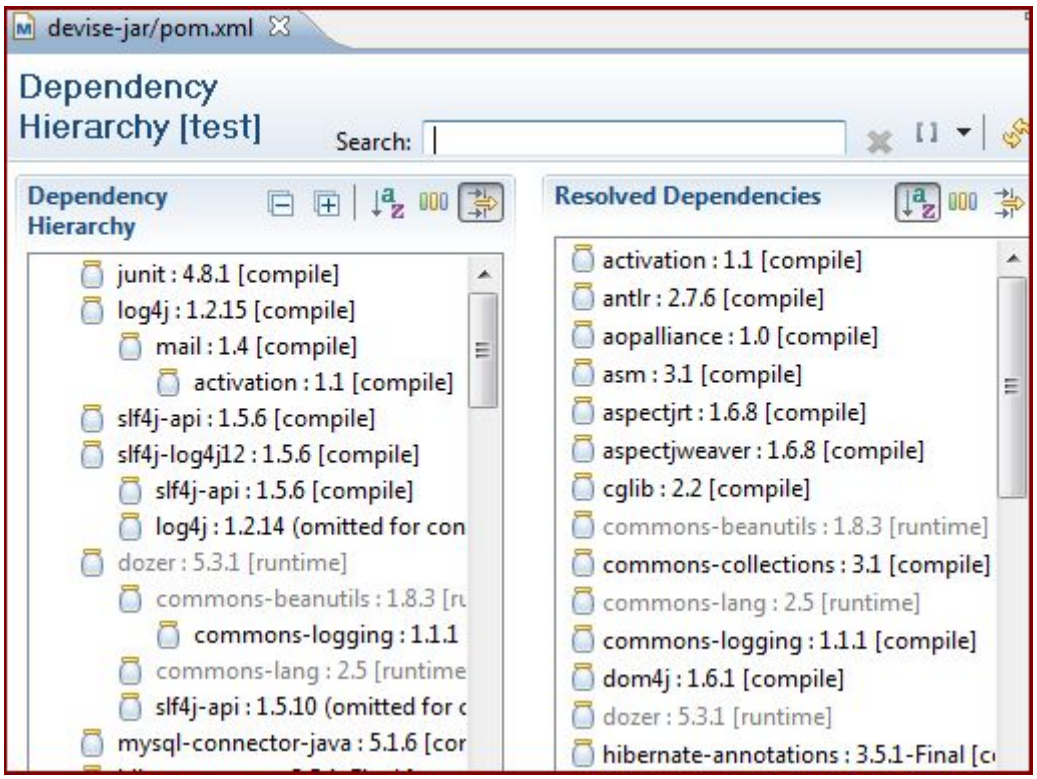


NB : Un archetype coorespond à un modèle de nouveau projet. Cela doit se préparer. **Tant qu'aucun archetype existant ne soit suffisamment intéressant, il est préférable de sauter la sélection d'un archetype en cochant la case "create simple project (skip archetype selection)"**.



1.3. Assistants "eclipse/m2e" pour paramétrer/visualiser pom.xml





NB :

- l'onglet "**pom.xml**" correspond au contenu exact du fichier "pom.xml" du projet courant.
- l'onglet "**Effective POM**" correspond à **la somme du contenu de pom.xml et de toute la configuration héritée** (projet parent + config par défaut) .
- l'onglet "**Dependency Hierarchy**" permet de bien visualiser les dépendances indirectes ce qui permet quelquefois de choisir la meilleur version en cas de conflit de versions et/ou d'alléger la configuration en n'explicitant que les dépendances essentielles .

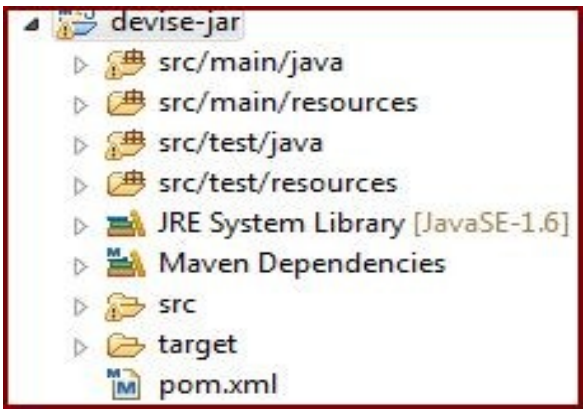
1.4. Structure des projets "eclipse_maven"

Plugin eclipse "m2e" pour maven (partie 2)

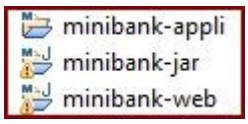
Via le plugin "m2e" , eclipse peut intégrer "maven" à travers des **projets "maven_dans_eclipse"** qui :

* ont une structure "maven" classique (***src/main/java, ...***)

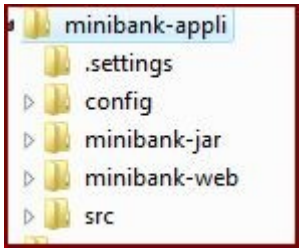
* n'ont pas la structure classique d'un projet java (~~src~~,~~bin~~)
ni la structure d'un "dynamic web projet" (~~webContent~~,...)



Cas d'une structure multi-modules:



(structure vue à plat dans eclipse)



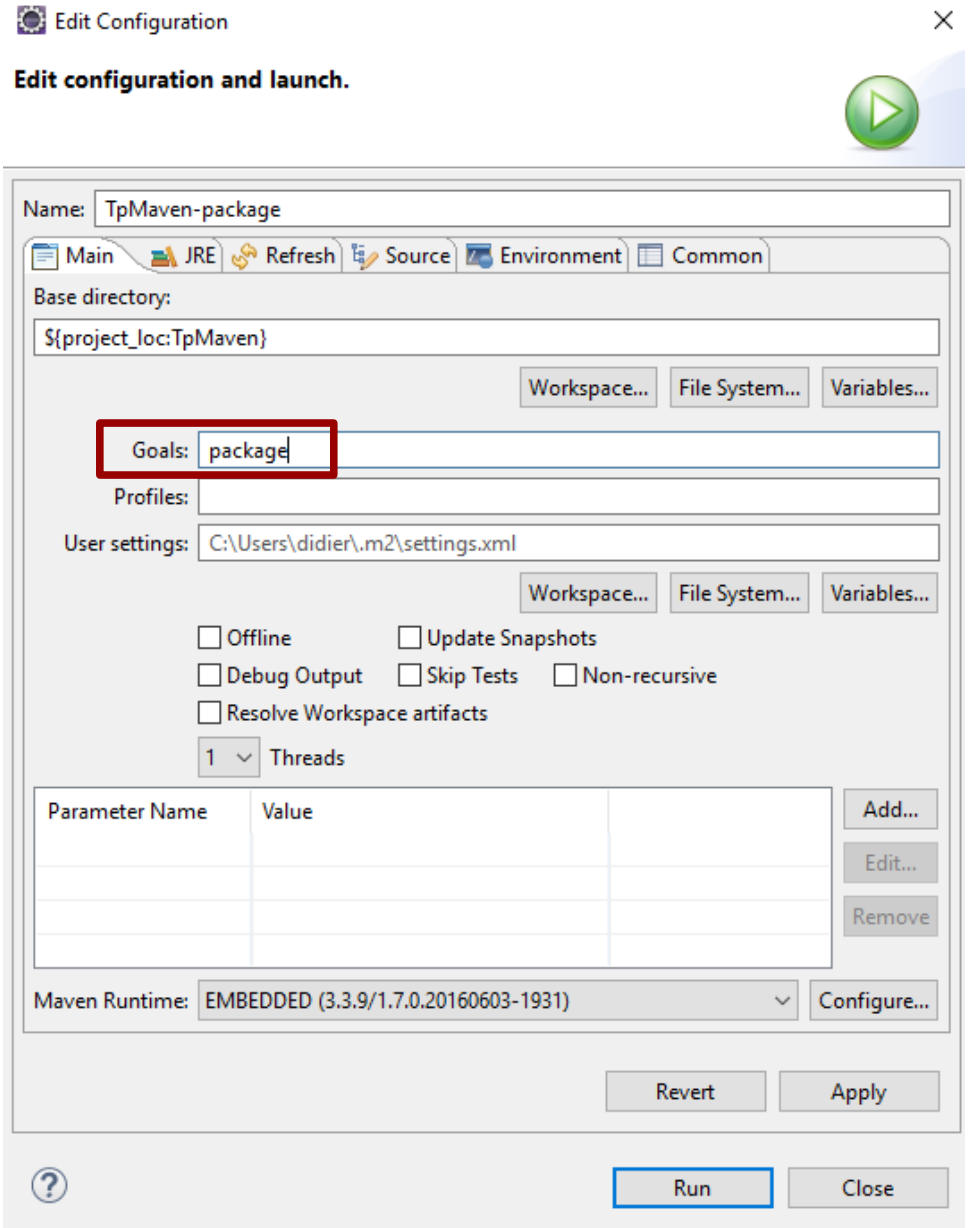
(structure arborescente sur le file system hôte).

1.5. Lancement d'un build maven depuis eclipse

Au sein des menus **"Run as / maven ...."** certains buts (classiques) sont directement accessibles :

- run as / **maven clean**
- run as / **maven compile**
- run as / **maven test**
- run as / **maven install** (package + copy artifact in .m2/repository)

Le menu "run as / **maven build ...**" permet de **faire apparaître une boite de dialogue où l'on peut saisir plein de détails dont le nom du but/goal à atteindre (ex : *package*)**



Il est souvent intéressant de donner un nom logique parlant à cette configuration (exemple : "*maven-package*") de façon à pouvoir relancer rapidement celle ci via le menu "run as / maven build" et une sélection de configuration antérieure .
NB : il est possible de retoucher certains paramétrages via le menu "project / run configurations ..."

1.6. Diverses astuces "eclipse/maven" :

- Un "Maven 3" est intégré au plugin m2e pour eclipse et une installation préalable de maven n'est pas indispensable (bien que possible et paramétrable dans le workspace eclipse).

- **Le plugin eclipse "m2e " nécessite absolument le jdk complet (plutôt que le JRE) pour fonctionner**. Il faut donc pointer vers le répertoire "**jdk** …" plutôt que "jre …" au niveau du menu "**Windows préférences / Java / Installed JREs …**"

- En cas de désynchronisation "maven-eclipse" , on peut tenter la séquence suivante :
  run as / maven clean
  run as / maven install
  project / clean (eclipse)
  server / tomcat / clean (si projet web)
  run as / run on server (si projet web)
  refresh navigateur (si projet web)

- En cas de bug inexpliqué et non identifié , il peut quelquefois être utile de redémarrer entièrement eclipse (comportant quelquefois quelques bugs ou bien rendu instable suite à des manipulations erronées) .

- Quelques fois , certains ".jar" sont mal téléchargés (mal recopiés , fichiers corrompus) et il faut manuellement **supprimer certains sous répertoires de $HOME/.m2/repository** en fonction des groupId/artifactId du message d'erreur maven .

VII - Configurations (référentiels , profils, ...)

1. Mise en place d'un référentiel "Maven"

Configuration des référentiels "maven" (partie 1)

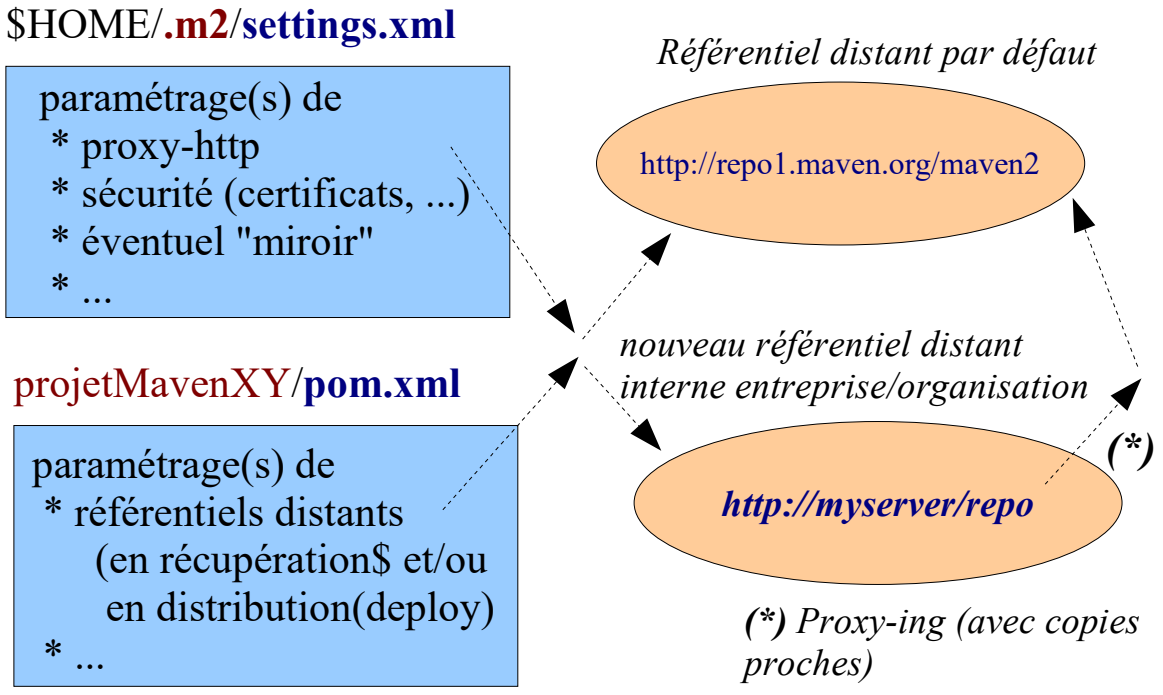
Un nouveau référentiel maven (interne à une entreprise/organisation) est simplement structuré comme le référentiel local (.m2/repository).
--> même contenu (à recopier ou alimenter)
même structure arborescente (selon groupId , artifactId et version)

Simples différences:

* Accès distant (en lecture) via **http** (ou ***https***)
* Accès distant en distribution (déploiement) de nouveaux "artifacts" via "***scp***" , "***ftp***" , "***http***" , "***webdav***" ou autre .

Les accès sécurisés (scp , https, ...) nécessitent une configuration au sein du fichier $HOME/.m2/***settings.xml***

Configuration des référentiels "maven" (partie 2)



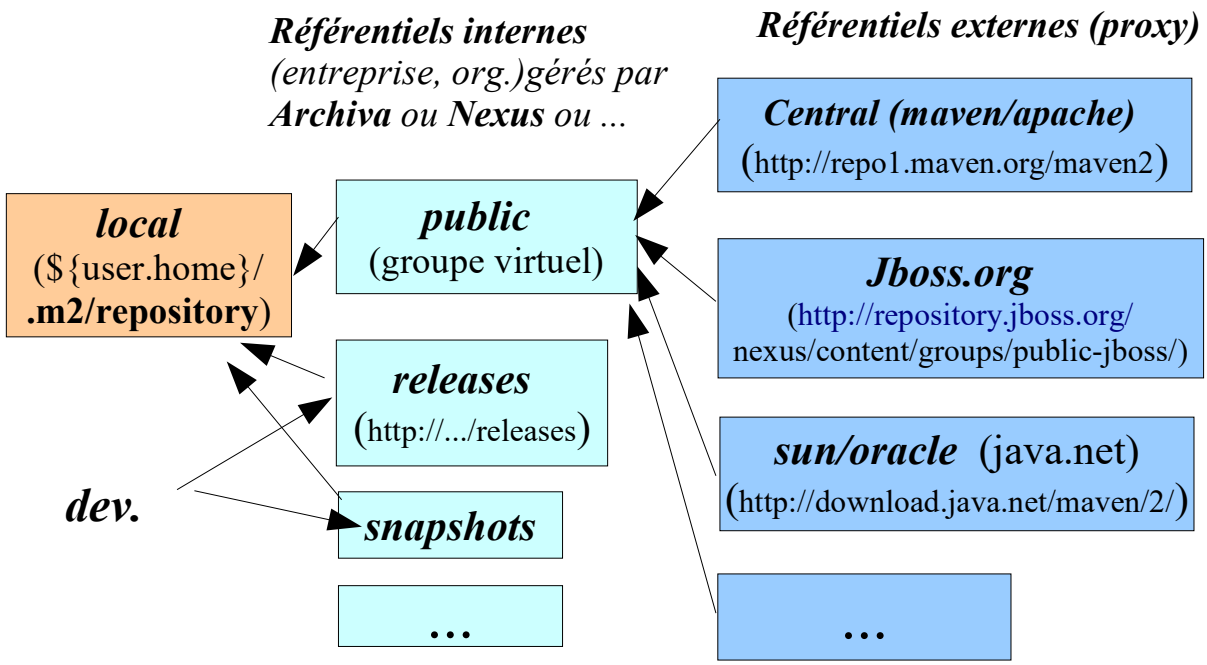
Configuration des référentiels "maven" (partie 3)

Un référentiel maven est très souvent géré par un logiciel spécialisé de type "***repository manager***".
Les produits les plus connus sont *Proximity* , *Nexus* et *Archiva* .

Archiva gère par exemple les fonctionnalités suivantes:

* ***proxy-ing vers d'autres référentiels externes*** *(avec constitution de copies des packages).*
* ***indexation*** des éléments du référentiel pour navigations et recherches rapides
* ***accès sécurisé*** vers le référentiel en écriture (via http ou autre , ...) - *comptes utilisateurs configurables*.

Vue globale sur les référentiels



1.1. Configuration du référentiel local (.m2/settings.xml)

Par défaut, lorsque rien n'est configuré , maven utilise le référentiel distant "http://repo1.maven.org/maven2" et le référentiel local $HOME/**.m2/repository** où "**.m2**" est un répertoire caché placé dans le répertoire de l'utilisateur .
...
Le fichier "$HOME/**.m2/settings.xml** peut éventuellement être modifié pour configurer le référentiel local :

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- empty maven settings.xml to copy in C:\Users\UserName\.m2\
    if C:\Users\UserName\.m2\settings.xml does not exist -->

<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0  http://maven.apache.org/xsd/settings-1.0.0.xsd">

   <!-- <localRepository>/path/to/local/repo/</localRepository>  -->


</settings>
```

1.2. Eventuelle configuration d'un proxy http (pour maven)

$HOME/.m2/settings.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<settings>
  <proxies>
    <proxy>
      <active>true</active>
      <protocol>http</protocol>
     <!-- <username>username</username>
          <password>pwd</password> -->
      <port>8080</port>
      <host>my.proxy.url</host>
      <!-- <nonProxyHosts>www.google.com|*.somewhere.com</nonProxyHosts> -->
      <!-- <id>idOfProxy</id> -->
    </proxy>
  </proxies>
</settings>
```

1.3. Référentiel spécifique (interne à l'entreprise)

Il est possible de configurer de nouveaux référentiels au sein d'une organisation (entreprise).
...
Une fois en place , ce nouveau référentiel pourra être référencé/utilisé de la façon suivante:

pom.xml

```
<project>
...
<repositories>
   <repository>
     <id>my-internal-site</id>
     <url>http://myserver/repo</url>
   </repository>
 </repositories>
</project>
```

ou bien être configuré comme un mirroir dans .m2/settings.xml:

```
<settings>
  ...
  <mirrors>
    <mirror>
      <id>internal-repository</id>
      <name>Maven Repository Manager running on repo.mycompany.com</name>
      <url>http://repo.mycompany.com/proxy</url>
      <mirrorOf>*</mirrorOf>
    </mirror>
  </mirrors>
  ...
</settings>
```

1.4. Préciser le référentiel de distribution (produits)

Pour déployer les ".jar" produits vers un référentiel Maven spécifique , on peut indiquer l'url du référentiel destination dans le "pom.xml" :

pom.xml

```
<project>
<distributionManagement>
  <repository>
    <id>mycompany-repository</id>
    <name>MyCompany Repository</name>
    <url>scp://repository.mycompany.com/repository/maven2</url>
    <!-- ou bien <url>file://localhost/z:/internalrepository/maven2</url>
     si un lecteur distant windows ou un lien nsf unix est configuré pour atteindre un répertoire
     partagé distant -->
  </repository>
 </distributionManagement>
</project>
```

Selon le mode de connexion (ici "scp") , des informations d'authentification vis à vis du référentiel sont quelquefois nécessaires et elles doivent être placées dans le fichier "settings.xml":

settings.xml

```
<settings 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/settings-1.0.0.xsd">
  ...
  <servers>
    <server>
      <id>mycompany-repository</id>
      <username>jvanzyl</username>
      <!-- Default value is ~/.ssh/id_dsa -->
      <privateKey>/path/to/identity</privateKey> (default is ~/.ssh/id_dsa)
      <passphrase>my_key_passphrase</passphrase>
    </server>
  </servers>
  ...
</settings>
```

2. Repository Manager (Nexus ou ...)

Pour prendre en charge un référentiel interne (spécifique à une entreprise) , il faut au **minimum** un **serveur Http** (ex: Apache ou Tomcat) .

L' idéal consiste à installer un gestionnaire de référentiel (*Repository Manager*) qui prendra en charge quelques unes des fonctionnalités suivantes:

- **indexation**
- **redirection vers référentiels externes** (avec constitution automatique de copies locales)
- **sécurisation des ajouts au référentiel** (via username/password , ...)
- ....

Les "***Repository Manager***" disponibles pour Maven sont (pour les plus connus):

- **Proximity** (assez ancien)
- **Nexus** de "*SonaType*" (avec bon système d'indexation) et simple à configurer/utiliser
- **Archiva** d'*Apache* (simple à configurer/utiliser)

2.1. Installation et configuration de "Sonatype Nexus 2"

Il suffit de télécharger l'archive "**nexus-2.1.2-bundle.zip**" et d'extraire son contenu sur une machine linux ou windows comportant une jvm java pour effectuer l'installation du produit.

Le numéro de port peut être facilement changé dans le fichier ***conf/nexus.properties*** (ex: ***8080*** → ***8484***) .

Le démarrage et l'arrêt du serveur peut s'effectuer via la commande "bin/**nexus start ou stop**"

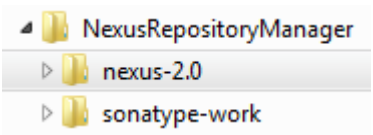
NB: sous windows , il est également possible de faire fonctionner **nexus** comme un s*ervice windows*. Pour cela il faut lancer l'instruction "**nexus install**" au sein d'une fenêtre de commande lancée en tant qu'administrateur (depuis raccourci et clic droit approprié).

L'URL menant à la console de nexus ressemble à "***http://localhost:8484/nexus***"

Par défaut (au moment de l'installation) , le mot de passe de l'administrateur (user="**admin**") est "**admin123**". Il peut être évidemment modifié par la suite.

Les accès ultérieurs seront effectués en mode anonyme (bridé en exploration/lecture) ou en mode "authentifié" (avec accès plus ou moins complet (selon rôle) pour administrer Nexus) .

Arborescence du produit Nexus2:



2.2. Configuration d'un référentiel sous Nexus

Par défaut (lors de l'installation) , Nexus est configuré avec les référentiels suivants:

- **releases** (pour les releases de notre entreprise/organisation)

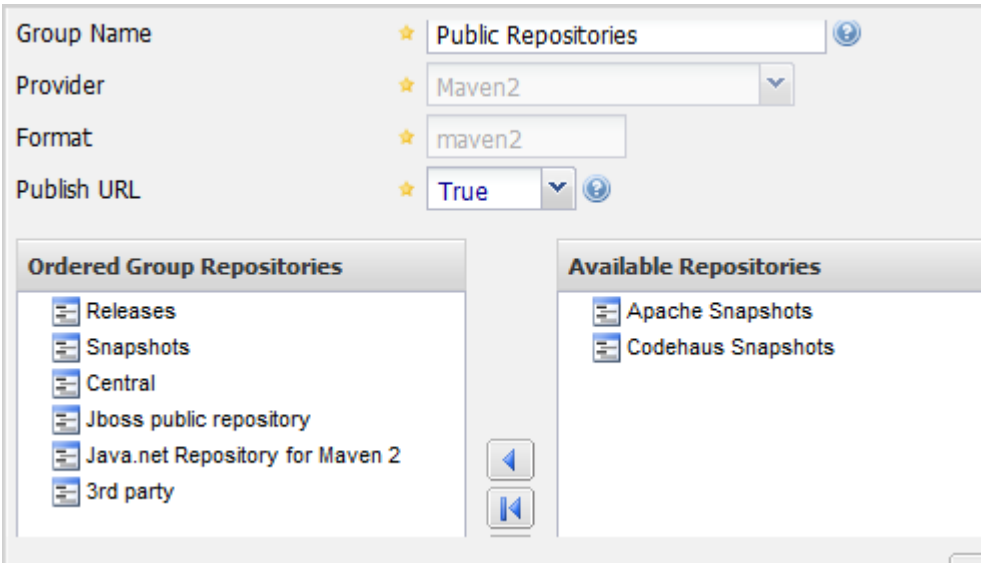
- **3rd party** (pour des ".jar" qui ne sont pas d'origine maven)
- **public** (groupe virtuel) vers "**central**" + "**…**" + *releases* + *3rdParty*

Ces référentiels peuvent être reconfigurés et on peut également créer d'autres référentiels .

L'administration (configuration) d'un référentiel peut se faire via la console web

Exemple : **Repositories** / **add...** (mode "**proxy**") , *repository_id* = ***"jboss.org"*** et :

Il faut en général penser également à ajouter les nouveaux référentiels dans le groupe "public" :



2.3. Utilisation de Nexus en "lecture/téléchargement"

Si l'on souhaite (comme souvent) , toujours passer indirectement par le référentiel "***public***" de nexus pour accéder aux référentiels externes (central maven , jboss , ...) , il faut commencer par éditer le fichier **settings.xml** de la façon suivante:

```
<settings>
  ...
  <mirrors>
```

```
    <mirror>
      <id>public</id>
      <url>http://localhost:8484/nexus/content/groups/public/</url>
 <!-- <mirrorOf>central,jboss.org,maven2-repository.dev.java.net</mirrorOf> -->
     <mirrorOf>*</mirrorOf>
    </mirror>
   /mirrors>
  ...
</settings>
```

Ceci permet de changer le comportement par défaut (qui habituellement consistait à interroger d'office en premier le référentiel "central" externe).

Il faut également ajouter la configuration d'un profil particulier "**nexus**" et activé d'office dans le fichier ***settings.xml*** :

```
<profiles>
  <profile>
    <id>nexus</id>
    <!--Enable snapshots for the built in central repo to direct -->
    <!--all requests to nexus via the mirror -->
    <repositories>
     <repository>
       <id>central</id>
       <url>http://repo1.maven.org/maven2</url>
       <releases><enabled>true</enabled></releases>
       <snapshots><enabled>true</enabled></snapshots>
     </repository>
    </repositories>
   <pluginRepositories>
     <pluginRepository>
       <id>central</id>
       <url>http://repo1.maven.org/maven2</url>
       <releases><enabled>true</enabled></releases>
       <snapshots><enabled>true</enabled></snapshots>
     </pluginRepository>
    </pluginRepositories>
  </profile>
 </profiles>
 <activeProfiles>
  <!--make the profile active all the time -->
  <activeProfile>nexus</activeProfile>
 </activeProfiles>
```

il faut également ajouter dans ***settings.xml*** les éléments de sécurité nécessaires:

```
<settings>  ...
  <servers>
     <server>
      <id>public</id>
      <username>admin</username>
      <password>admin123</password>
```

```
    </server>
  ...
  </servers>  ...
</settings>
```

2.4. Utilisation de Nexus en "écriture/alimentation"

Pour que "**mvn deploy**" puisse déployer l'artifact construit vers le référentiel géré par nexus il faut au minimum :

- Créer un (ou plusieurs) nouveau(x) **compte d'utilisateurs** au sein de nexus (avec [username, password] pour le déploiement. *(NB : en tp/formation on peut éventuellement utiliser le compte "admin/admin123")*
- Associer/affecter le rôle '***Nexus Deployment Role***' pour chaque user / repository vers lesquels on souhaite effectuer des déploiements.
- **Ajuster la sécurité dans 'settings.xml'**:

```
<settings> ...
 <servers>
  <server>
   <id>snapshots</id>  <username>admin</username> <password>admin123</password>
  </server>
  <server>
   <id>releases</id>
   <username>admin</username>    <!-- or specific deployment user -->
   <password>admin123</password> <!-- or specific deployment user password-->
  </server>
  ...
 </servers> ...
</settings>
```

il faut ensuite paramétrer (dans un ***pom.xml***) les **URLs pour le déploiement vers nexus** :

```
<project>
  ...
  <distributionManagement>
  <repository>
    <id>releases</id>
    <url>http://localhost:8484/nexus/content/repositories/releases</url>
  </repository>
  <snapshotRepository>
    <id>snapshots</id>
    <url>http://localhost:8484/nexus/content/repositories/snapshots</url>
  </snapshotRepository>
  </distributionManagement>
  ...
</project>
```

L'URL peut être formulée avec plusieurs protocoles supportés par nexus (selon contexte et O.S.) :

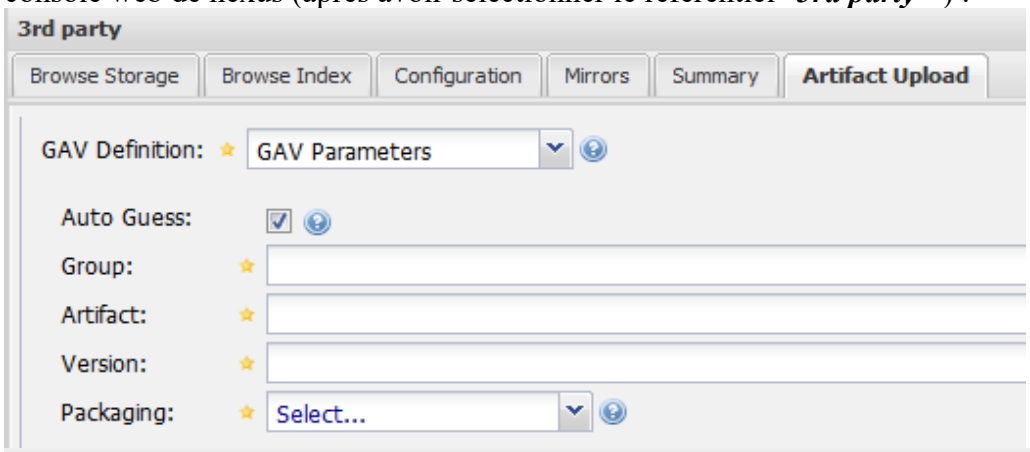
HTTP	`http://localhost:8484/nexus/content/repositories/releases`
WebDAV	`dav:http://....`

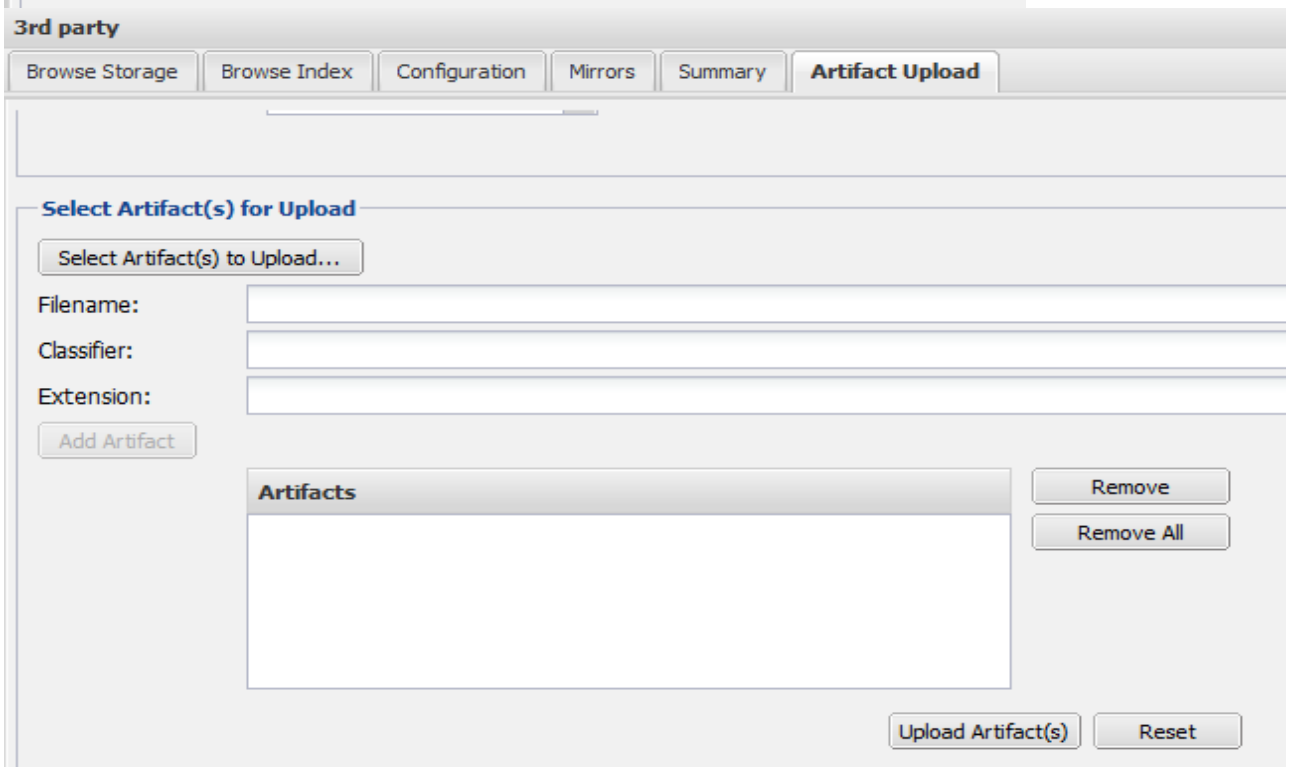
Il est éventuellement possible d'alimenter un référentiel à partir d'un artifact tierce-partie (".jar" issu d'un projet non maven):

```
mvn deploy:deploy-file -Dfile=filename.jar -DpomFile=filename.pom
```

```
    -DrepositoryId=thirdparty
    -Durl=http://localhost:8484/nexus/content/repositories/thirdparty
```

Au lieu d'utiliser les lignes de commandes "mvn deploy" ou "mvn deploy:deploy-file" on peut aussi alimenter un référentiel prise en charge par archiva en passant par l'onglet "**artifact upload**" de la console web de nexus (après avoir sélectionner le référentiel "***3rd party"***) :





2.5. Autres fonctionnalités de nexus

- Suppression d'un artifact maven via le menu contextuel "**delete**" de la console web
- Rendre caduque la valeur en cache/proxy pour forcer un nouveau futur téléchargement via le menu contextuel "**Expire Cache**" *[NB : ceci est très pratique et utile dans le cas où une première tentative de téléchargement a échoué suite à un problème de communication réseau / xxx.pom présent mais xxx.jar absent]*
- Navigation & recherche dans un référentiel
- Paramétrer un proxy http à usage interne (*Administration/server/default http proxy*)

3. Profils "maven"

Profils "maven" (utilité , configuration)

Pour **personnaliser une configuration d'un projet *selon certaines spécificités d'un environnement d'exécution*** *(ex: selon version jdk , selon o.s. , selon variable d'environnement, ...)* tout en gardant une bonne ***portabilité*** au niveau du fichier ***pom.xml*** , on peut paramétrer ***différents profils en parallèle*** (avec des ***variantes*** dans la ***configuration***).

Lors d'une invocation de maven , le **profil** adéquat sera automatiquement **activé** (en fonction du contexte ou d'un certain paramétrage).

```
<profiles>  <profile>
   <activation>   ....  </activation>
   ...
</profile></profiles>
```

Types de configuration possible des profiles "maven":

Portée	*Localisation de la configuration*
projet	**pom.xml** du projet
utilisateur	**%USER_HOME%/.m2/settings.xml**
global	**%M2_HOME%/conf/settings.xml**
Selon conf "profiles maven 3"	....

3.1. Activation selon la version de java

```
<profiles>
  <profile>
    <activation>
      <jdk>1.4</jdk>
    </activation>
    ...
  </profile>
</profiles>
```

Et depuis la version 2.1 , possibilité d'exprimer des plages :

```
<profiles>
  <profile>
    <activation>
      <jdk>[1.3,1.6)</jdk>  <!-- du 1.3 au 1.5 (1.6 exclus) -->
    </activation>
    ...
  </profile>
</profiles>
```

3.2. Activation selon le système d'exploitation (OS):

```
<profiles>
  <profile>
    <activation>
      <os>
        <name>Windows XP</name>
        <family>Windows</family>
        <arch>x86</arch>
        <version>5.1.2600</version>
      </os>
    </activation>
    ...
  </profile>
</profiles>
```

3.3. Activation selon une propriété système java

(paramètre -Dxx.yy d'une ligne de commande mvn groupId:artifactId:goal)

```
<profiles>
  <profile>
    <activation>
      <property>
        <name>debug</name>
        <!-- isDefined , any value -->
      </property>
    </activation>
    ...
  </profile>
</profiles>

<profiles>
  <profile>
    <activation>
      <property>
        <name>environment</name>
        <value>test</value>      <!-- if -Denvironment=test -->
      </property>
    </activation>
    ...
  </profile>
</profiles
```

Pour activer selon variable d'environnement:
-Dxxx.yyy=%VAR_XX_YY% dans .bat
ou -Dxxx.yyy=${VAR_XX_YY} dans .sh

3.4. Activation selon un fichier manquant ou existant

```
<profiles>
  <profile>
    <activation>
      <file>
        <missing>target/generated-sources/xxx/yyy</missing>
        <!-- ou bien  <exists>...</exists> -->
      </file>
    </activation>
    ...
  </profile>
</profiles>
```

3.5. Profils nommés à activer explicitement

```
<profiles>
  <profile>
    <id>profile-1</id>

    ...
  </profile>
  <profile>
    <id>profile-2</id>

    ...
  </profile>
</profiles>
```

et

mvn groupId:artifactId:goal -P profile-n

Eventuel profil nommé et activé "en dur":

```
<settings>
  ...
  <activeProfiles>
    <activeProfile>profile-1</activeProfile>
  </activeProfiles>
  ...
</settings>
```

ou encore

```
<profiles>
  <profile>
    <id>profile-1</id>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    ...
  </profile>
</profiles>
```

3.6. Exemple concret (switch de configuration)

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>....</groupId>
        <artifactId>....n</artifactId>
        <version>1.0</version>
        <configuration>
          <appserverHome>${appserver.home}</appserverHome>
        </configuration>
      </plugin>
      ...
    </plugins>
  </build>
  ...
</project>
```

avec ***${appserver.home}*** ***défini alternativement en "/path/to/dev/appserver" ou "/path/to/dev/appserver2" selon -Denv=dev ou bien -Denv=dev2 .***

```
<project>
  ...
  <profiles>
    <profile>
      <id>appserverConfig-dev</id>
      <activation>
        <property>
          <name>env</name>
          <value>dev</value>
        </property>
      </activation>
      <properties>
        <appserver.home>/path/to/dev/appserver</appserver.home>
      </properties>
    </profile>

    <profile>
      <id>appserverConfig-dev-2</id>
      <activation>
        <property>
          <name>env</name>
          <value>dev-2</value>
        </property>
      </activation>
      <properties>
        <appserver.home>/path/to/another/dev/appserver2</appserver.home>
      </properties>
    </profile>
  </profiles>
  ..
</project>
```

4. Filtrage des ressources

Principe:
Certains fichiers de ressources (.properties , .xml) peuvent éventuellement comporter des valeurs basées sur des **variables qui seront renseignées lors de la construction via maven**.
Ceci permet d'obtenir ***plusieurs configurations différentes*** (pour les bases de données par exemple) e***n fonction d'un choix de profile***.

Exemples:

exemple.properties (dans src/main/resources)

```
# les valeurs seront remplacees par celles qui seront choisies
# dans la configuration du profile maven actif
# en mode "filtering resource" (voir resultats dans "target")
xxx.yyy=${xxx.yyy}
xxx.zzz=${xxx.zzz}
```

exemple.xml (dans src/main/resources)

```
<exemple>
  <xxx_yyy>${xxx.yyy}</xxx_yyy>
  <xxx_zzz>${xxx.zzz}</xxx_zzz>
</exemple>
```

Configuration maven:

```
<project ...>. ...
  <profiles>
  <profile> <id>p1</id>
          <properties>
                  <xxx.yyy>jetty</xxx.yyy>    <!-- valeur de remplacement pour ${xxx.yyy} -->
                   <xxx.zzz>jetty2</xxx.zzz>
          </properties>
    </profile>
    <profile> <id>p2</id>
          <properties>
                  <xxx.yyy>tomcat</xxx.yyy>
                   <xxx.zzz>tomcat6</xxx.zzz>
          </properties>
    </profile>
  </profiles>  ...
  <build><plugins> ... </plugins>
          <resources>
            <resource>
               <directory>src/main/resources</directory>
               <filtering>true</filtering>
            </resource>
          </resources>
  </build>
</project>
```

NB: Pour activer le **filtrage de ressources** sur la partie "***tests unitaires***" , il faut ajouter (dans ***pom.xml***) la configuration suivante :

```
<testResources>
    <testResource>
        <directory>src/test/resources</directory>
        <filtering>true</filtering>
    </testResource>
</testResources>
```

en plus de <resources><resource> … pour la partie src/main/resources

5. Ajout (et éventuel filtrage) de ressources "web" externes

```
<project>  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-war-plugin</artifactId>
        <version>2.1.1</version>
        <configuration>
          <webResources>
            <resource>
              <!-- this is relative to the pom.xml directory -->
              <directory>resource2</directory>
            </resource>
          </webResources>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

---> **effet/résultats (dans target)** : tout le contenu du répertoire "resource2" est ajouté au contenu de src/main/webapp .

Possibilités de <includes> , <excludes> :

```
...
        <configuration>
          <webResources>
            <resource>
              <directory>resource2/</directory>
              <!-- the list has a default value of ** -->
              <includes>
                <include>image2/*.jpg</include>
              <includes>
              <!-- there's no default value for this -->
              <excludes>
                <exclude>**/*.jpg</exclude>
              </excludes>
            </resource>
          </webResources>
        </configuration>
        ...
```

Via **<targetPath>...</targetPath>** , un contenu externe peut être ajouté à un endroit précis (ex:

WEB_INF) .

D'autre part , certaines variables de certains fichiers de configurations peuvent être filtrées (c'est à dire remplacées par des valeurs paramétrables).

Exemple:

via le filtre

configurations/properties/**config.prop**

```
interpolated_property=some_config_value
```

la valeur de la variable ${interpolated_property} du fichier de configuration suivant sera automatiquement remplacée:

configurations/**config.cfg**

```
<another_ioc_container>
  <configuration>${interpolated_property}</configuration>
</another_ioc_container>
```

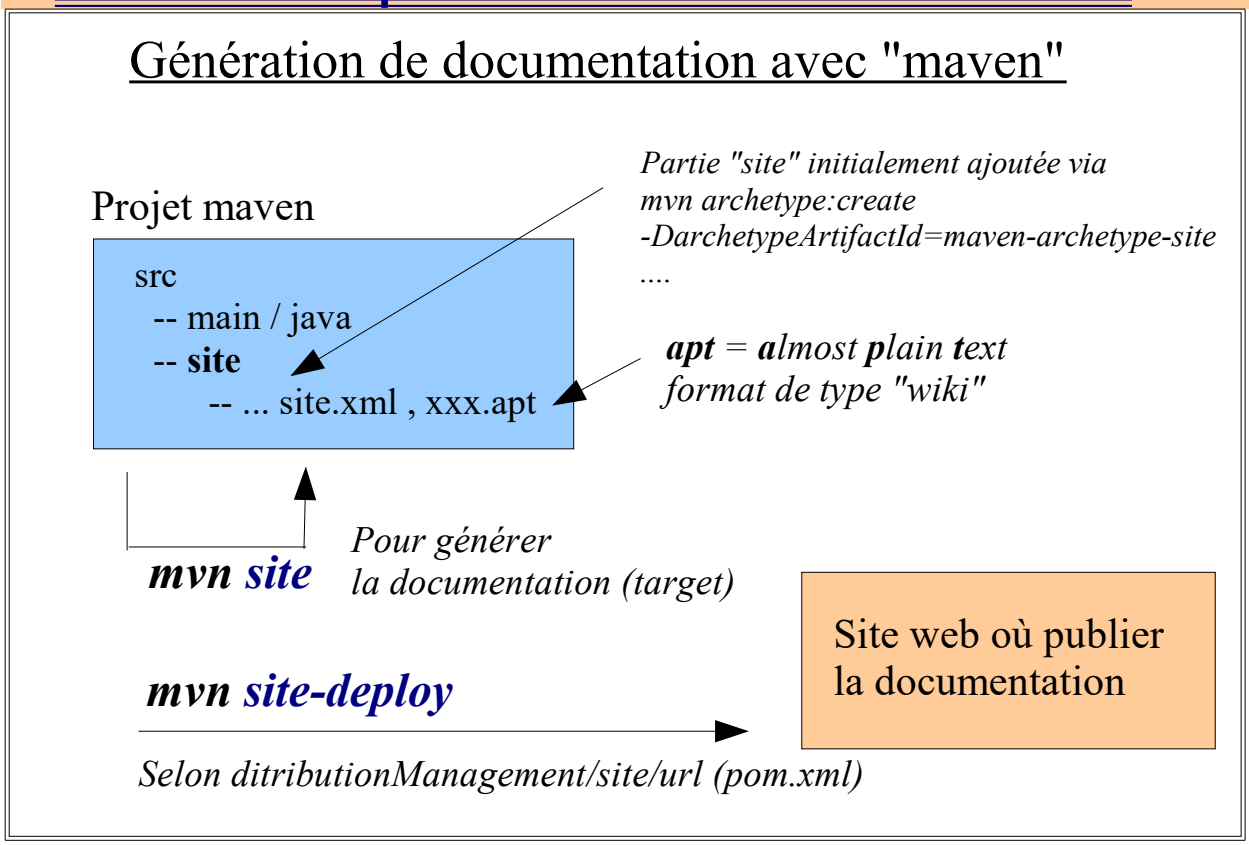
pom.xml

```
...
      <configuration>
        <filters>
         <filter>properties/config.prop</filter>
        </filters>
        <nonFilteredFileExtensions>
          <!-- default value contains jpg,jpeg,gif,bmp,png -->
          <nonFilteredFileExtension>pdf</nonFilteredFileExtension>
        </nonFilteredFileExtensions>
        <webResources>
          <resource>
            <directory>resource2</directory>
            <!-- it's not a good idea to filter binary files -->
            <filtering>false</filtering>
          </resource>
          <resource>
            <directory>configurations</directory>
             <!-- override the destination directory for this resource →
            <targetPath>WEB-INF</targetPath>
             <filtering>true</filtering>

            <excludes>
              <exclude>**/properties</exclude>
            </excludes>
          </resource>
      ...
```

VIII - Génération de documentation (mvn)

1. Génération et publication d'une documentation



Commande pour ajouter la partie "site" sur un projet maven existant:

```
mvn archetype:create \
  -DarchetypeGroupId=org.apache.maven.archetypes \
  -DarchetypeArtifactId=maven-archetype-site \
  -DgroupId=com.mycompany.app \
  -DartifactId=my-app-site
```

Arborescence générée:

```
my-app-site
|-- pom.xml
`-- src
    `-- site
        |-- apt
        |   |-- format.apt
        |   `-- index.apt
        |-- fml
        |   `-- faq.fml
        |-- fr
        |   |-- apt
        |   |   |-- format.apt
        |   |   `-- index.apt
        |   |-- fml
        |   |   `-- faq.fml
        |   `-- xdoc
        |       `-- xdoc.xml
        |-- xdoc
        |   `-- xdoc.xml
        |-- site.xml
```

```
        `-- site_fr.xml
```

NB: apt = "almost plain text" = format de type "wiki"
+ *voir la documentation de référence pour approfondir le sujet .*

Ajout de liens hypertextes dans **site.xml**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="Maven" xmlns="http://maven.apache.org/DECORATION/1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/DECORATION/1.0.0
http://maven.apache.org/xsd/decoration-1.0.0.xsd">

  <body>
    <links>
      <item name="Apache" href="http://www.apache.org/" />      ...
    </links>

    <menu name="Maven 2.0">
       <item name="FAQ" href="faq.html"/>
      <item name="javadoc (api)" href="apidocs/index.html"/>
      <item name="checkstyle-report" href="checkstyle.html"/>
      <item name="jdepend-report" href="jdepend-report.html"/>
    </menu>
  </body>
</project>
```

pom.xml

```
<project>
  ....
  <distributionManagement>
    <site>
      <id>website</id>
      <url>scp://webhost.company.com/www/website</url>
      <!-- ou en file: si répertoire distant partagé via nfs ou autre -->
    </site>
  </distributionManagement>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-site-plugin</artifactId>
        <configuration>
          <locales>en,fr</locales>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

mvn site *(pour générer la doc html)*

mvn site-deploy *(pour déployer la doc générée)*

IX - Rapports (et javadoc) avec maven

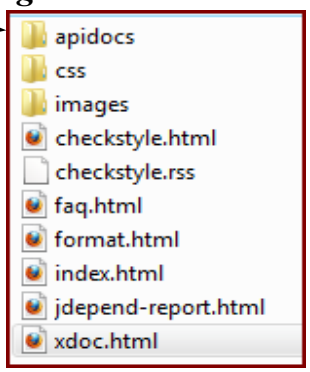
1. Javadoc (via maven)

Générer **javadoc** avec "maven"

```
... (dans pom.xml)
<reporting>
 <plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-javadoc-plugin</artifactId>
    <version>2.7</version>
    <!-- <configuration>
      <minmemory>128m</minmemory>
      <maxmemory>512m</maxmemory>
    </configuration> -->
  </plugin>
   ...
</reporting>
```

mvn javadoc:javadoc

target/site/



2. Rapports avec maven

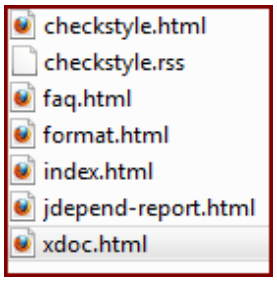
rapports avec "maven"

```
... (dans pom.xml)
<reporting>
 <plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-checkstyle-plugin</artifactId>
    <version>2.6</version>
  </plugin>
  <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>jdepend-maven-plugin</artifactId>
      <version>2.0-beta-2</version>
  </plugin>
  <!-- maven-pmd-plugin not found -->
 </plugins>
</reporting>
```

mvn checkstyle:checkstyle

mvn jdepend:generate

target/site/



X - Plugins pour maven (exécution, prog.)

1. Plugins pour maven

Plugins "maven" (utilité , structure)

• Les mécanismes internes de maven **délèguent** toutes les ***sous tâches spécialisées*** à des "***plugins***" (ex: un plugin pour "***clean***" , un autre plugin pour packager le code sous forme de fichier zip [".war", ".jar" , ".ear" , ...]).
• Il est possible de **programmer** de nouveaux ***plugins personnalisés*** et de les configurer pour qu'ils soient activés.

• Le ***développement d'un plugin*** s'effectue généralement sous la forme d'une **classe java** ("***MOJO***" : *M*aven P*OJO*) comportant une méthode "***execute***()" et codée au sein d'un projet maven de type "plugin" .

• L'***activation*** s'effectue souvent en ***associant un but d'un plugin en tant que tâche supplémentaire à exécuter lors d'une des phases de construction (ex: "compile" , "package" , ...)***

1.1. execution

Configuration d'un plugin "maven" à activer

```
<build>
  <plugins>
   <plugin>
    <groupId>sample.plugin</groupId>
    <artifactId>hello-maven-plugin</artifactId>
    <version>1.0-SNAPSHOT</version>
    <executions>
     <execution>
      <phase>compile</phase>
      <goals>
       <goal>sayhi</goal>
      </goals>
     </execution>
    </executions>
   </plugin>
  </plugins>
 </build>
```

1.2. Liste des principaux plugins de maven

** (Type): **B**uild or **R**eporting plugin*

Plugin	Type *	Version	Release Date	Description
Core plugins				**Plugins corresponding to default core phases (ie. clean, compile). They may have muliple goals as well.**
clean	B	2.4.1	2010-05-12	Clean up after the build.
compiler	B	2.3.2	2010-09-09	Compiles Java sources.
deploy	B	2.5	2009-12-24	Deploy the built artifact to the remote repository.
failsafe	B	2.8	2011-03-13	Run the Junit integration tests in an isolated classloader.
install	B	2.3.1	2010-05-21	Install the built artifact into the local repository.
resources	B	2.5	2011-02-27	Copy the resources to the output directory for including in the JAR.
site for Maven 2	B	2.2	2010-11-28	Generate a site for the current project.
site for Maven 3	B	3.0-beta-3	2010-10-18	Generate a site for the current project.
surefire	B	2.8	2011-03-13	Run the Junit unit tests in an isolated classloader.
verifier	B	1.0	2010-01-30	Useful for integration tests - verifies the existence of certain conditions.
Packaging types / tools				**These plugins relate to packaging respective artifact types.**
ear	B	2.5	2011-01-29	Generate an EAR from the current project.
ejb	B	2.3	2010-09-18	Build an EJB (and optional client) from the current project.
jar	B	2.3.1	2010-05-21	Build a JAR from the current project.
rar	B	2.2	2007-02-28	Build a RAR from the current project.
war	B	2.1.1	2010-11-03	Build a WAR from the current project.
app-client	B	1.0	2011-03-31	Build a JavaEE application client from the current project.
shade	B	1.4	2010-08-11	Build an Uber-JAR from the current project, including dependencies.

Reporting plugins				**Plugins which generate reports, are configured as reports in the POM and run under the site generation lifecycle.**
changelog	R	2.2	2010-05-28	Generate a list of recent changes from your SCM.
changes	B+R	2.4	2011-01-29	Generate a report from issue tracking or a change document.
checkstyle	B+R	2.6	2010-09-25	Generate a checkstyle report.
doap	B	1.1	2011-01-15	Generate a Description of a Project (DOAP) file from a POM.
docck	B	1.0	2008-11-16	Documentation checker plugin.
javadoc	B+R	2.7	2010-05-04	Generate Javadoc for the project.
jxr	R	2.2	2010-06-05	Generate a source cross reference.
linkcheck	R	1.1	2010-11-13	Generate a Linkcheck report of your project's documentation.
pmd	B+R	2.5	2010-05-04	Generate a PMD report.
project-info-reports	R	2.3.1	2010-12-21	Generate standard project reports.
surefire-report	R	2.8	2011-03-13	Generate a report based on the results of unit tests.
Tools				**These are miscellaneous tools available through Maven by default.**
ant	B	2.3	2009-11-11	Generate an Ant build file for the project.
antrun	B	1.6	2010-10-11	Run a set of ant tasks from a phase of the build.
archetype	B	2.0	2010-10-28	Generate a skeleton project structure from an archetype.
assembly	B	2.2.1	2011-02-27	Build an assembly (distribution) of sources and/or binaries.
dependency	B+R	2.2	2011-02-22	Dependency manipulation (copy, unpack) and analysis.
enforcer	B	1.0	2010-11-08	Environmental constraint checking (Maven Version, JDK etc), User Custom Rule Execution.
gpg	B	1.2	2011-03-24	Create signatures for the artifacts and poms.
help	B	2.1.1	2010-03-26	Get information about the working environment for the project.
invoker	B	1.5	2009-	Run a set of Maven projects and verify

			10-26	the output.
jarsigner	B	1.2	2009-09-30	Signs or verifies project artifacts.
one	B	1.2	2007-09-12	A plugin for interacting with legacy Maven 1.x repositories and builds.
patch	B	1.1.1	2010-01-06	Use the gnu patch tool to apply patch files to source code.
pdf	B	1.1	2009-12-13	Generate a PDF version of your project's documentation.
plugin	B+R	2.7	2011-02-09	Create a Maven plugin descriptor for any mojos found in the source tree, to include in the JAR.
release	B	2.1	2010-10-08	Release the current project - updating the POM and tagging in the SCM.
reactor	B	1.0	2008-09-27	Build a subset of interdependent projects in a reactor
remote-resources	B	1.2	2011-02-27	Copy remote resources to the output directory for inclusion in the artifact.
repository	B	2.3.1	2010-07-21	Plugin to help with repository-based tasks.
scm	B	1.4	2010-08-08	Generate a SCM for the current project.
source	B	2.1.2	2010-05-21	Build a JAR of sources for use in IDEs and distribution to the repository.
stage	B	1.0-alpha-2	2009-07-14	Assists with release staging and promotion.
toolchains	B	1.0	2009-11-01	Allows to share configuration across plugins.
IDEs				**Plugins that simplify integration with integrated developer environments.**
eclipse	B	2.8	2010-02-25	Generate an Eclipse project file for the current project.
idea	B	2.2	2008-08-08	Create/update an IDEA workspace for the current project (individual modules are created as IDEA modules)

2. Programmation d'un plugin pour maven

2.1. Plugin maven développé en java ("mojo")

Plugin "maven" basé sur java ("mojo")

```
/**
 * @goal touch
 * @phase process-sources
 */
class XxxMojo extends AbstractMojo {

/**
* @parameter expression="${project.build.directory}"
* @required
*/
private File outputDirectory;

public void execute(){
      ...
      }
}
```

NB: **Un plugin important (packagé comme un ".jar") est souvent constitué de plusieurs classes complémentaires** :

- classe "XxxMojo1" pour le but 1
- classe "XxxMojo2" pour le but 2
- classe "XxxMojoN" pour le but N
- ....
- classes "CommonUtil1" , "CommonUtilN" pour le code commun partagé entre les différents "buts/goals" .

Création d'un projet "plugin":

```
mvn  archetype:generate
        -DarchetypeArtifactId=maven-archetype-mojo
        -DgroupId=com.mycompany.plugin1
        -DartifactId=myplugin1-maven-plugin
```

==> Coder/Paramétrer la classe java
et construire le plugin via "**mvn install**"

Utilisation (invocation) directe:

```
REM  mvn goupIdOfPlugin:articfactIdOfPlugin:goalName
mvn  com.mycompany.plugin1:myplugin1-maven-plugin:touch
```

Utilisation dans la construction d'un autre projet:

```
<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.mycompany.app1</groupId>
  <artifactId>my-java-app1</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  ...
  <dependencies>...  </dependencies>
   <build>
     <plugins>
       <plugin>
         <groupId>com.mycompany.plugin1</groupId>
         <artifactId>myplugin1-maven-plugin</artifactId>
         <version>1.0-SNAPSHOT</version>
         <executions>
          <execution>
          <!-- <phase>process-sources</phase> already defined with @phase process-sources -->
          <!-- <configuration>
           </configuration> -->
          <goals>
             <goal>touch</goal>
          </goals>
         </execution>
       </executions>
       </plugin>
   </plugins>
  </build>

</project>
```

2.2. Plugin maven basé sur ant

Plugin "maven" basé sur l'intégration d'un script "ant"

Dans ***src/main/scripts***:

xxx.build.xml (le script ant)

xxx.mojos.xml *(le document de liaison "maven goal – ant target")*

```
<pluginMetadata>
 <mojos> <mojo>
   <goal>hello</goal>
   <call>hello</call> <!-- Ant -->
   <description> ...</description>
  </mojo> </mojos>
</pluginMetadata>
```

pom.xml

```
<project>...  <packaging>maven-plugin</packaging>
 <dependencies> <dependency>
   <groupId>org.apache.maven</groupId><artifactId>maven-script-ant</artifactId>
   <version>2.0.6</version> </dependency> </dependencies>
  <build><plugins> <plugin>
    <artifactId>maven-plugin-plugin</artifactId> <version>2.5</version>
     <dependencies> <dependency>
      <groupId>org.apache.maven.plugin-tools</groupId>
      <artifactId>maven-plugin-tools-ant</artifactId> <version>2.5</version>
     </dependency></dependencies>
    <configuration> <goalPrefix>hello</goalPrefix> </configuration>
   </plugin> </plugins> </build></project>
```

XI - GConf. maven (lien avec SVN, ...)

1. Plugins "scm" et "release" de maven

1.1. Liaison avec un référentiel de code source SVN

Lien entre "maven" et le référentiel de code source (scm)

```
<project>  ...
  <scm>    < !-- scm = configuration utilisée par les plugins "scm" et "release" -->
    <!-- ro -->
    <connection>scm:svn:http://127.0.0.1/svn/my-project</connection>

    <!-- rw -->
    <developerConnection>scm:svn:https://127.0.0.1/svn/my-project
    </developerConnection>
    <tag>HEAD</tag>
    <url>http://127.0.0.1/websvn/my-project</url>
  </scm>
  … </project>
```

Ou bien url de type :
scm:git:file:///media/sf_ext/tp/local-git-repositories/env-ic-my-java-app1
ou encore **scm:git:http://www.xy.com/my-repo.git**
ou encore **scm:cvs** :... , **scm:hg**:url_repo_mercurial

Le plugin maven "scm" permet de gérer uniformément "csv" , "svn" , "git" ou "mercurial" .

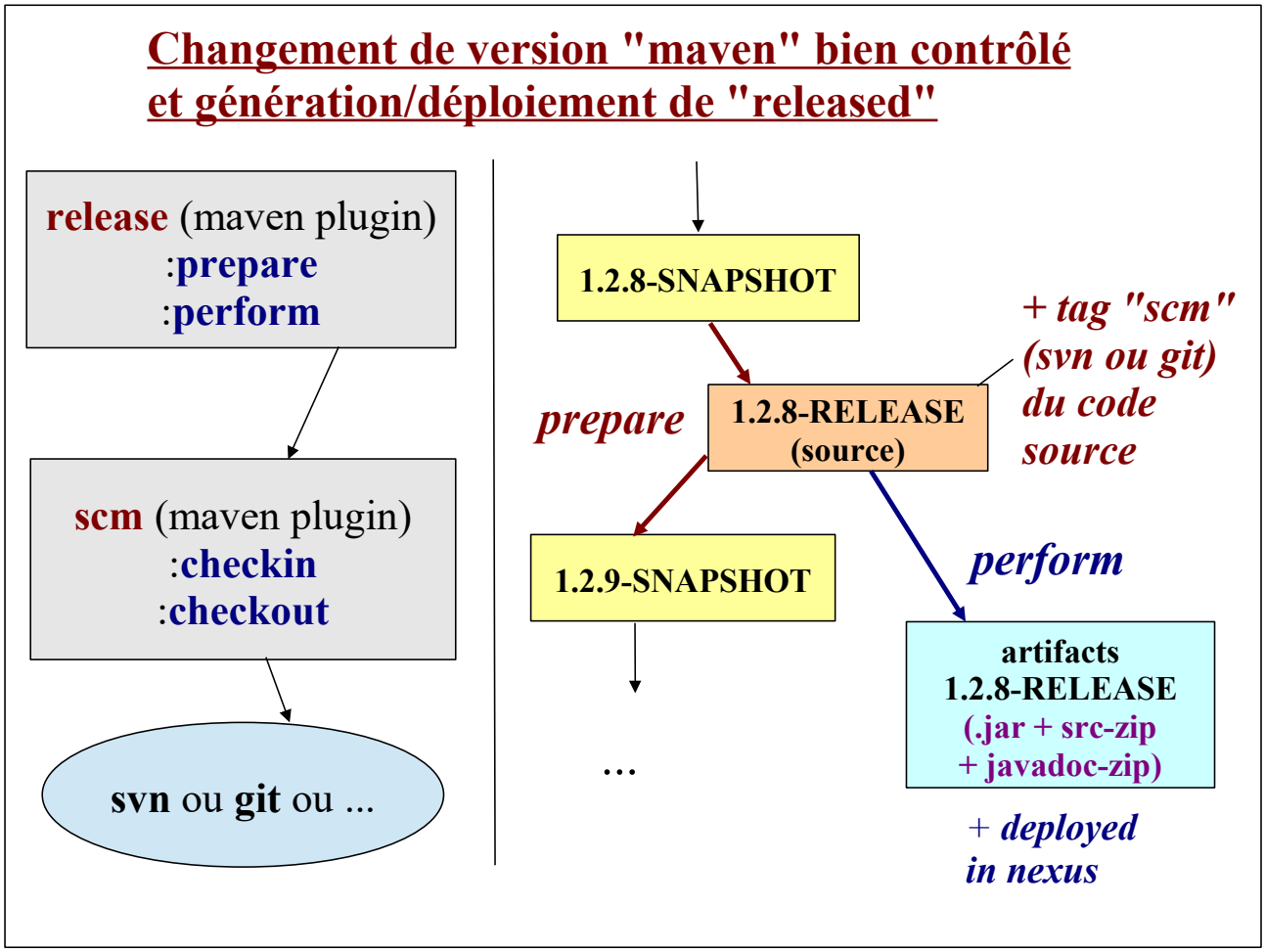
mvn scm:*goal_xy* ...

scm:checkin - command for commiting changes
scm:checkout - command for getting the source code
scm:status - command for showing the scm status of the working copy
scm:update - command for updating the working copy
with the latest changes
scm:tag - command for tagging a certain revision

===> étudier la documentation de référence pour consulter les détails.

NB : Le principal intérêt du plugin "scm" de maven tient dans le fait qu'il est a son tour réutilisé par un plugin de plus haut niveau intitulé "release" .

1.2. Gestion des "releases" pilotées par maven



Le principal intérêt du plugin "scm" de maven tient dans le fait qu'il est a son tour réutilisé par un plugin de plus haut niveau intitulé "release" .
Le plugin "**release**" sert à changer de version "maven" (ex : 1.1-SNAPSHOT vers 1.1-RELEASE) de façon cohérente à tous les niveaux (pom.xml de l'application , des sous modules , dans SCM).

Préparation d'une version "released" via maven

(a) ***mvn_scm_release_prepare_my_java_app1.sh***

```
cd my-java-app1
# vérifications effectuées par release:prepare :
#      * pas de modifications locales (sans commit) : même en TP !!!
#      * pas de dépendances vers des xxx-SNAPSHOT
# informations suggérées / demandées (que l'on peut saisir si pas -B):
#      * new mvn/pom relased version (ex: 1.2.8.-RELEASE)
#      * new SCM(git or svn or ...) released tg version (ex: my-java-app1-v1.2.8)
#      * new mvn/pom developpement version (ex: 1.2.9-SNAPSHOT)
# actions qui seront exécutées (si aucune erreur):
#      * met à jour "version realeased" dans le pom
#        (après sauvegarde dans pom.xml.releaseBackup)
#      * build & test with new version (clean verify)
#      * tag and commit released version in SCM(git or svn or ...)  et génère des lignes
#        dans release.properties (pour futur release:perform ou  release:rollback)
#      * update to new developpement version  (ex: 1.2.9-SNAPSHOT) in the pom
#      * commit in SCM
mvn  -B  release:prepare
# -B or --batch-mode is for non interactive mode (utile pour integration continue)
echo "fin"; read fin
```

Génération/construction d'une version "released" via maven

(b) ***mvn_scm_release_perform_my_java_app1.sh***

```
cd my-java-app1
# a lancer après "mvn release:prepare"
# utilise release.properties ( normalement généré par release:prepare) pour :
#       vérifier que  release:prepare s'est déroulé sans erreur
#       récupération / checkout (from "tag" ) des sources de la version "released"
#       depuis le scm   (svn ou git ou …)  et construction des "artifacts".
#       déploiement des "artifacts" en version"released" vers le référentiel maven
#       (ex: nexus)
#       trois ".jar" sont construits et déployés (artifact.jar , artifact-sources.jar ,
#                                                 artifact-javadoc.jar)
mvn release:perform
echo "fin"; read fin
```

NB: il existe "mvn release:rollback" en cas d'échec
de "mvn release:prepare" .

XII - Maven et intégration continue

1. Intégration continue

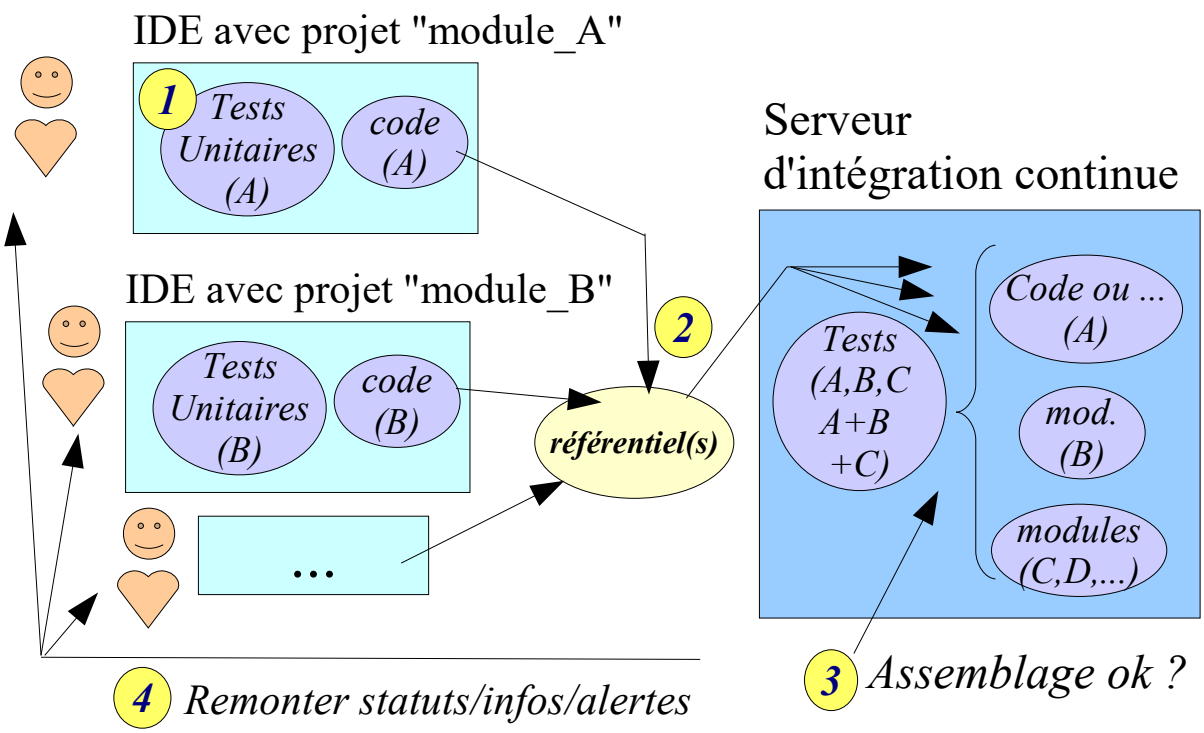
Intégration continue

Principes (intégration continue):
En plus des tests unitaires (à portée limitée à un module) , il est généralement nécessaire de ***régulièrement tester l'ensemble d'une application (tests d'intégration)***.

Il existe pour cela, des produits spécialisés dit "***d'intégration continue***" dont les principales fonctionnalités sont:

* récupérer les différents modules (src ou jar) selon version.
* recompiler certaines parties pour valider les liaisons.
* lancer quelques tests pour valider le bon comportement de l'assemblage (dans le cadre d'une certaine nouvelle version).
* remonter des statuts/alertes aux développeurs/... concernés (ex: succès / échecs des tests , nombre d'erreurs , ...).

Intégration continue



2. Maven et l'intégration continue

Maven et intégration continue

Principaux logiciels d'intégration continue (monde Java):
- **CruiseControl** (le pionnier)
- **Hudson** (évolution récente: **Jenkins**)
- **Continuum** (apache)

Ces différents logiciels sont (entre autres) capables de **s'appuyer sur "maven"** pour automatiser la récupération des nouvelles versions des modules, les compilations et les lancements des tests.

....

--> consulter éventuellement une des annexes pour approfondir la mise en oeuvre d'un serveur d'intégration continue.

NB: le serveur d'intégration "***Hudson/Jenkins***" est aujourd'hui l'un des plus populaire car il très facile à installer/configurer/utiliser .

ANNEXES

XIII - Annexe – variables "maven" et versions

1. Variables et versions (maven)

1.1. Variables prédéfinies de maven

NB: Tous les éléments (xml) présents dans le fichier ***pom.xml***, peuvent être référencés via le préfixe "**project.**" (ou l'ancien "pom." maintenant obsolète).
D'autre part, tous les éléments (xml) présents dans le fichier ***settings.xml***, peuvent être référencés via le préfixe "***settings.***"

Depuis Maven 3.0, toutes les propriétés "`pom.*`" sont "deprecated".
Il faut utiliser les propriétés en "`project.*`" à la place .

variables	***Significations (contenus)***
${basedir}	Répertoire contenant le fichier pom.xml
${version} (équivalent à ${project.version})	Version du projet courant
${project.build.directory}	Répertoire "target"
${project.build.outputDirectory}	Répertoire "target/classes"
${project.build.finalName}	Nom du fichier créé (xxx.war , xxx.jar)
${project.xxx.yyy}	Valeur de <xxx><yyy>...</yyy></xxx> dans pom.xml
...	
${settings.localRepository}	Référentiel local de l'utilisateur
${settings.xxx.yyy}	Valeur de <xxx><yyy>...</yyy></xxx> dans settings.xml
...	
${env.XXX}	Valeur de la variable d'environnement XXX
...	
${java.home} , ${java.version} , ...	JRE_HOME , ...
${user.name} , ${user.home} , ...	Toutes les "propriétés systèmes" de java
...	
`${project.parent.xxx}`	Propriétés du projet "parent"
...	

1.2. Versions des artifacts "maven"

Par convention, **une pré-version en cours de développement d'un projet** voit son numéro de version suivi d'un **-SNAPSHOT**.

Dans la gestion des dépendances, Maven va chercher à mettre à jour les versions SNAPSHOT régulièrement pour prendre en compte les derniers développements.

Utiliser une version SNAPSHOT permet de bénéficier des dernières fonctionnalités d'un projet, mais en contre-partie, cette version peut être appelée à être modifiée de façon importante, sans aucun préavis.

Exemples:

0.0.1-SNAPSHOT
1.0-SNAPSHOT
2.0-SNAPSHOT

Structure habituelle d'un numéro de version:

<major>.<mini>[.<micro>][-<qualifier>[-<buildnumber>]]

Incréments
Major : changement majeur
pas de rétro-compatibilité (descendante) garantie
Mini : ajouts fonctionnels
rétro-compatibilité garantie
Micro : maintenance corrective (*bug fix*)

1.3. Création d'une "release" en ligne de commande via le plugin "release"

La page ***guide-releasing.html*** de la documentation de référence "maven" explique comment créer proprement une "release" via un plugin adéquat .

2. Bonnes pratiques

Les noms des packages java d'une application doivent idéalement commencer par la valeur du "groupId" .

Exemple : si groupId = "*tp.myapp.xy*"
alors packages
"*tp.myapp.xy*.itf.domain.dto" , "*tp.myapp.xy*.itf.domain.service" ,
"*tp.myapp.xy*.impl.domain.service" , "*tp.myapp.xy*.impl.persistence.dao" , "
tp.myapp.xy.impl.persistence.entity" , …

Décomposer une application en plein de sous modules et de projet annexes :

Exemple :

app
- app-services-itf *(.jar , interfaces des services et structures dto)*
- app-services-local-impl *(.jar , implémentation locale avec spring/hibernate/jpa)*
- app-services-remote-delegate *(.jar ,business delegate vers web services distants)*
- app-web *(.war , ihm web)*

my-test-framework *(.jar , petit framework de test basé sur Junit et DBUnit)*
my-persistence-framework *(.jar , DAO génériques basés sur Hibernate ou JPA)*
my-xy-framework *(.jar , autre petit framework réutilisable)*

XIV - Annexe – Plugin "cargo"

1. Déploiement Jee avec CARGO (via maven)

Le produit "CARGO" de "CodeHaus" permet d'effectuer des déploiements d'application "Jee" vers différents serveurs/"conteneurs web" : Tomcat, JBoss , WebSphere , WebLogic .

CARGO peut (selon le type de serveur) gérer le conteneur web dans 1,2 ou 3 des trois grands modes suivants:

- "**embbeded**" (en mémoire dans même JVM -- possible avec "Jetty")
- "**installed**" (local)
- "**remote**"

Le déclenchement de "cargo" peut s'effectuer d'une des 3 façons suivantes:

- par code java (via API spécifique)
- via un script "ant
- via maven

La suite de cette annexe présente l'utilisation de "cargo" via maven.

Le déclenchement (depuis maven) du déploiement JEE (et du lancement du serveur) se fait via:

```
mvn  cargo:start
```

C'est à peu près l'équivalent "maven" du "run as / run on server" d'eclipse.

1.1. Configuration pour Tomcat 6 (en mode local/installed)

```
...
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.cargo</groupId>
      <artifactId>cargo-maven2-plugin</artifactId>
      <version>1.4.12</version>
       <!-- tomcat6x or tomcat7x (not tomcat6 / tomcat7) -->
       <!-- installed or remote : ok with tomcat , embedded ok only with Jetty  -->
       <!-- configuration "existing" nécessaire (en plus de installed) !!! -->
       <configuration>
         <wait>true</wait>          <!-- waiting for Ctrl-C -->
         <container>
            <containerId>tomcat6x</containerId>
             <type>installed</type>
             <home>C:\Prog\java\ServApp\Tomcat_6.0</home>
         </container>
         <configuration>
               <type>existing</type>
               <home>C:\Prog\java\ServApp\Tomcat_6.0</home>
         </configuration>
      </configuration>
   </plugin>
 </plugins>
</build>
```

1.2. Configuration pour Jetty (en mode "embedded")

```
...
```

```
<dependencies>
  <dependency>
      <groupId>javax.el</groupId>   <artifactId>el-api</artifactId>
      <version>2.2</version>    <scope>provided</scope>
 </dependency>
 <dependency>
      <groupId>org.glassfish.web</groupId>   <artifactId>el-impl</artifactId>
      <version>2.2</version>   <scope>provided</scope>
 </dependency>
     ....
<dependencies>

<build>
 <plugins>
  <plugin>
    <groupId>org.codehaus.cargo</groupId>
    <artifactId>cargo-maven2-plugin</artifactId>
    <version>1.4.12</version>
     <configuration>
       <wait>true</wait>          <!-- waiting for Ctrl-C -->
       <container>
          <containerId>jetty8x</containerId>
          <type>embedded</type>
           <dependencies>
            <dependency>
                  <groupId>javax.el</groupId>
                    <artifactId>el-api</artifactId>          < !--reference sans version -->
              </dependency>
              <dependency>
                    <groupId>org.glassfish.web</groupId>
                      <artifactId>el-impl</artifactId>
             </dependency>
             <dependency>
                    <groupId>javax.servlet.jsp</groupId>
                    <!-- <artifactId>jsp-api</artifactId> ancienne version -->
                    <artifactId>javax.servlet.jsp-api</artifactId>
              </dependency>
               <dependency>
                     <groupId>javax.servlet</groupId>
                     <artifactId>jstl</artifactId>
                 </dependency>
          </dependencies>
       </container>
    </configuration>
  </plugin>
 </plugins>
</build>
```

1.3. Configuration pour Jboss 5.1 (en mode local/installed)

```
<!-- ..... dans le pom.xml du sous projet "....-ear" ..... -->
<build>
   <plugins><plugin>
```

```
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-ear-plugin</artifactId>
        <version>2.3.1</version>
        <configuration>
                <generateApplicationXml>true</generateApplicationXml>
                <includeJar>false</includeJar>
                <defaultLibBundleDir>lib</defaultLibBundleDir>
                <modules>
                        <webModule>
                                <groupId>com.mycompany.jee5app1</groupId>
                                <artifactId>my-jee5app1-web</artifactId>
                                <contextRoot>my-jee5app1-web</contextRoot>
                        </webModule>
                        <ejbModule>
                                <groupId>com.mycompany.jee5app1</groupId>
                                <artifactId>my-jee5app1-ejb</artifactId>
                        </ejbModule>
                </modules>
        </configuration>
</plugin>
<plugin>
    <groupId>org.codehaus.cargo</groupId>
    <artifactId>cargo-maven2-plugin</artifactId>
    <version>1.4.12</version>
     <configuration>
       <wait>true</wait>          <!-- waiting for Ctrl-C -->
       <container>
          <containerId>jboss51x</containerId>
           <type>installed</type>
           <home>C:\Prog\java\ServApp\jboss-5.1.0.GA</home>
       </container>
       <configuration>
             <type>existing</type>
             <home>C:\Prog\java\ServApp\jboss-5.1.0.GA\server\default</home>
       </configuration>
    </configuration>
  </plugin>
 </plugins>
 <!-- finalName : "my-jee5app1" (.ear) not "my-jee5app1-ear" (.ear) -->
 <finalName>my-jee5app1</finalName>
</build>
```

Autres possibilités/configurations

--> voir le site de référence du produit "cargo"

XV - Annexe – Configuration JEE (pom.xml)

1. Configuration "maven" pour applications "JEE"

Cette annexe présente quelques configurations "types" pour application JEE.

1.1. Application "web" avec services "Spring" pour tomcat7/8

Organisation globale de l'application (niveau "parent"):

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>com.mycompany.webapp1</groupId>
  <artifactId>my-webapp1</artifactId>
  <packaging>pom</packaging>
  <version>1.0-SNAPSHOT</version>

 <modules>
    <module>my-webapp1-jar</module>
    <module>my-webapp1-web</module>
 </modules>

<build>
        <plugins>
                <!-- configuration (enventuellement heritee) pour compilation en java 7 -->
                    <plugin>
                            <groupId>org.apache.maven.plugins</groupId>
                            <artifactId>maven-compiler-plugin</artifactId>
                            <!-- default maven-compiler-plugin version -->
                            <configuration>
                                    <source>1.7</source>
                                    <target>1.7</target>
                            </configuration>
                    </plugin>
            </plugins>
        </build>
</project>
```

Dépendances du sous module de services "my-webapp1-jar":

```
<?xml version="1.0"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd" xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <modelVersion>4.0.0</modelVersion>
 <parent>
  <artifactId>my-webapp1</artifactId>    <groupId>com.mycompany.webapp1</groupId>
 </parent>
```

```
<groupId>com.mycompany.webapp1</groupId>
<artifactId>my-webapp1-jar</artifactId>  <version>1.0-SNAPSHOT</version>
<name>my-webapp1-jar</name>

        <properties>
          <org.springframework.version>4.1.1.RELEASE</org.springframework.version>
         <org.hibernate.version>4.3.6.Final</org.hibernate.version>
        </properties>

<dependencies>

         <dependency><groupId>org.dbunit</groupId>    <artifactId>dbunit</artifactId>
        <version>2.5.0</version>     </dependency>

       <dependency>  <groupId>junit</groupId><artifactId>junit</artifactId>
                     <version>4.11</version>      <scope>test</scope>  </dependency>

        <dependency> <groupId>org.mockito</groupId>   <artifactId>mockito-core</artifactId>
                     <version>1.10.19</version>  </dependency>

       <dependency><groupId>log4j</groupId> <artifactId>log4j</artifactId>
              <version>1.2.17</version><scope>runtime</scope></dependency>

        <dependency> <groupId>org.slf4j</groupId> <artifactId>slf4j-api</artifactId>
         <version>1.7.7</version><scope>compile</scope>          </dependency>

        <dependency>  <groupId>org.slf4j</groupId><artifactId>slf4j-log4j12</artifactId>
        <version>1.7.7</version>     <scope>runtime</scope> </dependency>

        <dependency><groupId>net.sf.dozer</groupId> <artifactId>dozer</artifactId>
               <version>5.5.1</version> <scope>compile</scope></dependency>

         <dependency><groupId>mysql</groupId>
                     <artifactId>mysql-connector-java</artifactId>
                     <version>5.1.30</version>    <scope>runtime</scope>      </dependency>

        <dependency> <groupId>org.hsqldb</groupId> <artifactId>hsqldb</artifactId>
                      <version>2.3.2</version></dependency>

       <dependency><groupId>org.hibernate</groupId>
                     <artifactId>hibernate-entitymanager</artifactId>
                     <!-- with indirect/transitive <artifactId>hibernate-core</artifactId> -->
                     <version>${org.hibernate.version}</version>
              </dependency>

              <!--
              <dependency>
                     <groupId>javax.transaction</groupId>
                     <artifactId>jta</artifactId>
                     <version>1.1</version>
              </dependency>
              -->
```

```
        <dependency> <groupId>javax.validation</groupId><artifactId>validation-
api</artifactId>
        <version>1.1.0.Final</version><scope>compile</scope>    </dependency>

        <dependency><groupId>org.hibernate</groupId>
                <artifactId>hibernate-validator</artifactId><version>5.1.2.Final</version>
                 <scope>runtime</scope>        </dependency>

    <!--  <artifactId>spring-core</artifactId>        et <artifactId>spring-beans</artifactId>
      et <artifactId>spring-aop</artifactId>        sont indirectement lies a spring-context -->

        <dependency><groupId>org.springframework</groupId>
                        <artifactId>spring-context</artifactId>
                        <version>${org.springframework.version}</version>
                        <scope>compile</scope> </dependency>
    <dependency>   <groupId>javax.inject</groupId>    <artifactId>javax.inject</artifactId>
         <version>1</version>         </dependency>

        <dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjrt</artifactId><!--  pour annotations @Before , @Around , .... -->
        <version>1.8.2</version>    <scope>compile</scope>  </dependency>

      <dependency>
        <groupId>org.aspectj</groupId> <artifactId>aspectjweaver</artifactId>
        <version>1.8.2</version>  <scope>runtime</scope>
     </dependency>


        <!-- <artifactId>spring-tx</artifactId> et
                    <artifactId>spring-jdbc</artifactId> sont indirectement lies a spring-orm -->
        <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-orm</artifactId>
                        <version>${org.springframework.version}</version>
                        <scope>compile</scope>                </dependency>
        <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-test</artifactId>
                        <version>${org.springframework.version}</version>
                        <scope>test</scope>
                </dependency>
        </dependencies>
        <build>
   <finalName>my-webapp1-jar</finalName>
  </build>
</project>
```

Dépendances du sous module web "my-webpp1-web":

```
<?xml version="1.0"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd" xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <modelVersion>4.0.0</modelVersion>
 <parent>
   <artifactId>my-webapp1</artifactId>    <groupId>com.mycompany.webapp1</groupId>
 </parent>
 <groupId>com.mycompany.webapp1</groupId>
 <artifactId>my-webapp1-web</artifactId>  <version>1.0-SNAPSHOT</version>
 <packaging>war</packaging>
 <name>my-webapp1-web Maven Webapp</name>
 <url>http://maven.apache.org</url>

<repositories>
          <!-- specific repository needed for richfaces 4-->
               <repository>
                    <id>jboss.org</id>
                    <url>http://repository.jboss.org/nexus/content/groups/public-jboss/</url>
               </repository>
       </repositories>

       <properties>
          <org.springframework.version>4.1.1.RELEASE</org.springframework.version>
          <org.apache.cxf.version>3.0.2</org.apache.cxf.version>
          <org.apache.myfaces.version>2.2.5</org.apache.myfaces.version>
       </properties>

       <dependencies>
         <dependency>
               <groupId>com.mycompany.webapp1</groupId>
               <artifactId>my-webapp1-jar</artifactId>
                 <version>1.0-SNAPSHOT</version><!-- <scope>compile</scope> -->
         </dependency>

                <dependency>
                     <groupId>javax.servlet</groupId>
                     <artifactId>javax.servlet-api</artifactId>
                     <!-- servlet-api 2.5 for tc6 et javax.servlet-api 3.0.1 for tc7 -->
                     <version>3.0.1</version>
                     <scope>provided</scope>
               </dependency>

               <dependency>
                     <groupId>javax.servlet.jsp</groupId>
                     <artifactId>javax.servlet.jsp-api</artifactId>
                     <version>2.2.1</version>
                     <!-- jsp-api 2.1 for tc6 et servlet.jsp-api 2.2.1 for tc7 -->
                     <scope>provided</scope>
               </dependency>
       <dependency>
               <groupId>log4j</groupId>   <artifactId>log4j</artifactId>
```

```
        <version>1.2.17</version> <scope>compile</scope>
    </dependency>

    <dependency>
        <groupId>org.slf4j</groupId>        <artifactId>slf4j-api</artifactId>
        <version>1.7.7</version>            <scope>compile</scope>
    </dependency>

    <dependency>
        <groupId>org.slf4j</groupId>        <artifactId>slf4j-log4j12</artifactId>
        <version>1.7.7</version>            <scope>compile</scope>
    </dependency>

    <dependency>
     <groupId>javax.validation</groupId>    <artifactId>validation-api</artifactId>
     <version>1.1.0.Final</version>         <scope>compile</scope>
    </dependency>

    <dependency>
        <groupId>org.hibernate</groupId><artifactId>hibernate-validator</artifactId>
        <version>5.1.2.Final</version>  <!--  3.0.0.ga , 4.0.2.GA ,4.1.0-Final? -->
      </dependency>

     <!--  <artifactId>spring-core</artifactId>      et <artifactId>spring-beans</artifactId>
    et <artifactId>spring-aop</artifactId>      sont indirectement lies a spring-context -->

    <dependency>
       <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>
       <version>${org.springframework.version}</version> <scope>compile</scope>
    </dependency>

    <dependency>
        <groupId>javax.inject</groupId>  <artifactId>javax.inject</artifactId>
        <version>1</version>        <scope>compile</scope>
    </dependency>

    <dependency>
       <groupId>org.springframework</groupId> <artifactId>spring-web</artifactId>
       <version>${org.springframework.version}</version> <scope>compile</scope>
    </dependency>

    <dependency>
        <groupId>javax.servlet</groupId>  <artifactId>jstl</artifactId>
        <version>1.2</version> <!-- old: 1.1.2 not for jsf2 --> <scope>compile</scope>
    </dependency>

    <dependency>
       <groupId>org.apache.myfaces.core</groupId> <artifactId>myfaces-api</artifactId>
       <version>${org.apache.myfaces.version}</version> <scope>compile</scope>
    </dependency>
                    <!-- MyFaces 2 =  JSF 2 implementation -->
    <dependency>
      <groupId>org.apache.myfaces.core</groupId> <artifactId>myfaces-impl</artifactId>
```

```
            <version>${org.apache.myfaces.version}</version> <scope>runtime</scope>
        </dependency>

        <!--
         <dependency>
                <groupId>org.primefaces</groupId>
                <artifactId>primefaces</artifactId>
                <version>5.1</version>
        </dependency>
         -->
                                <!-- CXF for WebServices -->

        <dependency>
            <groupId>org.apache.cxf</groupId><artifactId>cxf-rt-frontend-jaxws</artifactId>
            <version>${org.apache.cxf.version}</version>
        </dependency>

        <dependency>
           <groupId>org.apache.cxf</groupId> <artifactId>cxf-rt-transports-http</artifactId>
           <version>${org.apache.cxf.version}</version>
        </dependency>
     </dependencies>
    <build>
   <finalName>my-webapp1-web</finalName>
  </build>
</project>
```

1.2. Application "JEE5" avec "EJB3" pour Jboss 5.1

Organisation globale de l'application JEE (module "parent")

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>com.mycompany.jee5app1</groupId>
   <artifactId>my-jee5app1</artifactId>
   <packaging>pom</packaging>
   <version>1.0-SNAPSHOT</version>
 <modules>
        <module>my-jee5app1-ejb</module>
        <module>my-jee5app1-web</module>
        <module>my-jee5app1-ear</module>
  </modules>
</project>
```

sous module "ear" pour packaging et déploiement vers le serveur "Jboss"

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
v4_0_0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <parent>
     <artifactId>my-jee5app1</artifactId>
     <groupId>com.mycompany.jee5app1</groupId>    <version>1.0-SNAPSHOT</version>
  </parent>
 <groupId>com.mycompany.jee5app1</groupId>
 <artifactId>my-jee5app1-ear</artifactId>
 <packaging>ear</packaging>  <version>1.0-SNAPSHOT</version>
 <name>my-jee5app1-ear Maven JEE5 Assembly</name>

 <dependencies>
 <dependency>
    <groupId>com.mycompany.jee5app1</groupId> <artifactId>my-jee5app1-ejb</artifactId>
    <version>1.0-SNAPSHOT</version>      <type>ejb</type>
  </dependency>

  <dependency>
    <groupId>com.mycompany.jee5app1</groupId> <artifactId>my-jee5app1-web</artifactId>
    <version>1.0-SNAPSHOT</version>      <type>war</type>
  </dependency>
 </dependencies>

 <build>
  <plugins>
   <plugin>
       <groupId>org.apache.maven.plugins</groupId>
       <artifactId>maven-ear-plugin</artifactId>
       <version>2.3.1</version>
       <configuration>
              <generateApplicationXml>true</generateApplicationXml>
              <includeJar>false</includeJar>
              <defaultLibBundleDir>lib</defaultLibBundleDir>
              <modules>
                     <webModule>
                            <groupId>com.mycompany.jee5app1</groupId>
                            <artifactId>my-jee5app1-web</artifactId>
                            <contextRoot>my-jee5app1-web</contextRoot>
                     </webModule>
                     <ejbModule>
                            <groupId>com.mycompany.jee5app1</groupId>
                            <artifactId>my-jee5app1-ejb</artifactId>
                     </ejbModule>
              </modules>
       </configuration>
   </plugin>

   <plugin>
     <groupId>org.codehaus.cargo</groupId>
     <artifactId>cargo-maven2-plugin</artifactId> <version>1.1.0</version>
      <configuration>
```

```
          <wait>true</wait>  <!-- waiting for Ctrl-C -->
          <container>
              <containerId>jboss51x</containerId>
               <type>installed</type>
               <home>C:\Prog\java\ServApp\jboss-5.1.0.GA</home>
          </container>
          <configuration>
                  <type>existing</type>
                  <home>C:\Prog\java\ServApp\jboss-5.1.0.GA\server\default</home>
          </configuration>
      </configuration>
     </plugin>

   </plugins>
   <!--  finalName : "my-jee5app1" (.ear) not "my-jee5app1-ear" (.ear) -->
   <finalName>my-jee5app1</finalName>
  </build>

</project>
```

sous module "ejb" (avec plein de dépendances en mode "provided")

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>  <modelVersion>4.0.0</modelVersion>
  <parent>
       <artifactId>my-jee5app1</artifactId>  <groupId>com.mycompany.jee5app1</groupId>
    </parent>
  <groupId>com.mycompany.jee5app1</groupId>
  <artifactId>my-jee5app1-ejb</artifactId>
  <packaging>ejb</packaging>  <version>1.0-SNAPSHOT</version>
  <name>my-jee5app1-ejb Maven JEE5 EJB</name>
…
```

sous module "web" (avec dépendances adéquates):

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
 <modelVersion>4.0.0</modelVersion>
 <parent>
     <artifactId>my-jee5app1</artifactId>   <groupId>com.mycompany.jee5app1</groupId>
  </parent>
 <groupId>com.mycompany.jee5app1</groupId>
 <artifactId>my-jee5app1-web</artifactId>
 <packaging>war</packaging>  <version>1.0-SNAPSHOT</version>
 <name>my-jee5app1-web Maven JEE5 Webapp</name>

  <repositories>
                <!-- specific repository needed for jee5 api -->
                <repository>
                        <id>java.net2</id>
                        <name>hosts the javaee-api dependency</name>
                        <url>http://download.java.net/maven/2</url>
                 </repository>
   </repositories>

 <dependencies>
  <dependency>
   <groupId>com.mycompany.jee5app1</groupId> <artifactId>my-jee5app1-ejb</artifactId>
   <version>1.0-SNAPSHOT</version>                <scope>provided</scope>
  </dependency>

  <dependency>
   <groupId>javaee</groupId> <artifactId>javaee-api</artifactId>
   <version>5</version>    <scope>provided</scope>
  </dependency>
….
```

XVI - Annexe – "B.O.M."

1. Gestion avancée des dépendances (BOM)

1.1. DependencyManagement

Une expression simple des dépendances s'effectue via une liste de <dependency> au sein de la partie
<project><dependencies> ... </dependencies></project> .

Il est possible d'exprimer certaines dépendances en deux phases:

- détailler certaines dépendances (à réutiliser) dans la partie <dependencyManagement>
- faire référence (sans détails) à ces dépendances dans la partie project/dependencies .

Détails apparaissant généralement dans la partie "**dependencyManagement**":

- version précise
- portée (scope)
- exclusions
- (et groupId/artifactId) , ...

Elements à préciser au sein d'une référence à une dépendance :

- groupId , artifactId
- ~~version~~
- packaging (si différent de "jar")

NB:

- La partie "**dependencyManagement**" n'est réellement intéressante que si elle peut être partagée (ou factorisée) .
- La réutilisation d'une partie "dependencyManagement" passe par l'une des deux solutions suivantes:
  * héritage de configuration depuis un projet "parent".
  * import de dépendances depuis un projet "BOM"

1.2. import de gestion de dépendances et "BOM"

"**BOM**" signifie "***B**ill **O**f **M**aterials*" (soit à peu près en français "*note de configuration technique*"). Il s'agit d'un projet spécial (ayant le **packaging** fixé à "**pom**" et hébergeant au moins un bloc "**dependencyManagement**" prévu pour être ultérieurement importé .

Exemple1 (concret):

Utilisation de richeFaces4 (de jboss) avec versions paramétrées via un "bom":

```
<repositories>
        <!-- specific repository needed for richfaces  4 (different for 3.3) -->
            <repository>
                    <id>jboss.org</id>
                    <url>http://repository.jboss.org/nexus/content/groups/public-jboss/</url>
            </repository>
</repositories>

    <properties>
        <org.richfaces.bom.version>4.0.0.Final</org.richfaces.bom.version>
    </properties>

    <dependencyManagement>
        <dependencies>
            <dependency>
                    <groupId>org.richfaces</groupId>
                    <artifactId>richfaces-bom</artifactId>
                    <version>${org.richfaces.bom.version}</version>
                    <scope>import</scope>
                    <type>pom</type>
            </dependency>
        </dependencies>
    </dependencyManagement>

<dependencies>
        ...
  <dependency>
      <groupId>org.richfaces.ui</groupId>
      <artifactId>richfaces-components-ui</artifactId>
       <!-- pas de version à choisir ici , déjà précisée dans le BOM -->
   </dependency>
   <dependency>
      <groupId>org.richfaces.core</groupId>
      <artifactId>richfaces-core-impl</artifactId>
       <!-- pas de version à choisir ici , déjà précisée dans le BOM -->
   </dependency>
<dependencies>
...
```

Exemple2 ("BOM" maison pour slf4j-log4j):

```
<?xml version="1.0"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd" xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <modelVersion>4.0.0</modelVersion>
 <artifactId>slf4j-log4j-bom</artifactId>
 <groupId>com.mycompany.bom.util</groupId>  <version>1.0-SNAPSHOT</version>
 <name>slf4j-log4j-bom</name>
 <packaging>pom</packaging>

 <dependencyManagement>
 <dependencies>
        <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.17</version>
        <scope>runtime</scope>
        </dependency>

        <dependency>
                 <groupId>org.slf4j</groupId>
                <artifactId>slf4j-api</artifactId>
                <version>1.7.7</version>
                <scope>compile</scope>
        </dependency>

        <dependency>
                <groupId>org.slf4j</groupId>
                <artifactId>slf4j-log4j12</artifactId>
                <version>1.7.7</version>
                <scope>runtime</scope>
        </dependency>

  </dependencies>
 </dependencyManagement>
</project>
```

```
<!-- à réutiliser depuis un autre pom.xml via :
        <dependencyManagement>
                <dependencies>

                        <dependency>
           <artifactId>slf4j-log4j-bom</artifactId>
           <groupId>com.mycompany.bom.util</groupId>
           <version>1.0-SNAPSHOT</version>
           <scope>import</scope>
           <type>pom</type>
         </dependency>
      <dependency> ... </dependency>
       </dependencies>
 </dependencyManagement>
```

```
  ET AUSSI :
 <dependencies>
       <dependency>
                 <groupId>log4j</groupId>
                    <artifactId>log4j</artifactId>
              </dependency>
               <dependency>
                   <groupId>org.slf4j</groupId>
                     <artifactId>slf4j-log4j12</artifactId>
              </dependency>
              <dependency>
              <groupId>org.slf4j</groupId>
                    <artifactId>slf4j-log4j12</artifactId>
                 </dependency>
</dependencies>
-->
```

NB : cette notion de pack ne fonctionne bien que pour une portée (scope) = compile .
La transitivité est parfaitement bien gérée par maven en scope=compile .
En scope="runtime" , la transitivité n'est pas automatique et la notion de pack est inopérante .

XVII - Annexe – Hudson - Jenkins

1. Hudson/Jenkins (intégration continue)

"**Hudson / Jenkins**" est actuellement un **logiciel d'intégration continue très en vogue** car il est **très simple à configurer et à utiliser**.

1.1. Installation de "Hudson / Jenkins"

Recopier **jenkins.war** dans **TOMCAT_HOME/webapps** (avec un éventuel Tomcat pour intégration continue configuré sur le port 8585).

Etant donné que la configuration de jenkins ne nécessite pas de base de données relationnelle (mais de simples fichiers sur le disque dur) , il n'y a rien d'autre à configurer lors de l'installation .

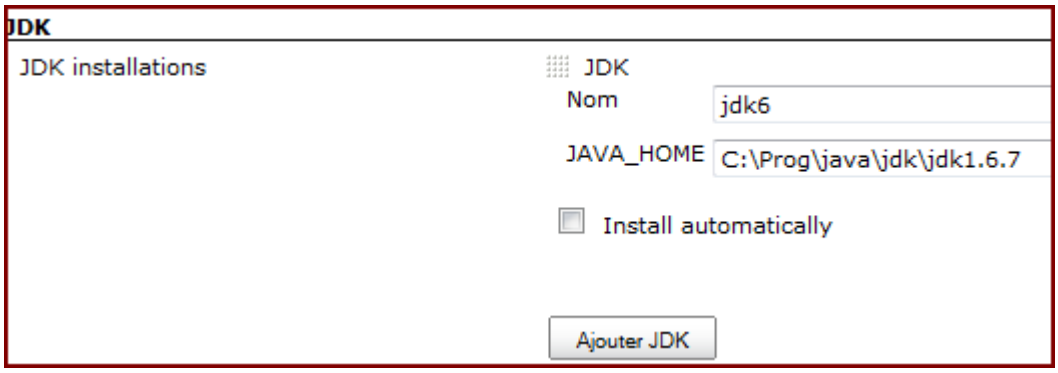
1.2. Configuration système de "Hudson / Jenkins"

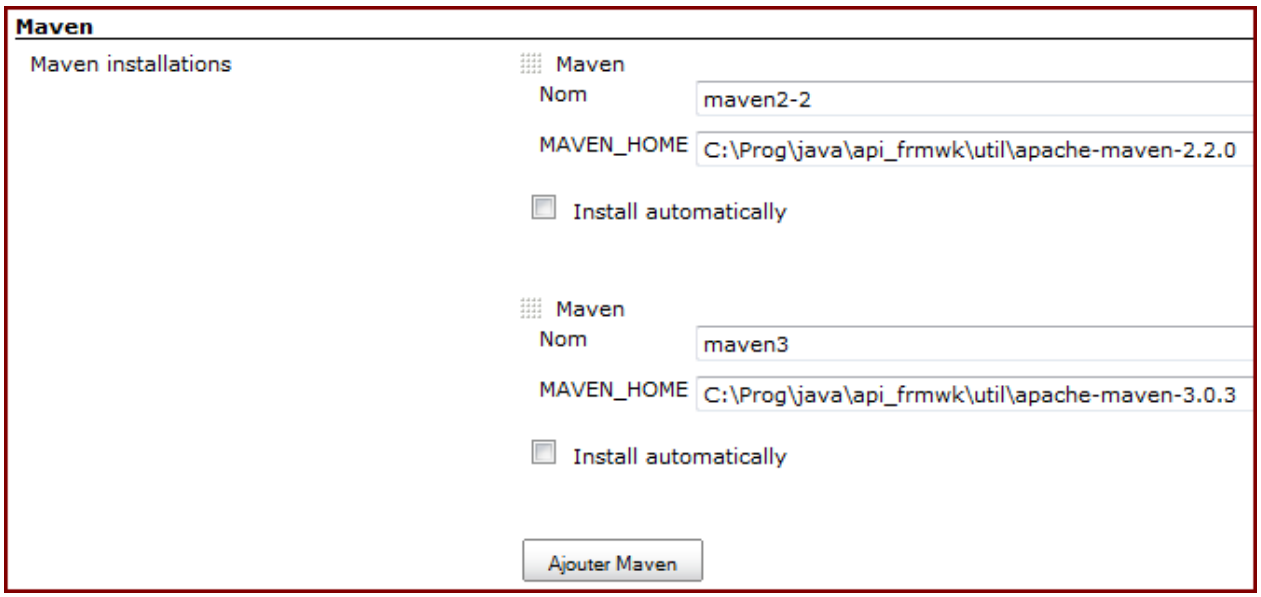
http://localhost:8585/jenkins

Premier menu à activer : ***Administrer Jenkins / Configurer le système*** .

Configurer dans un premier temps tous les chemins permettant d'accéder aux éléments suivants:

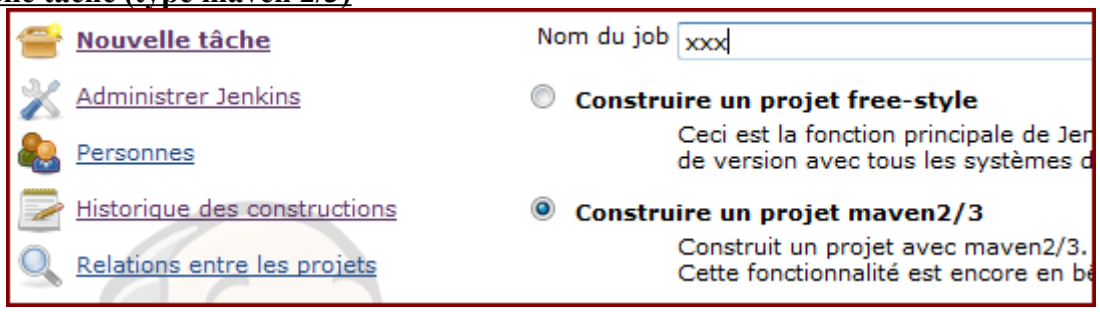
- **jdk**
- **maven**
- **version de svn** *(ex: 1.6)*



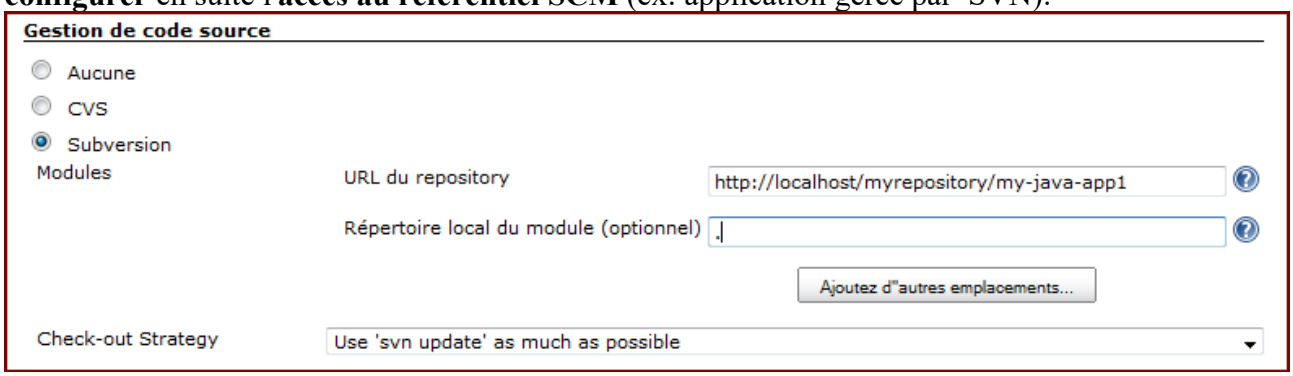


1.3. Configuration d'un Job (tâche) dans "Hudson / Jenkins"

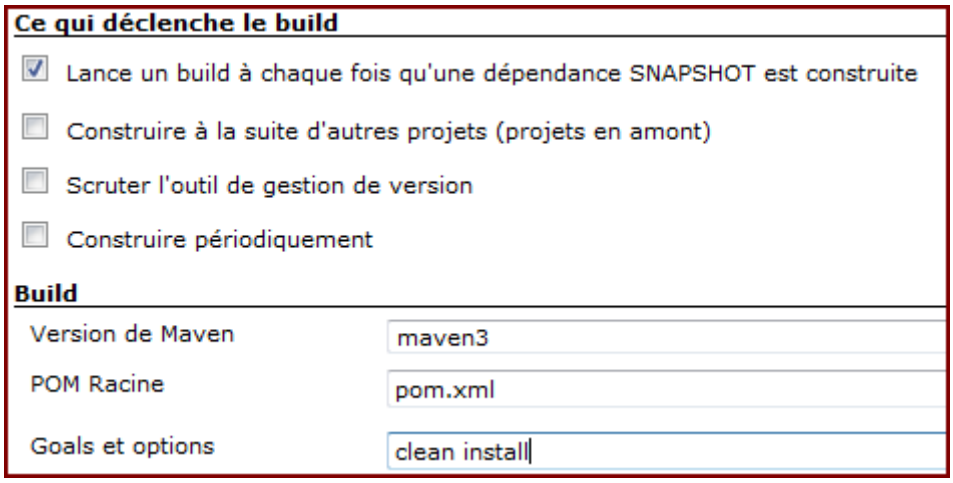
Nouvelle tâche (type maven 2/3)



configurer en suite l'**accès au référentiel SCM** (ex: application gérée par SVN):



Spécifier ensuite certains paramètres de "**build**" :



1.4. Lancement d'un "build" au sein de Hudson/Jenkins

2. Notification des développeurs concernés

2.1. Notifications élémentaires par flux RSS

Hudson / jenkins créer des "**flux RSS**" **attachés à chaque projet pris en charge**.
Un flux RSS "**tous les builds**" permet d'être averti du résultat de chaque nouveau build .
Un flux RSS "**tous les echecs**" permet de n'être averti qu'en cas d'échec lors d'un build.

En cliquant sur l'un des icônes "RSS" de l'interface graphique de Jenkins , on peut (via le menu

contextuel "copier l'adresse du lien") récupérer l'URL du flux (exemple : http://localhost:8585/jenkins/job/my-java-app1-in-svn-repo1/rssAll) pour ensuite paramétrer un lien dans un navigateur internet (tel que firefox) ou bien dans un logiciel de consultation des emails (tel que ThunderBird ou Oulook).

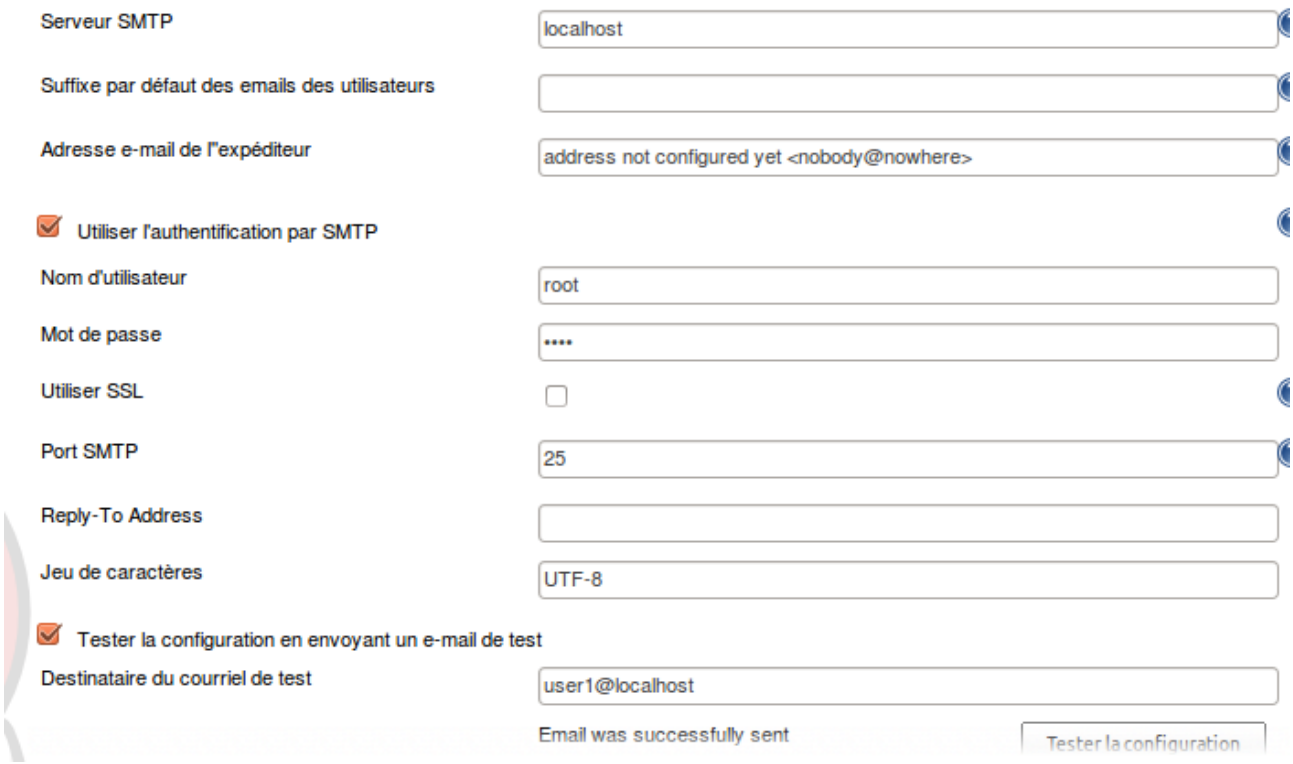
Pour accrocher un flux rss à *thunderbird*, le mode opératoire est le suivant :
Edition ou outils / paramètres des comptes / gestions des comptes / nouveau compte de type "blog & news" / gérer les abonnements puis saisir l'adresse du flux et ajouter .

NB : le contenu du flux RSS comporte simplement un message "build … réussi ou en échec" et un lien hypertexte qui renvoie sur la partie "web" de jenkins qui détaille les résultats .

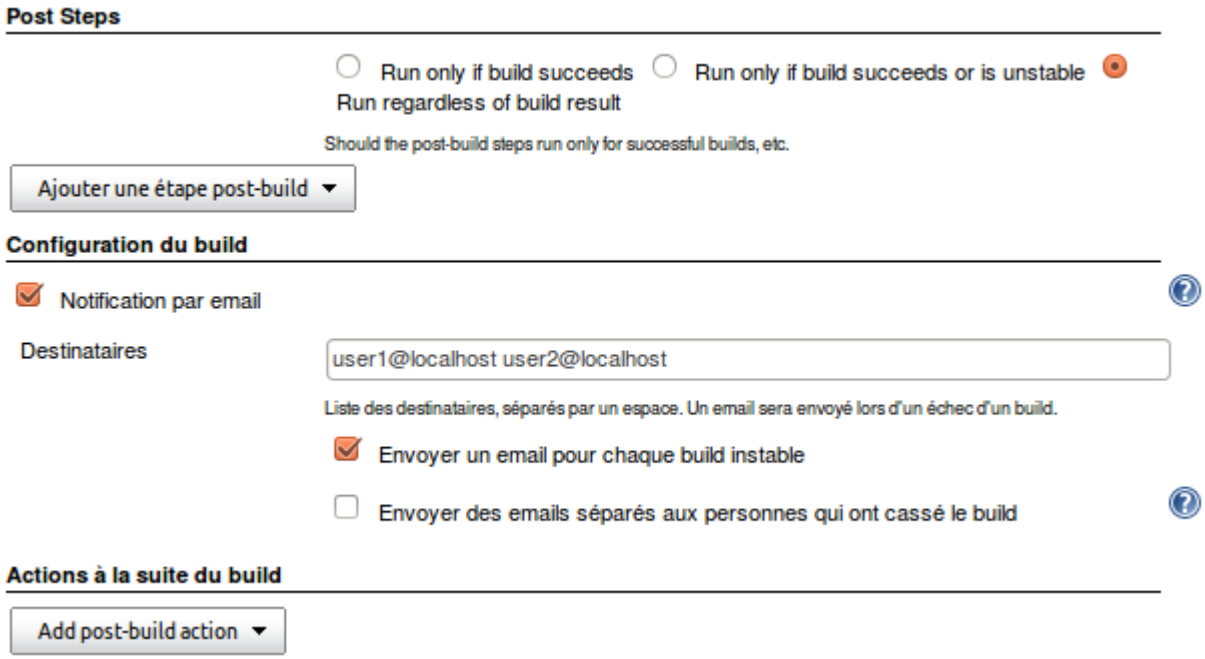
2.2. Notification par email

Pour activer la notification par emails , il faut commencer par paramétrer un lien vers un serveur SMTP depuis la partie "administration" de la console de Hudson/Jenkins.

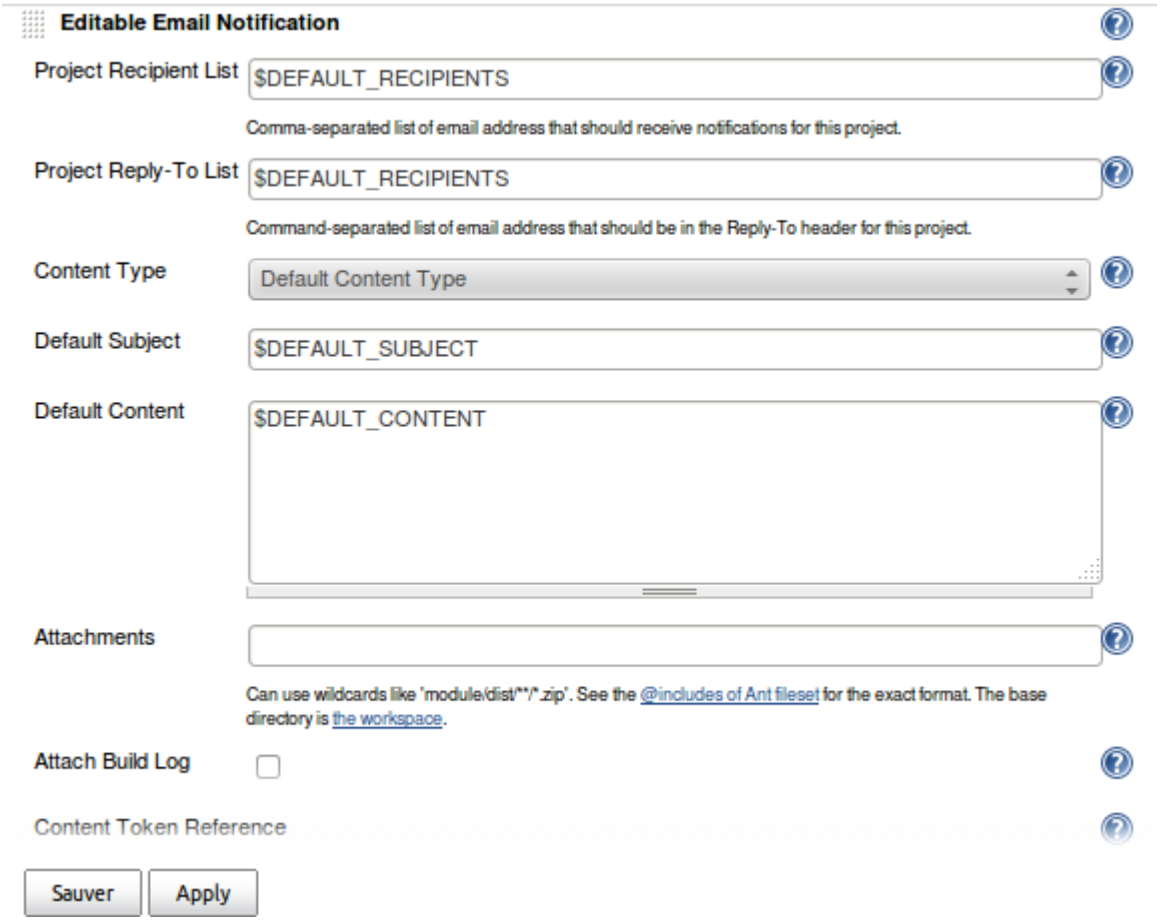
Exemple :



Ensuite, au niveau d'un "build" (application à construire), on peut configurer près des "post-action" une notification basique par email (avertir seulement en cas d' échec) en précisant une liste d'emails pour les développeurs destinataires :



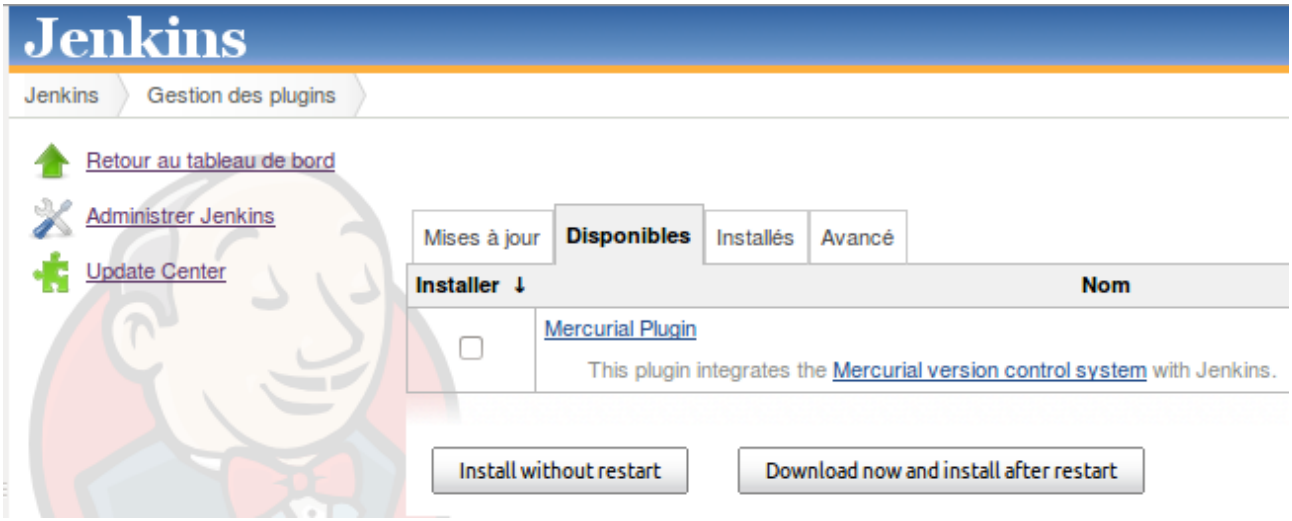
D'autre part, pour contrôler de façon plus fine les notifications envoyées, on peut éventuellement installer un plugin jenkins supplémentaire "**email-ext.hpi**" puis s'en servir via la post-action "***Editable Email Notification***":



3. Installation d'un plugin jenkins pour GIT / Mercurial

Par défaut Hudson/Jenkins est prévu pour se connecter à SVN (ou CVS) , si l'on souhaite s'interfacer avec un autre "scm" (tel que GIT ou Mercurial) , il faut alors installer un nouveau plugin.

L'installation d'un plugin peut quelquefois se faire directement depuis la partie "**administration / gestion des plugins**" lorsque le réseau externe (internet) est accessible.



L'installation d'un nouveau plugin pour Hudson/Jenkins peut se faire manuellement via :

1) **télécharger** un plugin "jenkins" (*ex*: **git.hpi**) depuis le site "http://updates.jenkins-ci.org/download/plugins/" *(+ site "https://wiki.jenkins-ci.org/display/JENKINS/Plugins" pour la documentation sur les plugins existants"*

2) recopier ce plugin (fichier d'extension ".hpi") dans le répertoire **plugins** de jenkins ***(ex : /opt/tomcat-7-port8585/webapps/jenkins/WEB-INF/plugins)***

3) activer le menu "**administration / gestion des plugins**" de Jenkins pour vérifier et peaufiner éventuellement l'installation du plugin .

4. Paramétrages fins des builds de hudson/jenkins

Pour paramétrer finement un build déclenché par hudson/jenkins , on pourra essentiellement s'appuyer sur les réglages suivants :

- préciser d'éventuelles "**pré-actions**" (ex : ré-initialisations de données , ….)
- **bien paramétrer** la ou les **commande(s) maven** en spécifiant des options de type **profils éventuellement complémentaires** (ex : package **-P**profile1 **-P**profile2)
- préciser d'éventuelles "**post-actions**" (ex : stocker l'artifact construit en cas de succès)

NB1 : on peut ajouter de nouveaux types de post-action en installant des plugins .
NB2 : il est possible de chaîner certains builds (lancement dans un ordre contrôlé).

XVIII - Annexe – Apache-Continuum

1. Continuum (intégration continue)

Continuum est un logiciel d'intégration continue (géré par Apache group) et qui s'appuie à fond sur maven.

1.1. Installation d'apache-continuum

Recopier **apache-continuum-1.3.7.war** dans **TOMCAT_HOME/webapps** (en renommant la copie "**continuum.war**")

ajuster TOMCAT_HOME/startup.bat (ou équivalent)
de façon à ajouter la variable d'environnement suivante:

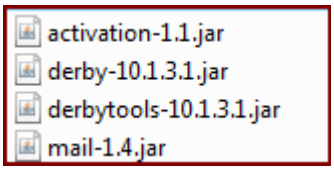
```
set CATALINA_OPTS="-Dappserver.base=C:\Prog\java\ServApp\Tomcat6_IntegrationContinue"
```

(valeur de appserver.base = généralement valeur de CATALINA_HOME)

D'autre part, il faut configurer une base de données et un accès mail pour continuum:

* ajouter le fichier "**continuum.xml**" dans **conf/Catalina/localhost**

* **ajouter les ".jar" nécessaires dans TOMCAT_HOME/lib (ou common/lib)**



Exemple de fichier "***continuum.xml***" (pour conf/Catalina/localhost):

```
<Context path="/continuum">

  <Resource name="jdbc/users"
        auth="Container"        type="javax.sql.DataSource"
        username="sa"           password=""
        driverClassName="org.apache.derby.jdbc.EmbeddedDriver"
        url="jdbc:derby:database/users;create=true" />

  <Resource name="jdbc/continuum"
        auth="Container"        type="javax.sql.DataSource"
        username="sa"           password=""
        driverClassName="org.apache.derby.jdbc.EmbeddedDriver"
        url="jdbc:derby:database/continuum;create=true" />

  <Resource name="mail/Session"         auth="Container"
        type="javax.mail.Session"       mail.smtp.host="localhost"/>
</Context>
```

NB: ajouter "C:\Prog\java\api_frmwk\util**apache-maven-3.0.3\bin**" ou équivalent
dans le **path** système (de windows ou ...)

1.2. Configuration d'apache-continuum

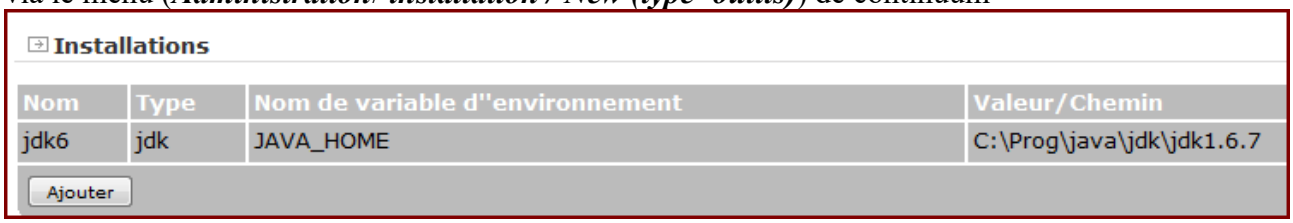
http://localhost:8085/continuum

Créer un compte "administrateur"
exemple:

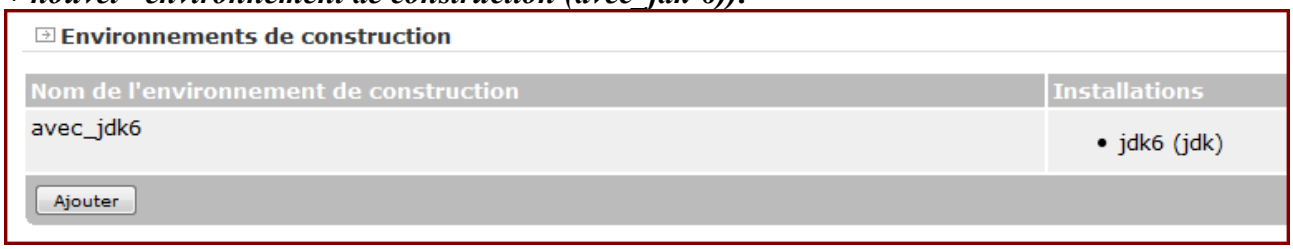
admin , admin@localhost , password: admin1

==========

Configurer le **jdk** (C:\Prog\java\jdk\jdk1.6.7)
via le menu (***Administration/ installation / New (type=outils)***) de continuum

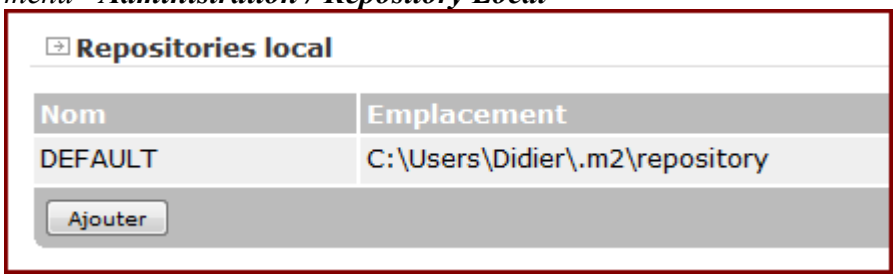


+ nouvel "environnement de construction (avec_jdk-6)):

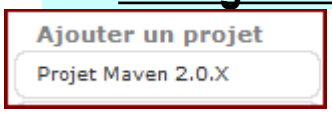


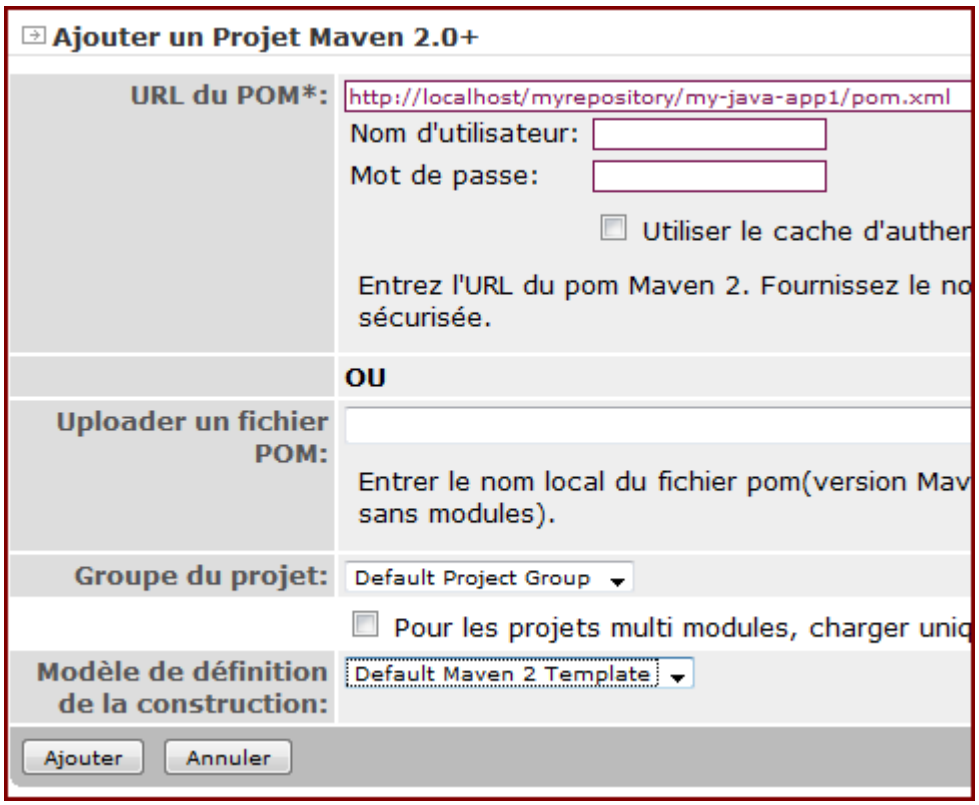
Vérifier (et si besoin ajuster) la configuration Maven:

menu ***"Administration / Repository Local"***



1.3. Configuration d'un projet "continuum"

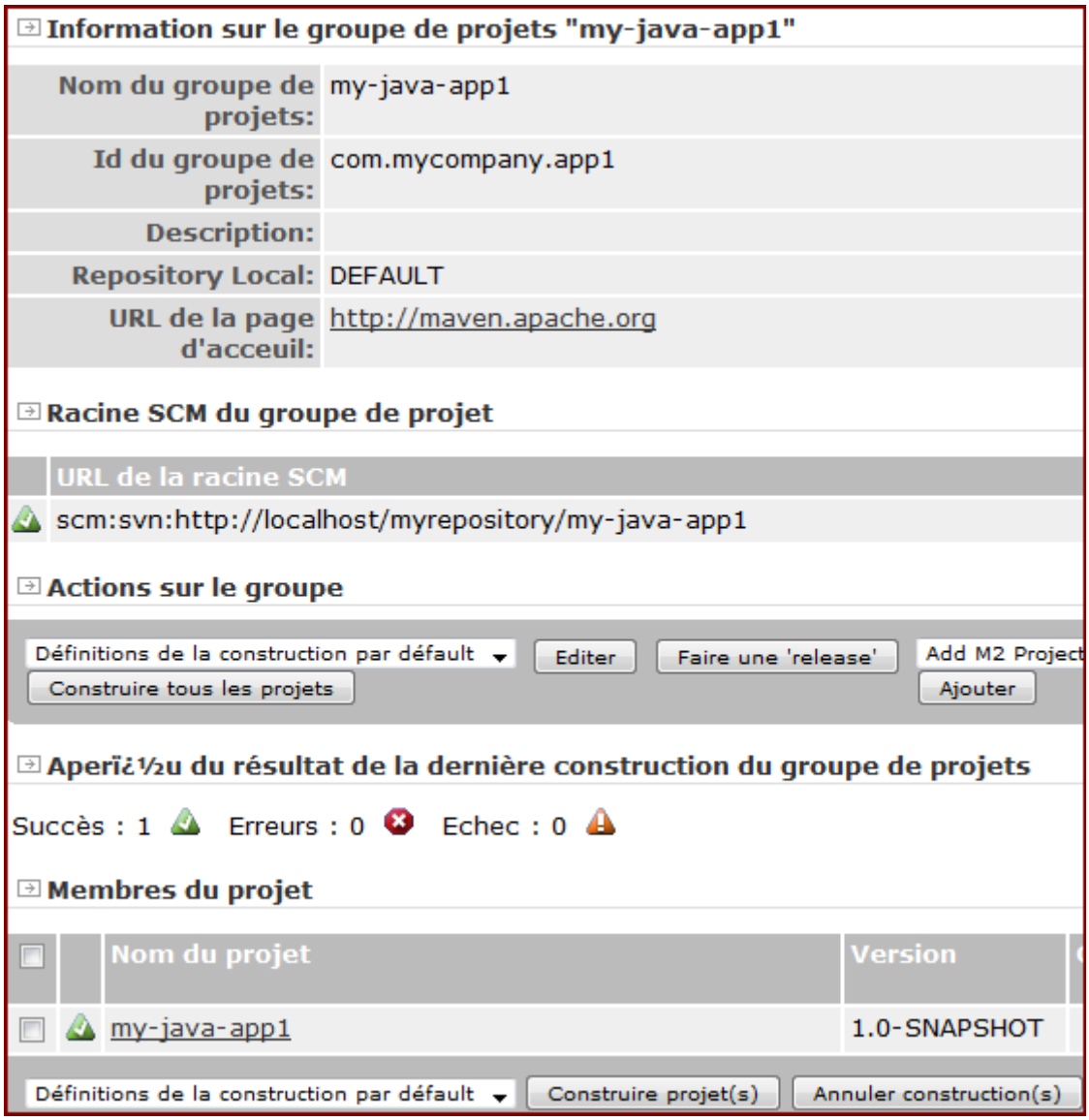




NB: pour préciser le référentiel SVN , il faut en fait préciser l'URL du fichier pom.xml (qui comporte lui même l'url svn dans sa partie <scm>)

pom.xml

```
<project ...>  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app1</groupId>
  <artifactId>my-java-app1</artifactId>  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>  <name>my-java-app1</name>
   <scm>
          <connection>scm:svn:http://localhost/myrepository/my-java-app1</connection>
          <url>http://localhost/myrepository/my-java-app1</url>
  </scm>
  <dependencies>    ...  </dependencies>
  <build> <plugins> <plugin>
        <groupId>org.apache.maven.plugins</groupId>
       <artifactId>maven-scm-plugin</artifactId>  <version>1.5</version>
       <configuration>
         <connectionType>connection</connectionType> <!-- or developerConnection -->
       </configuration>
     </plugin>    /plugins>  </build>
</project>
```


Information sur le groupe de projets "my-java-app1"
Nom du groupe de projets:
my-java-app1
Id du groupe de projets:
com.mycompany.app1
Description:
Repository Local:
DEFAULT
URL de la page d'acceuil:
http://maven.apache.org
Racine SCM du groupe de projet
URL de la racine SCM
scm:svn:http://localhost/myrepository/my-java-app1
Actions sur le groupe
Définitions de la construction par défault
Editer
Faire une 'release'
Add M2 Project
Construire tous les projets
Ajouter
Aperï¿½u du résultat de la dernière construction du groupe de projets
Succès : 1
Erreurs : 0
Echec : 0
Membres du projet
Nom du projet
Version
my-java-app1
1.0-SNAPSHOT
Définitions de la construction par défault
Construire projet(s)
Annuler construction(s)

1.4. Configuration de maven liée à "Continuum":

Certains serveurs d'intégration continue (tel que Hudson/Jenkins) configurent à leur niveau le lien avec le référentiel SVN (ou git ou CVS) et n'ont pas absolument besoin de configuration supplémentaire dans le fichier "pom.xml".

A l'inverse , le serveur d'intégration continue "***Continuum***"(de la communauté Apache) est prévu pour utiliser les parties suivantes du fichier "pom.xml" d'un projet pour indirectement connaître l'URL du référentiel de source (CVS , SVN ou GIT) et les adresses e-mail des personnes à avertir à l'issu d'un build :

```
[...]
<ciManagement>
        <system>continuum</system>
        <url>http://localhost:8080/continuum</url>
        <notifiers>
                <notifier>
                        <type>mail</type>
                        <configuration>
                                <address>youremail@yourdomain.com</address>
                        </configuration>
                </notifier>
        </notifiers>
</ciManagement>
[...]
```

pour paramétrer la liste des personnes à notifier (sur le résultat de l'intégration et des tests)

et

```
[...]
<scm>
   <connection>scm:svn:file://localhost/C:/.../svn/xxx/yyy</connection>
   <developerConnection>scm:svn:file://localhost/C:/.../svn/ xxx/yyy</developerConnection>
</scm>
[...]
<distributionManagement>
        <site>
                <id>website</id>
                <url>file://localhost/C:/.../sites/xxx/reference/${project.version}</url>
        </site>
</distributionManagement>
[...]
```

pour paramétrer les connexions au référentiel SVN et au site web où éventuellement déployer le "site" (documentation) .

XIX - Annexe – Bibliographie, Liens WEB + TP

1. Bibliographie et liens vers sites "internet"

http://maven.apache.org	Site web officiel sur maven
http://m2eclipse.sonatype.org/	Site pour plugin eclipse "m2e"
http://dcabasson.d***eveloppez.com***/articles/java/maven/introduction-maven2/	Introduction à maven2
Livre "Apache Maven" • Auteur(s) : Nicolas De Loof , Arnaud Héritier • Editeur : Pearson Education	Livre en français sur maven

2. TP

Liste de "Tp" progressifs sur maven (selon instruction du formateur) avec:

- scripts (.bat) préparés qui lancent les lignes de commandes "maven" (à étudier et lancer dans l'ordre).
- exemples de code java fournis durant la formation.