
Maven et intégration continue

Table des matières

I - Infrastructure pour intégration continue.....	4
1. Principes de l'intégration continue avec maven.....	4
2. Apports de maven.....	6
3. Configuration de Jenkins.....	11
4. exercices/TP.....	18
II - Référentiel maven d'entreprise (nexus oss).....	19
1. Serveur de librairies / Référentiel d'entreprise.....	19
2. Proxy vers référentiels externes.....	23
3. Configuration en lecture.....	25
4. Configuration en écriture (avec droits d'accès).....	27
5. Paramétrages selon le contexte.....	29
6. Exercices / TP autour de "nexus oss".....	30

III - Projet "maven" multi-modules , cargo , JEE.....	31
1. Maven et Jenkins (liaisons basiques).....	31
2. Maven et applications multi-modules.....	36
3. Configuration "maven" pour applications "JEE".....	37
4. Exemple de configuration du plugin "cargo" pour déploiement vers serveur Jee.....	47
5. Exercices/Tp.....	50
IV - Test d'intégration (via plugin failsafe).....	51
1. Phases pour les tests d'intégration.....	51
2. plugin "failsafe" et tests d'intégration.....	52
3. Tp (test d'intégration).....	53
V - Test d'intégration web via selenium.....	54
1. Selenium (presentation et installation).....	54
2. Enregistrement séquence "web" via selenium-IDE.....	55
3. Tests d'intégration avec selenium.....	57
4. Tp (selenium + test intégration / web).....	58
VI - Bon usage des profils "maven" (ic).....	59
1. Rappels sur profils "maven"	59
2. Profils pour tests rapides avec base h2 ou hsSql.....	63
3. Génération de livrables et de "release"	63
4. Profils pour tests sophistiqués (analyse , perf ,...).....	68
5. Tp (profils "maven").....	68
VII - Aspects divers et avancés ("maven" et "ic").....	69
1. Génération de rapports , de "javadoc" ,	69
2. Bonnes pratiques "maven".....	69
3. Gestion avancée des dépendances (BOM).....	69
VIII - Annexe – Archetype "maven".....	75
1. Notion d'archetype.....	75
2. Utilisation d'un (nouvel) archetype maven.....	75
3. Création d'un nouvel archetype maven.....	76
IX - Aspects divers et avancés de "maven".....	77
1. Ajout (et éventuel filtrage) de ressources "web" externes.....	77

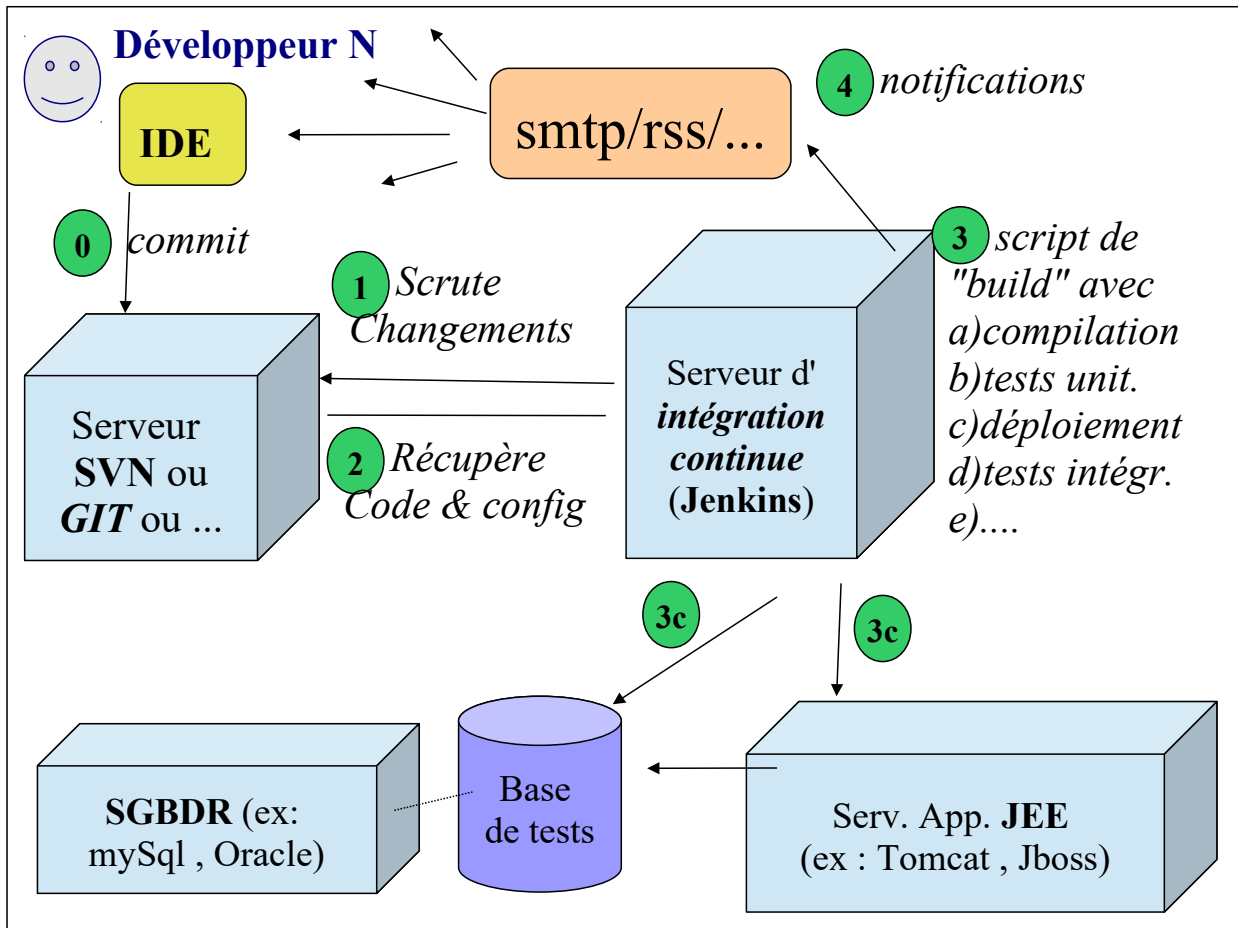
2. Activations de profils.....	78
3. Plugins pour maven.....	81
4. Programmation d'un plugin pour maven.....	82
5. Génération et publication d'une documentation.....	85
6. Javadoc (via maven).....	87
7. Rapports avec maven.....	88

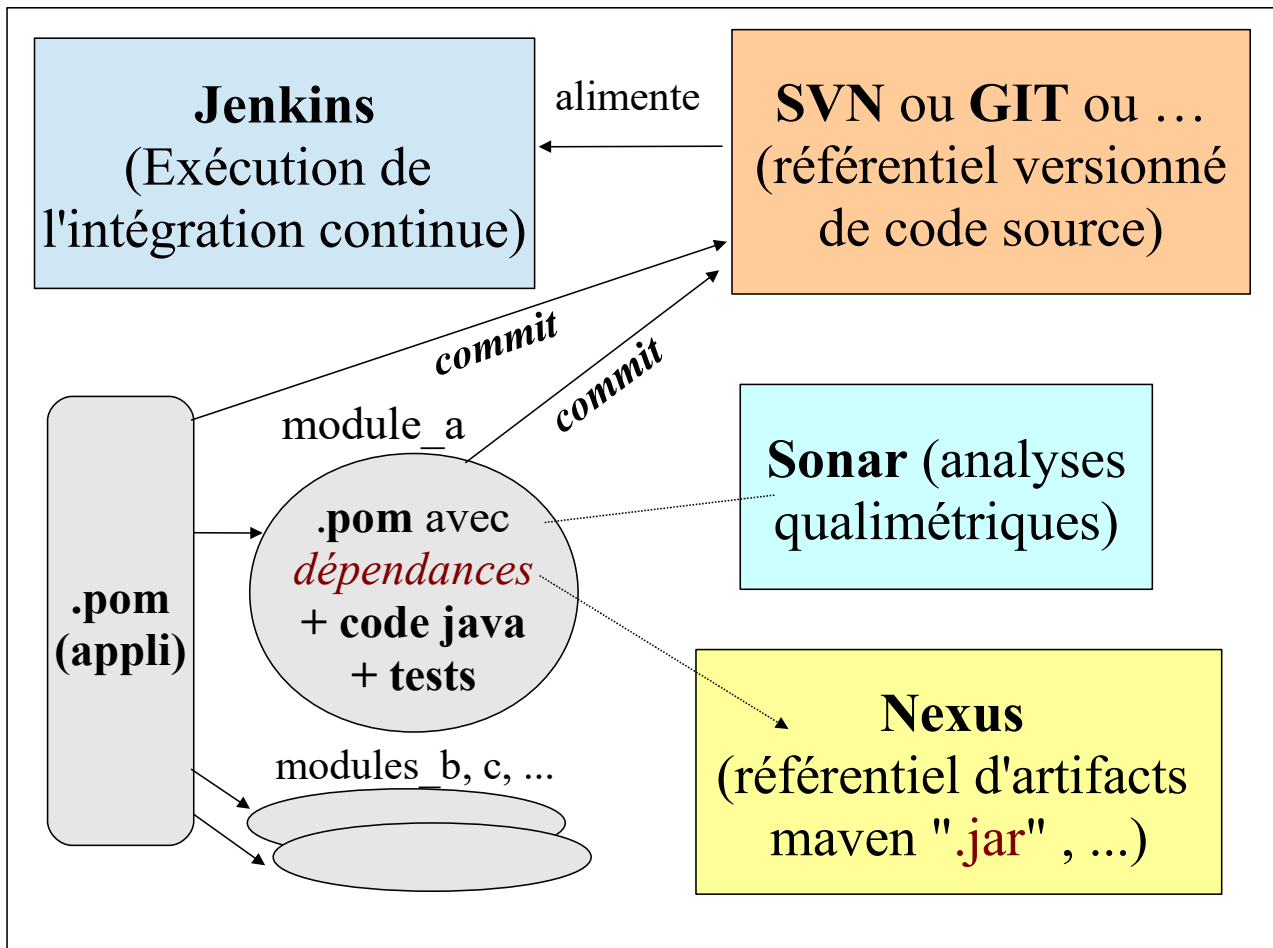
X - Annexe – m2e (maven2eclipse).....	89
--	-----------

1. Lien entre maven et eclipse (m2e).....	89
---	----

I - Infrastructure pour intégration continue

1. Principes de l'intégration continue avec maven





Rappels des rôles des éléments de la chaîne d'intégration continue

Nexus (ou archiva ou ...) est un gestionnaire de référentiel maven qui permet essentiellement de stocker et récupérer sur demande (de façon versionnée) les librairies (".jar") utilisées par les modules du projet. (ex : hibernate-core-3.5.jar , ...) .

Ceci permet d'éviter de stocker inutilement des tonnes de ".jar" dans le référentiel de code source (svn ou git ou ...) car grâce aux dépendances exprimées dans les fichiers "pom.xml" des modules du projet, la technologie maven sera capable de récupérer automatiquement les dépendances (librairies, ...) au sein de nexus pour reconstruire WEB-INF/lib/liste_des_jars ou un équivalent .

Sonar (contrôlé/piloté par maven ou ant) permet d'effectuer quasi automatiquement des analyses qualimétriques sur le code du logiciel (ex : couverture des tests , respects de certaines règles au sein de la structure orientée objet et modulaire du code , style et convention , ...) . les rapports effectués par Sonar sont facilement accessibles au bout d'une URL.

Le gestionnaire/référentiel de code source (**GIT** ou **SVN** ou) sert essentiellement à **stocker tout le code source des modules d'un projet** .

Le gestionnaire d'intégration continue (**Jenkins** ou ...) pourra ensuite récupérer régulièrement dans GIT ou SVN le code et la configuration maven des modules d'une application pour :

- *reconstruire (recompiler) les modules
- *relancer tous les tests (unitaires , intégration selon profil, ...)
- *notifier les développeurs des résultats des tests et des constructions

2. Apports 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**" (Project Object Model).

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

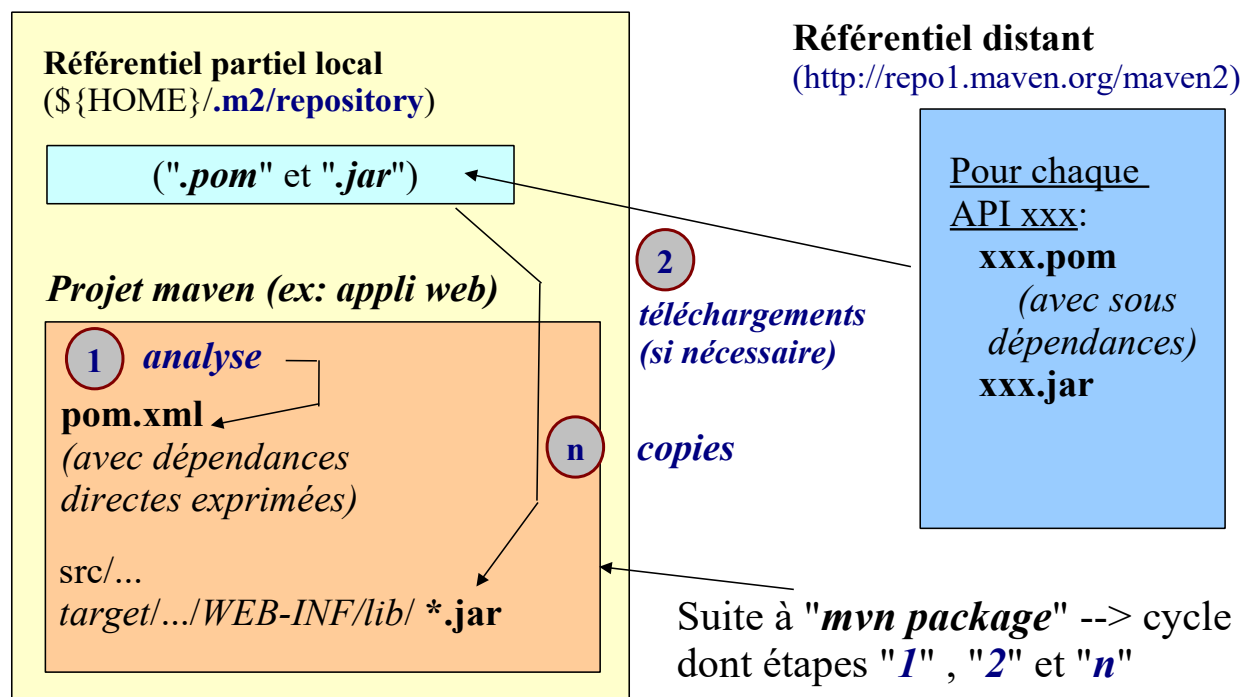
Gestion distribuée (sur le web) des **dépendances** inter-projets.

---> identification et **téléchargement automatique des ".jar"** nécessaires selon les API déclarées en dépendances.

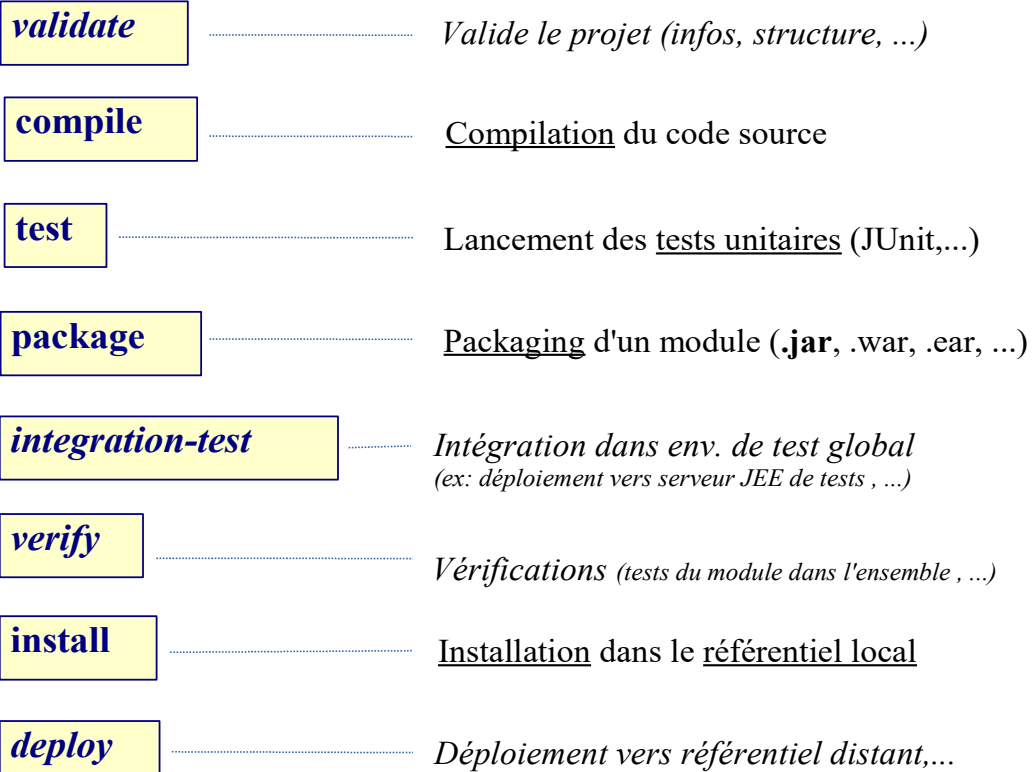
Gère toutes les phases (compile/build, tests unitaires, packaging, stockage dans le référentiel, éventuel lien avec SVN, ...).

Téléchargement automatique des librairies nécessaires

(avec prise en compte des dépendances indirectes)



cycle de vie d'un "build" maven



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 .

Cycle , phases et buts (goals)

mvn install

Ligne de commande avec but (goal) demandé

validate (1)

generate-sources (2)

process-sources (3)

generate-resources (4)

process-resources (5)

compile (6)

...

test (13)

...

package (15)

...

install (20)

deploy (21)

xy? (goal / plugin)

But/ plugin à définir et à associer à une phase

compile (goal / plugin)

Cycle à 21 phases

Buts / plugins prédéfinis

install (goal / plugin)

pas déclenché (après install)

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

À déclencher via `mvn site` ou `mvn site-deploy`.

2.1. 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)

2.2. Lancement des tests unitaires depuis maven

`mvn test` ---> lance tous les tests

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

`mvn test -Dtest=*xxTest` -> lance tous les tests finissant 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.

3. Configuration de Jenkins

Premiers pas avec Jenkins

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

Installation de Jenkins :

Recopier **jenkins.war** dans **TOMCAT_HOME/webapps** (avec un éventuel Tomcat dédié à l'intégration continue configuré sur le port 8585 ou autre).

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 .

Url de la console "jenkins" :

<http://localhost:8585/jenkins>

Premier menu à activer :

Administrer Jenkins / Configurer le système



Administrer Jenkins / Configurer le système

JDK

Installations JDK



JDK

Nom

openjdk7

JAVA_HOME

/usr/lib/jvm/java-7-openjdk-i386

☐

Install automatically



Ant

Nom

ant_1.9.3

ANT_HOME

/usr/share/ant



Maven

Nom

maven_3.0.5

MAVEN_HOME

/usr/share/maven

☐

Install automatically

Enregistrer

Appliquer

éventuelle installation de plugins pour Jenkins

Menu "Administrer Jenkins / gestion plugins"



Installation/Mise à jour des Plugins

Préparation

- Vérification de la connexion à Internet
- Vérification de la connexion à jenkins-ci.org
- Succès

Git Client Plugin ● Succès

SCM API Plugin ● Succès

Credentials Plugin ● Le plugin credentials est déjà installé. Jenkins doit être redémarré mise à jour soit effective.

Git Plugin ● Succès

Configuration d'un nouveau "job"/"item" au sein de Jenkins

Nom du Item

☐ Construire un projet free-style

Ceci est la fonction principale
Vous pouvez intégrer tous les outils de build.
Il est même possible d'utiliser d'autres outils.

☒ Construire un projet maven

Gestion de code source

- ☐ Aucune
- ☐ CVS
- ☐ CVS Projectset
- ☒ Git

Repositories

Repository URL

[?](#)

[Sauver](#)

[Appliquer](#)

Exemple (pour Tp) :
file:///media/sf_ext/tp/local-git-repositories/env-ic-my-java-app1

Tâche basée (classiquement) sur maven

Build

POM Racine ?

Goals et options ?

Spécifie les goals (cibles Maven) à exécuter, comme "clean install" ou "deploy". Ce champ peut accepter toutes les options en ligne de commande Maven, comme "-e" ou "-Dmaven.test.skip=true".

Le paramétrage principal "*goals et options*" pourra être formulé au moyen de profil(s) "maven" (éventuellement complémentaires).

Exemple :

clean deploy -PfullTest

package -Pprofile1 -Pprofile2

Le paramétrage fondamental des "profils maven" sera étudié dans la séquence suivante.

Lancement sur demande d'un "build" (associé à un job jenkins configuré)



Accès au code source du projet "my-java-app1" (tp) pour effectuer des tests "échecs" ou "réussites" : `/home/formation/Bureau/tp/linux-eclipse-workspaces/eclipse-env-integration-continue.sh`

[Référentiel "GIT" : `/home/formation/Bureau/tp/local-git-repositories/env-ic-my-java-app1` et/ou `https://github.com/didier-mycontrib/env-ic-my-java-app1`]

Et `Calculateur.add()` avec `a+b` ou `a+b+1` puis menu **Team / commit ...**

Affichage des résultats via la console de jenkins

La logique de navigation/sélection de jenkins est la suivante :

Jenkins (server) > **Job (name/type/config)** > **number of instance**
(with status/results)

Exemple:

Jenkins ▶ my-java-app1 ▶ #4

Après avoir sélectionné un des niveaux , on accède à un menu (coté gauche) pour :

- * créer/activer de nouveaux éléments
- * (re)configurer plus en détails l'élément sélectionné
- * afficher des détails sur l'élément sélectionné
- * ...

Concernant les résultats d'un build, la partie la plus intéressante est souvent "*sortie console*" :



Sortie de la console

```
-----  
[INFO] BUILD SUCCESS  
[INFO]  
-----  
[INFO] Total time: 24.003s  
[INFO] Finished at: Tue Apr 21 14:51:42 CEST 2015  
[INFO] Final Memory: 12M/32M  
[INFO]  
-----
```

Notifications élémentaires par flux rss (jenkins)

 [RSS des builds](#)  [RSS des échecs](#)

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 échecs**" 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/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 Outlook).

Pour accrocher un flux rss à *thunderbird*, le mode opératoire est le suivant :

.... / 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 .

Différents types de "builds"

Types de "build"	Commentaires/considérations
Local / privé	Lancé manuellement (et idéalement fréquemment) par le développeur (depuis son IDE) . → <i>Permet de savoir si ses propres changements fonctionnent</i>
Intégration rapide (de jour)	Tests d'intégration rapides déclenchés après chaque commit ou bien régulièrement (ex : toutes les 20 minutes). Seuls les tests rapides (unitaires + intégrations) sont lancés → <i>Permet de savoir si l'assemblage des changements de tous les développeur fonctionne .</i>
Intégration journalière poussée/sophistiquée à heure fixe (nightly build)	Tests sophistiqués (longs) , tests de performance , génération de documentation, de rapports , → <i>Permet de savoir si la dernière version produite du logiciel est en état de marche .</i>

Réglages de fréquence via une syntaxe "crontab" (par exemple dans Jenkins) :

Syntaxe "crontab" :

mm hh jj MM JJ [tâche]

mm représente les minutes (de 0 à 59)

hh représente l'heure (de 0 à 23)

jj représente le numéro du jour du mois (de 1 à 31)

MM représente le numéro du mois (de 1 à 12)

JJ représente le numéro du jour dans la semaine
(0 : dimanche , 1 : lundi , 6 : samedi , 7:dimanche)

Si, sur la même ligne, le « *numéro du jour du mois* » et le « *jour de la semaine* » sont renseignés, alors **cron** (ou) n'exécutera la *tâche* que quand ceux-ci coïncident .

.../...

Réglage de la fréquence des "builds"

Pour chaque valeur numérique (mm, hh, jj, MMM, JJJ) les notations possibles sont :

* : à chaque unité (0, 1, 2, 3, 4...)

5,8 : les unités 5 et 8

2-5 : les unités de 2 à 5 (2, 3, 4, 5)

*/3 : toutes les 3 unités (0, 3, 6, 9...)

10-20/3 : toutes les 3 unités, entre la dixième et la vingtième (10, 13, 16, 19)

Et donc pour un nightly build d'intégration continue :

---> **0 3 * * 1-6** (tous les jours à 3h du matin
sauf les dimanches)

et pour un build rapide de jour :

→ ***/15 8-20 * * 1-5** (tous les jours sauf les week-ends ,
toutes les 15 minutes de 8h à 20h)

Lancement périodique (ex journalier) au niveau de Jenkins

☒ Construire périodiquement

Planning

0 3 * * 1-6

Ce champ suit la syntaxe de cron (avec des différences mineures). Chaque ligne consiste en 5 champs séparés par des TABs ou des espaces :

MINUTES HEURES JOURMOIS MOIS JOURSEMAINE

MINUTES Les minutes dans une heure (0-59)

HEURES Les heures dans une journée (0-23)

JOURMOIS Le jour dans un mois (1-31)

MOIS Le mois (1-12)

JOURSEMAINE Le jour de la semaine (0-7) où 0 et 7 représentent le dimanche

dans cet exemple : tous les jours (de lundi à samedi) à 3h du matin. "nightly build"

Lancement sur détection périodique des changements (SVN/GIT)

☒ Scrutation de l'outil de gestion de version

Planning

`*/*15 8-20 * * 1-5`

dans cet exemple : Jenkins vérifie toutes les 15 minutes si certains changements ont eu lieu au niveau du référentiel de code source (de lundi à vendredi et de 8h à 20h). En cas de changement détecté → lancement (potentiellement très fréquent) d'un build d'intégration.

Suppression des anciens builds (jenkins)

☒ Supprimer les anciens builds

Nombre de jours de conservation des builds

15

si non vide, les enregistrements de build seront conservés au maximum ce nombre de jours

Nombre maximum de builds à conserver

150

si non vide, pas plus de ce nombre de builds ne sera conservé

4. exercices/TP

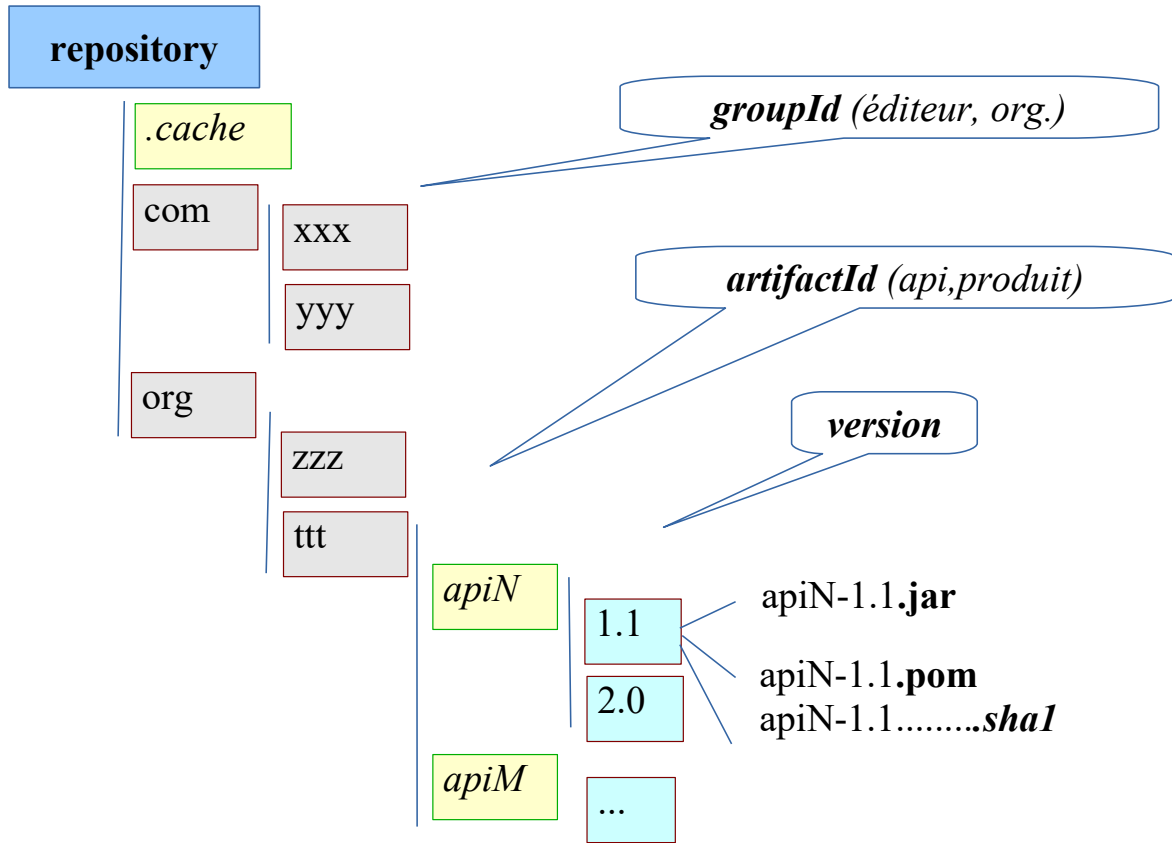
TP (id/num)	objectif_TP	Durée prévue ,
Tp1	Exploration de la configuration essentielle (projet "maven" parent , settings.xml , nexus sans détails , ...) , consoles	20 mn

II - Référentiel maven d'entreprise (nexus oss)

1. Serveur de librairies / Référentiel d'entreprise

Structure d'un référentiel "maven"

(local ".m2" ou distant)



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)

Simplees 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)

\$HOME/.m2/**settings.xml**

paramétrage(s) de

- * proxy-http
- * sécurité (certificats, ...)
- * éventuel "miroir"
- * ...

projetMavenXY/**pom.xml**

paramétrage(s) de

- * référentiels distants (en récupération\$ et/ou en distribution(deploy))
- * ...

Référentiel distant par défaut ("central maven")

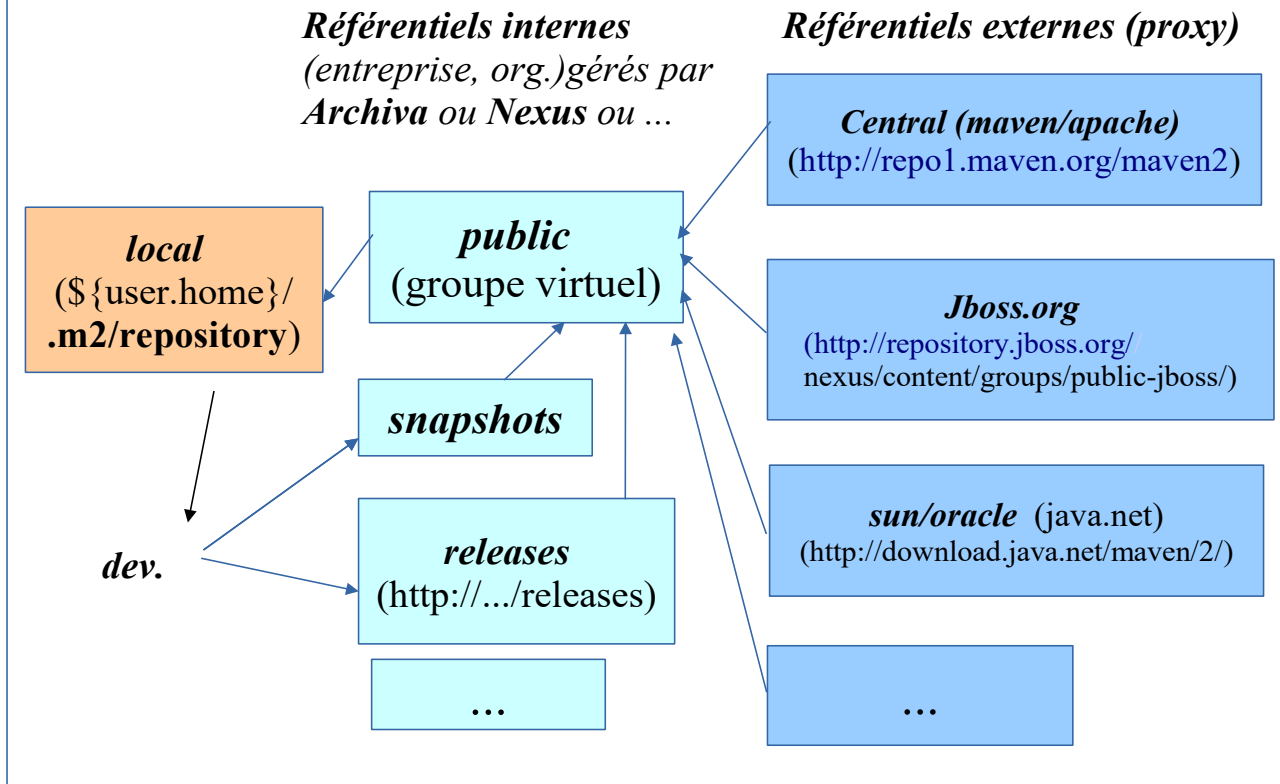
**http://
repo1.maven.org/maven2**

*nouveau référentiel interne
entreprise/organisation*

http://myserver/repo

(*) Proxy-ing (avec copies proches)

Vue globale sur les référentiels "maven"



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, ...) lors des accès distants en écriture (upload)

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

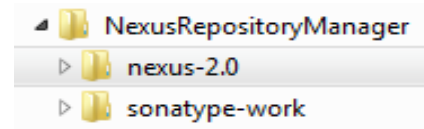
- **Proximity** (assez ancien)
- **Nexus O.S.S.** de "SonaType" (avec bon système d'indexation) et simple à configurer/utiliser
- **Archiva** d'Apache (simple à configurer/utiliser)

Installation et configuration de "Sonatype Nexus OSS"

Il suffit de télécharger l'archive "**nexus-2.10-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 *service 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**"

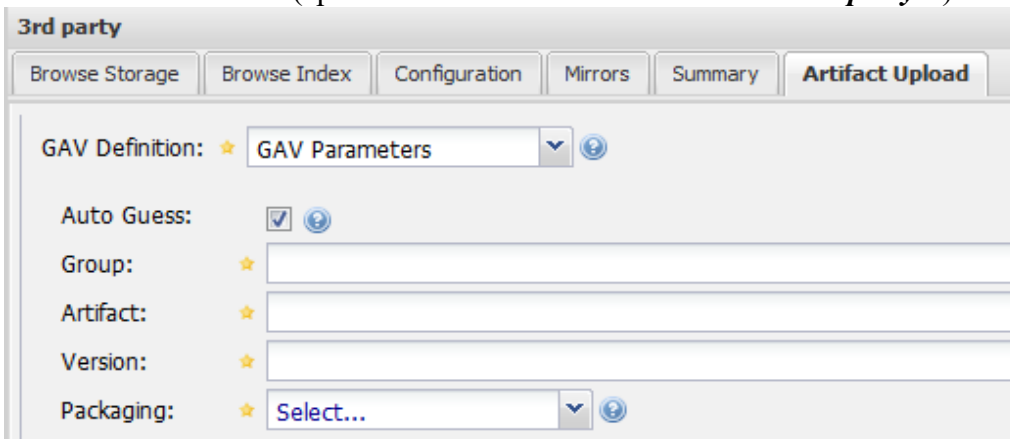
Par défaut (au moment de l'installation) , le mot de passe de l'administrateur (user="**admin**") est "**admin123**". Il peut être évidemment modifié par la suite.

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) .

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**") :



3rd party

Browse Storage Browse Index Configuration Mirrors Summary **Artifact Upload**

Select Artifact(s) for Upload

Select Artifact(s) to Upload...

Filename:

Classifier:

Extension:

Add Artifact

Artifacts

Remove Remove All

Upload Artifact(s) Reset

1.1. 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*)

2. Proxy vers référentiels externes

Configuration des référentiels sous Nexus

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

- **snapshots** (pour les snapshots de notre entreprise/organisation)
- **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 d'un référentiel peut se faire via la console web

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

Repository ID: jboss.org

Repository Name: Jboss public repository

Repository Type: proxy

Provider: Maven2

Format: maven2

Repository Policy: Release

Default Local Storage Location: file:/C:/Prog/java/divers/NexusRepositoryManager/nexus-2.0/./../soni

Override Local Storage Location:

Remote Repository Access:

Remote Storage Location: http://repository.jboss.org/nexus/content/groups/public-jboss/

Configuration des référentiels sous Nexus (suite)

Ajout d'un nouveau référentiel dans le groupe virtuel "public" :

Group Name: Public Repositories

Provider: Maven2

Format: maven2

Publish URL: True

Ordered Group Repositories:

- Releases
- Snapshots
- Central
- Jboss public repository
- Java.net Repository for Maven 2
- 3rd party

Available Repositories:

- Apache Snapshots
- Codehaus Snapshots

Souvent utile :

- 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]

Profil particulier "nexus" à activé dans *settings.xml*

```

<profiles> <profile>
  <id>nexus</id>
  <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>

```

*Configuration
nécessaire*

3. Configuration en lecture

Configuration d'un **miroir** maven pour nexus

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>
```

Configuration facultative mais souvent conseillée

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

Configuration d'un accès en lecture vers **public** de nexus

pom.xml (du projet ou d'un **parent** si héritage)

```
<project> ...
<repositories>
  <repository>
    <id>public</id> <!-- <id>my-internal-repo-id</id> -->
    <url>http://localhost:8484/nexus/content/groups/public/</url>
    <!-- <url>http://myserver/myrepo</url> -->
  </repository>
</repositories>
</project>
```

Configuration nécessaire

il faudra si besoin également ajouter dans **settings.xml** les éléments de sécurité nécessaires:

```
<settings> ...
<servers>
  <server>
    <id>public</id>
    <username>admin</username>      <!-- idéalement autre compte -->
    <password>admin123</password>  <!-- idéalement autre mot de passe -->
  </server> ...
</servers> ... </settings>
```

4. Configuration en écriture (avec droits d'accès)

4.1. Configuration d'un accès "maven" en écriture (pour deploy)

Utilisation (essentielle) 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/formaion 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>
```

Configuration nécessaire

NB :

on peut préférer utiliser le compte prédéfini

username=**deployment**

password=**deployment123** (par défaut)

qui est associé aux droits en écriture vers les référentiels (sans pour autant avoir tous les droits de "admin / admin123")

Paramétrage (dans *pom.xml*) des **URLs pour le déploiement vers nexus**

```

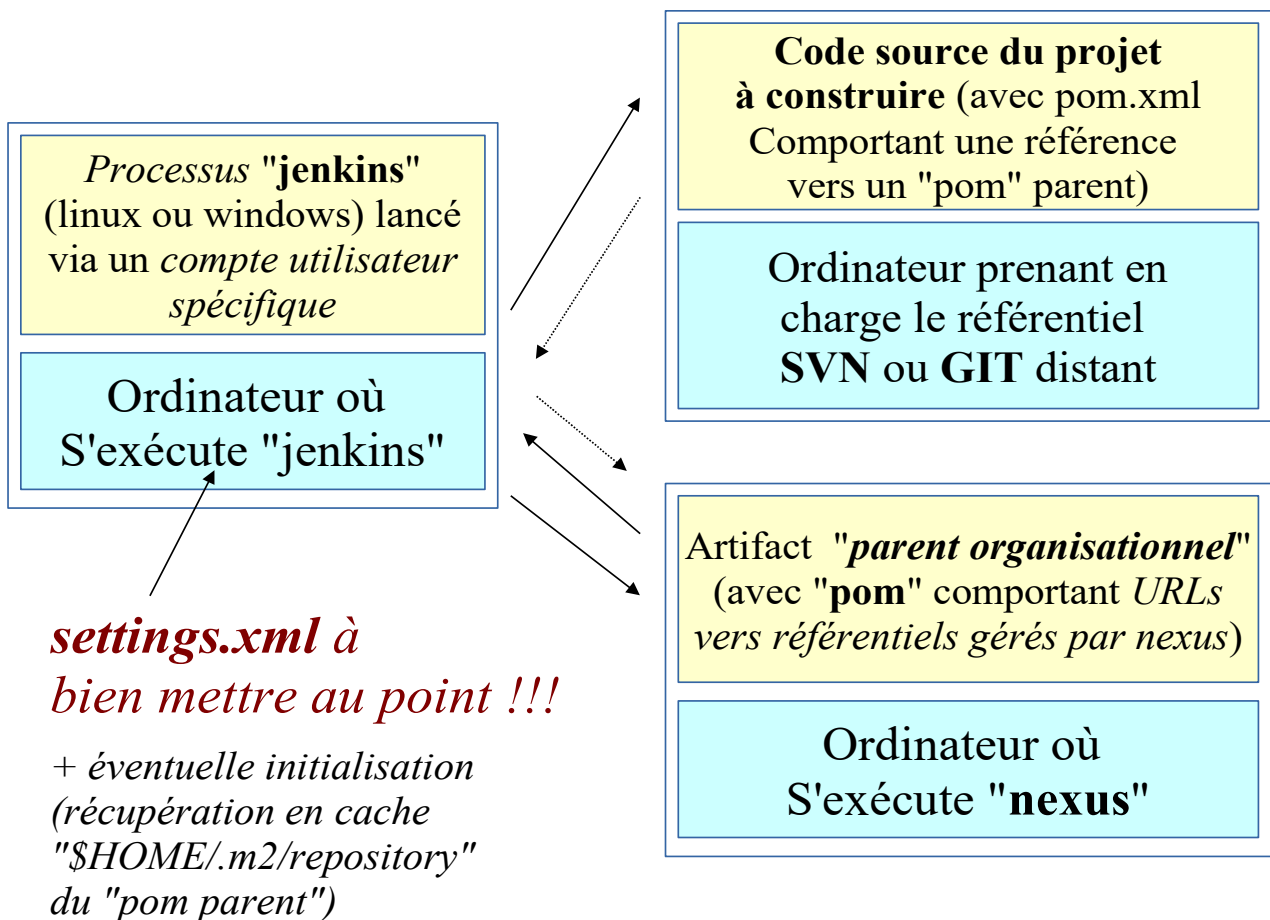
<project>
...
<distributionManagement>

  <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>
    
```

Configuration nécessaire



4.2. Paramétrage d'un compte spécifique (dans nexus) pour le déploiement

Rappel :

le compte prédéfini

username=**deployment**

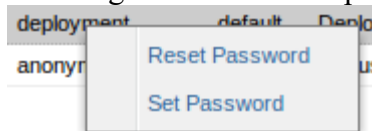
password=**deployment123** (par défaut)

est associé aux droits en écriture vers les référentiels (sans pour autant avoir tous les droits de "admin / admin123")

On peut visualiser tous les utilisateurs prédéfinis via le menu "**security / users**" de nexus oss :

User ID	Realm	First Name	Last Name	Email	Status	Roles
admin	default	Administrator	User	changeme...	Active	Nexus Administrator Role
deployment	default	Deployment	User	changeme1...	Active	Repo: All Repositories (Full Control), Nexus Deployment Role
anonymous	default	Nexus	Anonymous User	changeme2...	Active	Nexus Anonymous Role, Repo: All Repositories (Read)

Le changement de mot de passe (souvent suffisant) s'effectue via le menu contextuel suivant:



(après avoir sélectionné le userId "deployment")

Dans certains cas pointus , on pourra créer un nouvel "user" et associé le rôle adéquat ou bien encore paramétrer un nouveau rôle .

5. Paramétrages selon le contexte

5.1. 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>

```

Si besoin , paramétrer également le proxy http au sein de :

- **jenkins** (configurer le serveur jee (ex : tomcat) qui fait fonctionner jenkins.war sur l'aspect "proxy http")
- **navigateur internet** (firefox ou chrome ou ...)
- **eclipse** (menu "windows / preferences / general / network connection / ... ")
- **nexus** (menu "administration / serveur / Default HTTP proxy settings")

→ penser à paramétrer une exception pour **localhost 127.0.0.1** (accès sans proxy http)

6. Exercices / TP autour de "nexus oss"

TP (id/num)	objectif_TP	Durée prévue ,
TP2a	Configuration d'un repo externe (mode proxy) + tests	20 mn
TP2b	Ajustement de certains paramètres(changement du mot de passe pour "deployment" , ajuster en conséquence les droits d'accès dans settings.xml , ...) + config lecture vers "public"	20 mn

III - Projet "maven" multi-modules , cargo , JEE

1. Maven et Jenkins (liaisons basiques)

1.1. Rappels fondamentaux sur "maven" et les 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* ,
pas inséré dans .war construit
(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.

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>javax.servlet-api</artifactId>
      <version>3.0.1</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>1.7.7</version>
      <scope>compile</scope>
    </dependency> ...
  </dependencies>
  <build>....</build>
</project>

```


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>
        <version>3.2</version>
        <configuration>
          <source>1.7</source>
          <target>1.7</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>4.1.1.RELEASE</org.springframework.version>
    <org.apache.myfaces.version>2.2.5</org.apache.myfaces.version>
  </properties>
  <dependencies> <dependency>...
    <version>${org.springframework.version}</version>
  </dependency> ...</dependencies> <build>....</build>
</project>

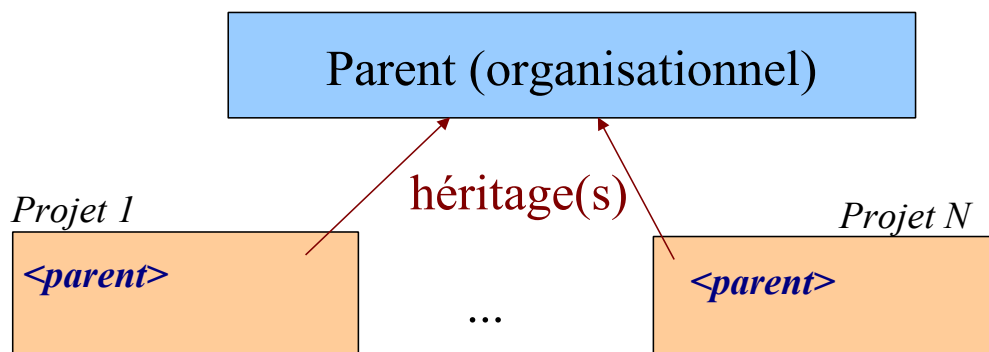
```

Héritage "organisationnel" au niveau de "maven"

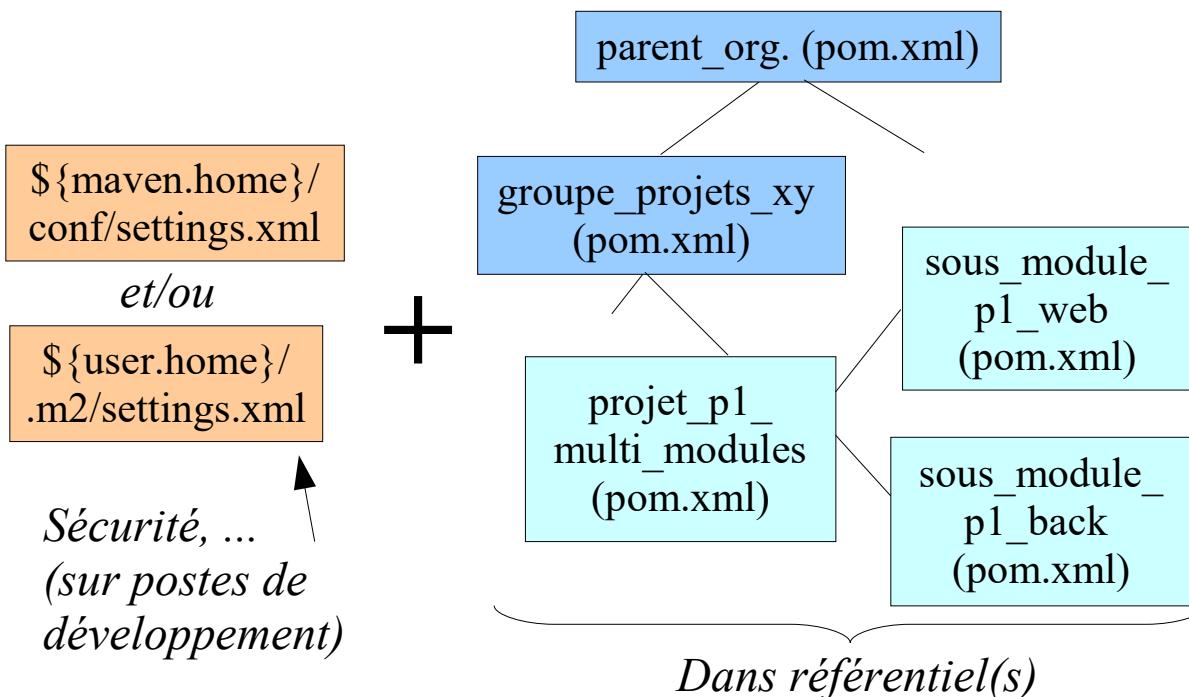
Dans l'absence d'une organisation multi-modules, la balise **<parent>** d'un fichier "**pom.xml**" permet de définir un lien d'héritage entre le projet courant et le projet parent :

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".



Vue globale sur la configuration "maven"



1.2. 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 .

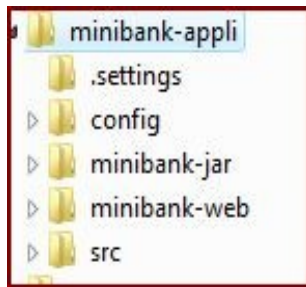
<i>variables</i>	<i>Significations (contenus)</i>
<code>\${basedir}</code>	Répertoire contenant le fichier pom.xml
<code>\${version}</code> (équivalent à <code>\${project.version}</code>)	Version du projet courant
<code>\${project.build.directory}</code>	Répertoire "target"
<code>\${project.build.outputDirectory}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

Maven et Junit sur un projet java simple ,
exécution des tests unitaires au sein de
Jenkins

Remontée des résultats (statistiques , console
jenkins)

2. Maven et applications multi-modules

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



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 (parent)

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

Arborescence globale conseillée:

my-global-app

```
pom.xml
| minibank-jar (module de services)
|   ...
|   pom.xml
| mywebapp
|   pom.xml
|   ...
```

Structure plate déconseillée (mais partiellement supportée):

my-global-app

```
pom.xml      (avec chemins relatifs vers sous modules en "../xy" )
minibank-jar (module de services)
  ...
  pom.xml
mywebapp
  pom.xml
  ...
```

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>
```

3. Configuration "maven" pour applications "JEE"

3.1. 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 (eventuellement 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>
    -->

    <!-- CFX 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-webappl-web</finalName>
</build>
</project>

```

3.2. 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.jee5appl</groupId>
    <artifactId>my-jee5appl</artifactId>
    <packaging>pom</packaging>
    <version>1.0-SNAPSHOT</version>
    <modules>
        <module>my-jee5appl-ejb</module>
        <module>my-jee5appl-web</module>
        <module>my-jee5appl-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>
  </dependencies>
  ....

```

Exemples de configurations "maven" pour des application JEE multi-modules (avec "Spring" ou avec "EJB3")

Illustration via appli web

4. Exemple de configuration du plugin "cargo" pour déploiement vers serveur Jee

Exemple de configuration du plugin "cargo"

```

... <plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <version>1.4.12</version>
  <executions>
    <execution>
      <id>start-container</id>
      <phase>pre-integration-test</phase>
      <goals> <goal>start</goal> </goals>
    </execution>
    <execution>
      <id>stop-container</id>
      <phase>post-integration-test</phase>
      <goals> <goal>stop</goal> </goals>
    </execution>
  </executions>
  <configuration>
    <wait>false</wait>
    <container>
      <containerId>jetty8x</containerId>
      <type>embedded</type>
      <dependencies> ....</dependencies>
    </container>
  </configuration>
</plugin>

```

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.

4.1. 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>

```

4.2. 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>jetty8</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>

```

4.3. 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>
</plugins>
</build>

```

```

    </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>jboss51</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"

Exemples de configuration du plugin "cargo" pour les principaux serveurs JEE (ex : Jetty, Tomcat, Jboss)	
---	--

5. Exercices/Tp

TP3a (facultatif)	Configuration et lancement d'un build "maven" au sein de "Hudson/Jenkins" pour une application java simple .	20 mn
TP3b (facultatif)	Ajustement des test unitaires , analyse des résultats remontés par jenkins (console , e-mails) .	15mn
TP3c (important)	Ajustement de la configuration maven associée à une application "JEE/Spring" (plugin "cargo" ,)	15mn

IV - Test d'intégration (via plugin failsafe)

1. Phases pour les tests d'intégration

Positionnement des tests d'intégration avec maven

Dernières phases du cycle "build" de maven :

Phases	Tâches prévues	Exemples
package	Fabriquer les artifacts (packages)	Construire ".jar" , ".war" , ".ear" dans "target"
pre-integration-test	Préparer l'environnement nécessaire pour les tests d'intégration	Lancer un serveur d'application (Tomcat ou) via le plugin cargo. Eventuellement initialiser le contenu d'une base de données.
integration-test	Lancer des tests d'intégration (basés sur l'ensemble de l'application (avec modules assemblés)	Lancer des tests "http" via selenium ou ...
post-integration-test	Post-traitements (intégration)	Arrêter un serveur d'application,...
verify	Vérifier certaines conditions/ résultats / statuts	
install	Recopier l'artifact construit dans le référentiel local	Vers \$HOME/.m2/repository
deploy	Déployer le ".jar" , ".war" ou ".ear" construit vers un référentiel maven	Ex: déploiement vers "nexus"

2. plugin "failsafe" et tests d'intégration

Plugin "failsafe" et conventions "IT"

Par défaut , aucun plugin (et aucun "goal") n'est associé aux phases "pre_integration-test" , "integration-test" , "post-integration-test" et "verify" du cycle de construction de maven.

Il faut donc **configurer** dans un fichier *pom.xml* quelques plugins (et "goal") qui seront alors **explicitement** associés aux phases proches de "integration-test" .

En règle générale , on utilisera principalement le plugin "*maven-failsafe-plugin*" qui est une version dérivée de "*maven-surefire-plugin*" spécialement conçue pour les **tests d'intégration** .

	maven-surefire-plugin	maven-failsafe-plugin
Utilité principale	Tests unitaires	Tests d'intégration
Phases associées	test	pre-integration-test integration-test post-integration-test verify
Phase où se constate l'échec	test	verify
Classes prises en charge (par défaut)	**/*Test*.java **/*Test.java **/*TestCase.java	**/*IT*.java **/*IT.java **/*ITCase.java
Répertoire de sortie (par défaut)	\${basedir}/target/surefire-reports	\${basedir}/target/failsafe-reports

Par défaut , le plugin "*maven-failsafe-plugin*" déclenchera tous les tests qui se trouvent dans des classes java qui sont situées dans la branche habituelle "*src/test/java*" et qui ont des noms comportant "**IT**" au début ou à la fin :

****/*IT*.java**

****/*IT.java**

Exemple de configuration du plugin "*maven-failsafe-plugin*" dans *pom.xml* :

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-failsafe-plugin</artifactId>
  <version>2.18</version>
  <executions>
    <execution>
      <goals>
        <goal>integration-test</goal>
        <goal>verify</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Exemples classiques de tests d'intégration

Tests d'intégration sur une application "java/web" :

- **démarrer un serveur d'application** (ex : Jetty) au niveau de la phase "*pre-integration-test*" via par exemple le plugin "cargo"
- **lancer quelques tests d'intégration** (**/*IT.java) basés sur **Junit** et **Selenium** de façon à tester si l'application construite et déployée fonctionne bien (page d'accueil accessible ,) .
- **stopper un serveur d'application** (ex : Jetty) au niveau de la phase "*post-integration-test*" via par exemple le plugin "cargo"
- **constater l'échec (ou la réussite) des tests** (et du cycle maven) au niveau de la phase "*verify*".

Autre types de tests d'intégration (sans selenium):

- Tester un service web SOAP via l'api JAX-WS
- Tester un accès distant (RMI) vers un EJB

3. Tp (test d'intégration)

Mise en œuvre d'un test d'intégration "JUnit et "maven/failsafe/IT" (ajustements au sein d'un build privé sous eclipse puis lancement d'un build automatique via Hudson/jenkins).
Ce tp est "sans selenium" (ex: test page accueil accessible via URL / get ou bien test de web service)

V - Test d'intégration web via selenium

1. Selenium (presentation et installation)

1.1. Présentation de Selenium

La suite "Selenium" (de *seleniumHQ*) permet de simplement mettre en place des **tests automatisés d'interfaces graphiques web** (basées sur *HTTP* et *HTML*).

Deux logiciels complémentaires et une extension optionnelle :

- **Selenium-IDE** = *plugin pour firefox* permettant d'**enregistrer un dialogue HTTP** (pour le rejouer plus tard de façon paramétrable au sein des futurs tests) .
- **Selenium-WebDriver**= api utilisée pour lancer les tests .
- Selenium-grid = produit facultatif de la suite Selenium pour lancer des tests en parallèle .

NB : Grâce aux enregistrements effectués par Selenium-IDE , on peut assez facilement produire des "tests web" sans avoir absolument à connaître une syntaxe de script particulière .

C'est cette facilité d'utilisation qui a fait la popularité du produit Selenium.

L'autre atout de Selenium est *sa licence open source "Apache 2.0"* permettant de diffuser librement le produit .

1.2. Installation du plugin "Selenium-IDE" pour firefox :

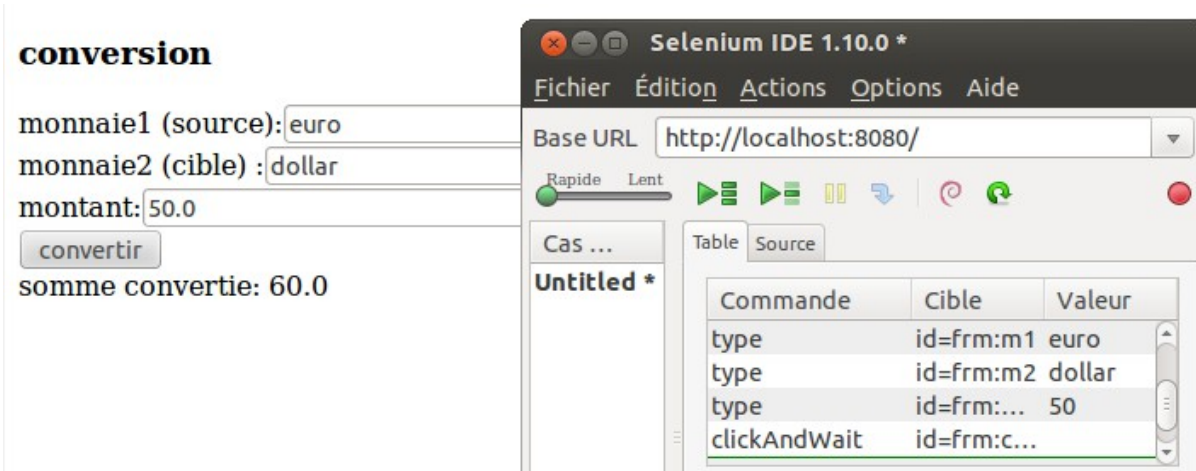
Depuis un navigateur "**Firefox**" récent lancer l'installation du plugin "*selenium-ide*" via une URL ressemblant à la suivante (ici en version 2.8) :

<http://release.seleniumhq.org/selenium-ide/2.8.0/selenium-ide-2.8.0.xpi>

NB : *Les versions récentes de Selenium permettent d'utiliser "WebDriver" à la place de "Selenium-server" !!!!*

2. Enregistrement séquence "web" via selenium-IDE.

Enregistrement de séquence web via selenium-IDE



Etant positionné sur la page d'accueil d'une application web (ex : index.html) on peut déclencher un nouvel enregistrement de séquence "html/http" en activant le menu **"Outils / Selenium IDE"** de **Firefox** .

Il suffit ensuite d'utiliser normalement l'application (clicks sur liens hypertextes , saisies de valeurs dans des formulaires , ...) pour que toutes les actions effectuées par l'utilisateur soient au fur et à mesure enregistrées par le plugin firefox "Selenium IDE" .

Une fois la séquence terminée, il faut cliquer sur l'**icône rouge** "*en cours d'enregistrement, cliquer pour terminer l'enregistrement*".

Enregistrement de séquence web via selenium-IDE (suite)

Pour visualiser la séquence enregistrée, il suffit ensuite de :

- choisir une vitesse (Rapide ou lent)
- cliquer sur l'un des triangles verts ("rejouer l'enregistrement")

A partir de la séquence enregistrée , il est possible de générer directement une classe de test basée sur JUnit4 pour java via le menu "**Fichier / Exporter le test sous ... / Java / JUnit4 / web driver**" de "**Selenium-IDE**".

En choisissant un nom de type "**SequenceYyyyIT.java**", cette classe de test pourra être utilisée (*de façon "à peine remaniée"*) en tant que test d'intégration déclenchable par maven .

NB : *Les versions récentes de Selenium permettent d'utiliser "WebDriver" à la place de "Selenium-server" !!!!*

Cette évolution apporte deux intérêts :

- plus besoin de démarrer et arrêter "selenium-server" au niveau de la configuration du plugin maven pour selenium
- possibilité d'utiliser **HtmlUnitDriver** à la place de *FirefoxDriver* de façon à ce que les tests puissent s'exécuter sans réel navigateur internet (et donc sans aucune interface graphique) sur un serveur d'intégration potentiellement placé sur un ordinateur sans écran.

Attention (petit piège JSF) :

Il faut absolument définir des "id" explicites au niveau de chaque composant des pages ".jsp" ou ".xhtml" de JSF pour que ceux-ci soient stables entre l'enregistrement et les futurs tests (sinon id générés automatiquement et non stables) :

`<h:form id="frm">`

`monnaie1 (source):<h:inputText id="m1" value="#{conversion.monnaie1}"/>
`

`...`

3. Tests d'intégration avec selenium

Selenium au sein d'un test d'intégration "maven"

Dépendance "maven" pour utiliser "selenium-java" dans les classes de tests :

```
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <!-- <artifactId>selenium-server</artifactId> -->
  <artifactId>selenium-java</artifactId>
  <version>2.44.0</version>
  <scope>test</scope>
</dependency>
```

Ceci permet d'utiliser les types "**HtmlUnitDriver**" et "**WebDriver**" dans les classes de tests. Exemple :

```
package tp.app.zz.it.test;

import static org.junit.Assert.fail; import java.util.concurrent.TimeUnit;
import org.junit.After; import org.junit.Before; import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
//import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.htmlunit.HtmlUnitDriver;
.../...
```

test d'intégration basé sur selenium (suite)

```

public class SequenceWebDriverIT {
    private WebDriver driver; private String baseUrl;
    private StringBuffer verificationErrors = new StringBuffer();

    @Before
    public void setUp() throws Exception {
        //driver = new FirefoxDriver();//visible browser during test
        driver = new HtmlUnitDriver(); //invisible browser (with limitations)
        ((HtmlUnitDriver)driver).setJavascriptEnabled(true);
        baseUrl = "http://localhost:8080/";
        driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
    }

    @Test
    public void testIT() throws Exception {
        driver.get(baseUrl + "/my-spring-jeeapp1-web/");
        driver.findElement(By.linkText("bienvenue")).click();
        driver.findElement(By.linkText("conversion")).click();
        driver.findElement(By.id("frm:m1")).clear();
        driver.findElement(By.id("frm:m1")).sendKeys("euro");
        driver.findElement(By.id("frm:m2")).clear();
        driver.findElement(By.id("frm:m2")).sendKeys("dollar");
        driver.findElement(By.id("frm:montantInput")).clear();
        driver.findElement(By.id("frm:montantInput")).sendKeys("60");
        driver.findElement(By.id("frm:convertButton")).click();
    } ... }

```

```

    @After
    public void tearDown() throws Exception {
        driver.quit();
        String verificationErrorString = verificationErrors.toString();
        if (!"".equals(verificationErrorString)) {
            fail(verificationErrorString);
        }
    }
}

```

4. Tp (selenium + test intégration / web)

Enregistrement selenium + réajustage classe de test java	15mn
Lancement du test selenium via "maven/failsafe/IT" (build privé sous eclipse puis lancement d'un build automatique via Hudson/jenkins).	15mn

VI - Bon usage des profils "maven" (ic)

1. Rappels sur profils "maven"

Bien utiliser la notion de « profils maven »

La technologie "maven" a été conçue pour lancer automatiquement les tests (avec un éventuel paramétrage fin sur les tests à lancer ou pas).

Ex: `mvn test , mvn package -DskipTest=true ,`

Dans la plupart des projets, on a besoin de lancer des tests selon un **contexte variable** (environnement de développement , d'intégration , de recette , de pré-production ,).

Pour chaque contexte, il faudra que les tests s'adaptent à des configurations spécifiques des ressources utilisées par l'application (ex : base de données HsqSql ou MySql ou , listes d'utilisateurs sous forme de liste xml ou bien récupérés depuis annuaire LDAP,) .

La technologie "maven" comporte heureusement la **notion de profils (variantes pouvant cohabiter dans la configuration "pom.xml" d'un projet)**.

Par exemple, un profil pourra permettre d'effectuer des tests élémentaires (avec une petite base HsSql et sans ldap) et un deuxième profil pourra déclencher des tests plus proche de la future production (avec base "Mysql" ou "Oracle" et avec un annuaire LDAP) .

Un profil de maven peut être associé à la notion de **filtrage de ressources** : **remplacement de variables** `${db.username}` , `${bd.url}` , .. au sein des **fichiers internes de configurations** (ex : Spring,) de l'application à partir de valeurs figurant dans le paramétrage des profils de maven (dans pom.xml) .

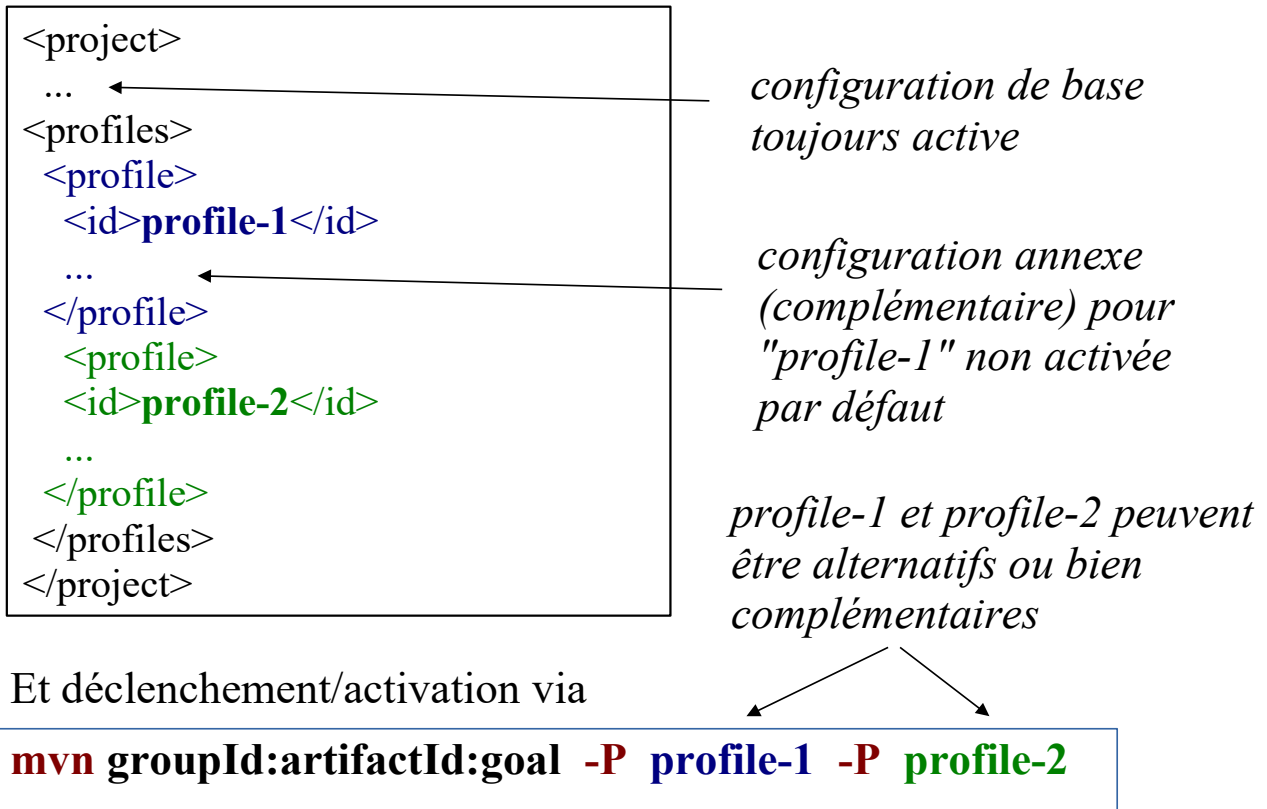
Une autre solution assez classique consiste à utiliser le filtrage de ressource sur des variables de type `${spring.subConfigurationFileXY}` de façon à contrôler des **"switchs"** entre différents sous fichiers de configuration (spring ou ...) en fonction de l'environnement cible associé au profil.

Attention : dans tous les cas, il est indispensable de **bien documenter** le paramétrage et l'activation des profils "maven" de l'application.

Principales variations en fonction des profils :

- * le compilateur java (jdk 1.4 ou 1.5 ou 1.6 ou 1.7)
- * les ressources (url et type de base de données , annuaire ldap , ...)
- * les jeux de données à charger pour les tests (fichiers xml pour DbUnit ,...)

Syntaxe la plus courante pour les profils "maven"



Filtrage des ressources (maven)

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) *en fonction d'un choix de profil*.

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 pour le filtrage des ressources

```

<project ...>. ...
  <profiles>
    <profile> <id>p1</id>
      <properties>
        <xxx.yyy>jetty</xxx.yyy> <!-- valeur 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

2. Profils pour tests rapides avec base h2 ou hsSql

Build type "rapide" à déclencher fréquemment le jour

Profil maven de type "*quick_build*" avec :

- Tests unitaires simples (*éventuel filtrage des tests importants à lancer*)
- si possible serveur d'application léger de type "**jetty**" si tests d'intégration sur une partie "web"
- Base de données rapide (en mémoire) de type "**hsqldb**" ou "**h2**"

3. Génération de livrables et de "release"

3.1. 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*)

Qualificateurs:

SNAPSHOT (Maven) : version en évolution

alpha1 : version alpha (très instable et incomplète)

beta1, b1, b2 : version bêta (instable)

rc1, rc2 : release candidate

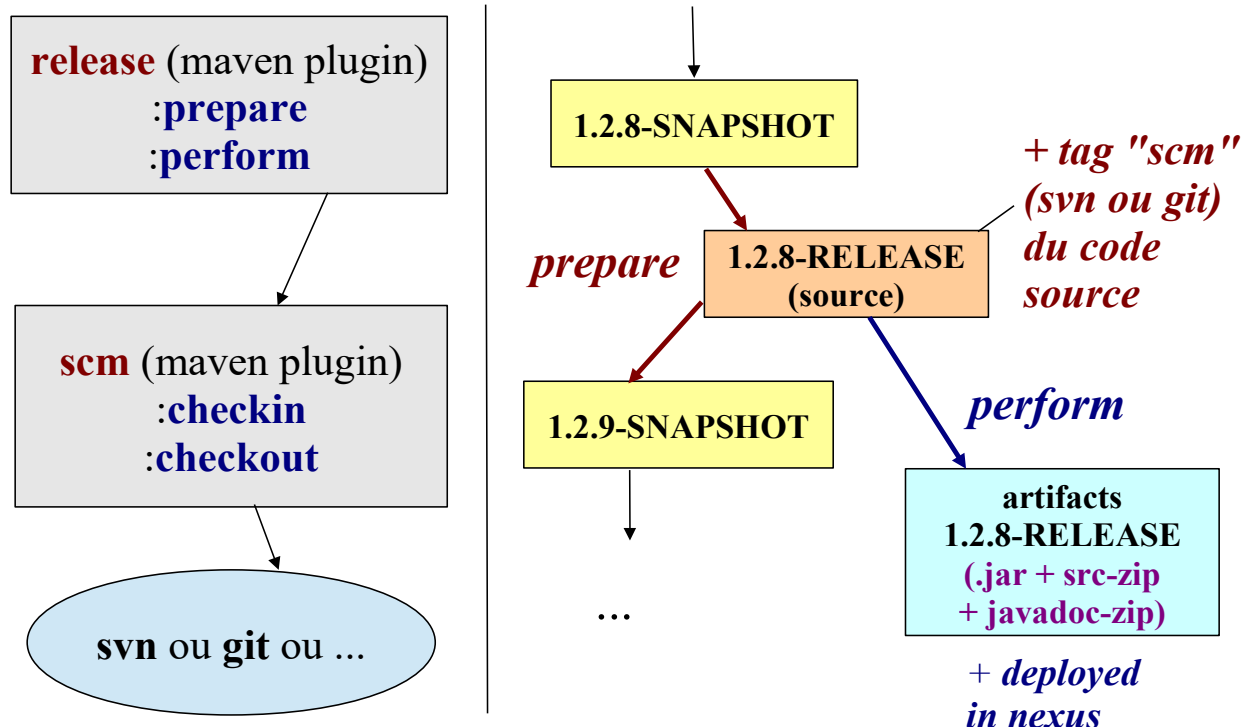
m1, m2 : milestone

ea : early access

GA , RELEASE , Final , : version mature (utilisable en production)

3.2. Plugins "scm" et "release"

Changement de version "maven" bien contrôlé et génération/déploiement de "released"



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 committing 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" .

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)
# * 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
```

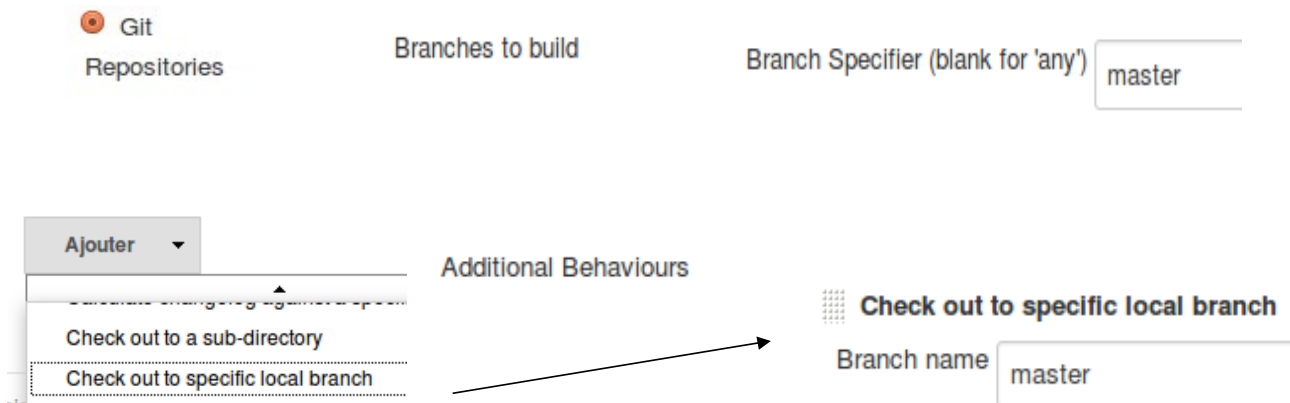
NB: il existe "mvn **release:rollback**" en cas d'échec de "mvn **release:prepare**".

Remède à un problème spécifique "jenkins+git" :

Le lien entre "jenkins" et "mvn release" fonctionne très bien avec SVN. Avec git , on obtient par défaut le message d'erreur suivant :

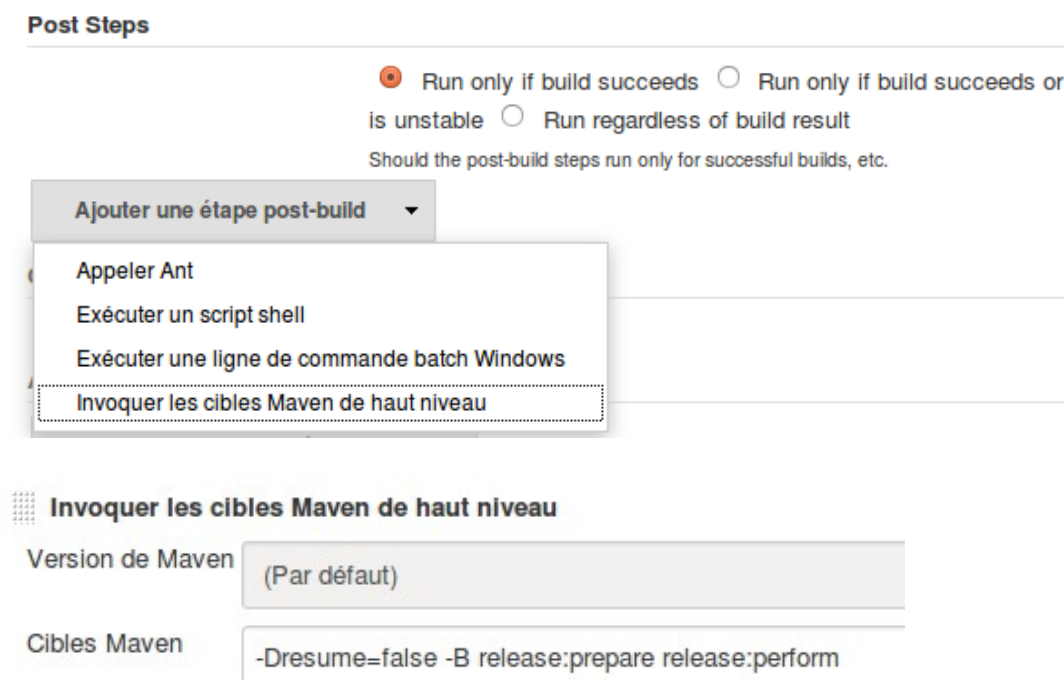
**"Detecting the current branch failed: fatal:
ref HEAD is not a symbolic ref "**

Comme remède à ce problème , on peut peaufiner la configuration "GIT" dans jenkins de la façon suivante :



Déclenchement (automatique) d'une release maven depuis Jenkins

En tant qu'alternative possible au plugin "*m2 release plugin*" on peut configurer automatiquement une "*post step build*" basé sur "*maven*" sur un "nightly build/job" :



assemblies

4. Profils pour tests sophistiqués (analyse , perf ,...)

Build type "sophistiqué" à déclencher la nuit

Profils et commandes "maven" de type "*full_build*" avec :

- * tests poussés (avec détails fins)
- * éventuel utilisation d'une base de données évoluées (mysql , db2 , oracle , ...) avec un important "jeu de données".
- * tests de performance ,
- * **lancement d'analyse qualimétrique** (via sonarQube ou autre).
- * **génération de "release" (si pas d'erreur) et incrémentation de la version "snapshot" (de développement)**
- *



5. Tp (profils "maven")

Ajustements interne de profils "maven" (selon base de données , selon type de build "journalier/ de nuit" ,)

VII - Aspects divers et avancés ("maven" et "ic")

1. Génération de rapports , de "javadoc" , ...

→ sonarQube (cours EJ15T)

→ annexe "aspects divers et avancés maven" de ce support de cours

2. Bonnes pratiques "maven"

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.persistance.dao" , " tp.myapp.xy.impl.persistance.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-persistance-framework (.jar , DAO génériques basés sur Hibernate ou JPA)

my-xy-framework (.jar , autre petit framework réutilisable)

3. Gestion avancée des dépendances (BOM)

3.1. 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"

3.2. import de gestion de dépendances et "BOM"

"BOM" signifie "*Bill Of Materials*" (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 richFaces4 (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 .

ANNEXES

VIII - Annexe – Archetype "maven"

1. Notion d'archetype

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+xf_cxf_pour_tc6" *à mettre au point*
 - * ...

2. Utilisation d'un (nouvel) archetype maven

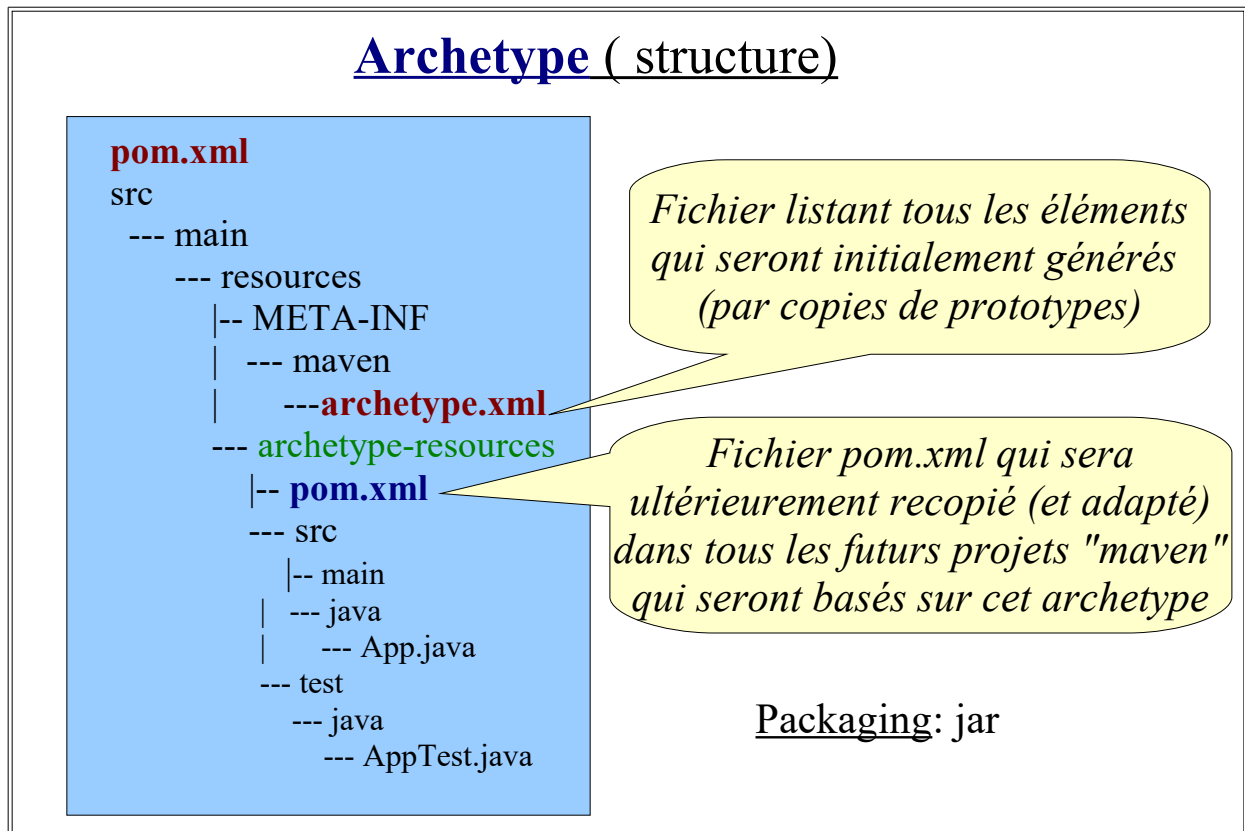
mvn archetype:generate

```
-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 , ...) .

3. Création d'un nouvel archetype maven



3.1. 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 .

IX - Aspects divers et avancés de "maven"

1. 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>
  </webResources>
</configuration>
...
```

2. Activations de profils

2.1. 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>
```

2.2. 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>
```

2.3. 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

2.4. 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>

```

2.5. 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. 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**" : Maven *POJO*) 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*", ...)

3.1. 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>
```

4. Programmation d'un plugin pour maven

4.1. 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 groupIdOfPlugin:artifactIdOfPlugin: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>
```

4.2. 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")

pom.xml

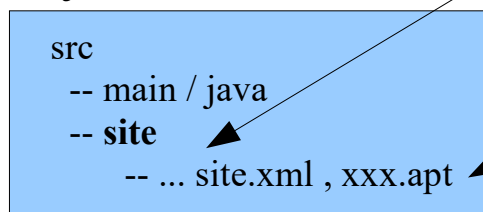
```
<pluginMetadata>
  <mojos> <mojo>
    <goal>hello</goal>
    <call>hello</call> <!-- Ant -->
    <description> ...</description>
  </mojo> </mojos>
</pluginMetadata>
```

```
<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>
```

5. Génération et publication d'une documentation

Génération de documentation avec "maven"

Projet maven



Partie "site" initialement ajoutée via
 mvn archetype:create
 -DarchetypeArtifactId=maven-archetype-site

apt = *almost plain text*
 format de type "wiki"

Pour générer
 la documentation (target)
mvn site

mvn site-deploy

Selon distributionManagement/site/url (pom.xml)

Site web où publier
 la 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)

6. Javadoc (via maven)

Générer javadoc avec "maven"

... (dans pom.xml)

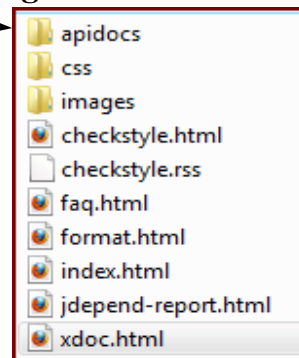
```

<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>
  </plugins>
  ...
</reporting>

```

mvn javadoc:javadoc

target/site/



7. Rapports avec maven

rapports avec "maven"

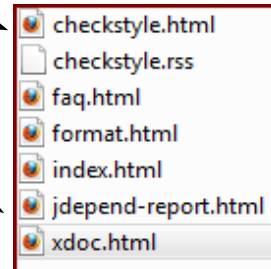
... (dans pom.xml)

```
<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 - Annexe – m2e (maven2eclipse)

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'eclipse 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"

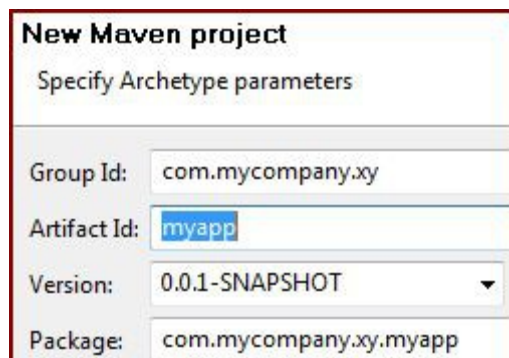
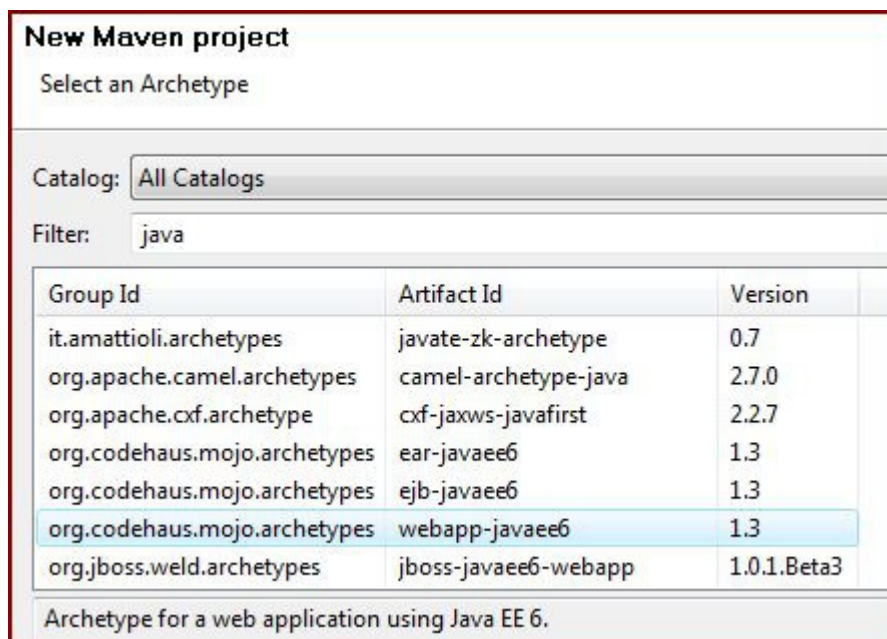
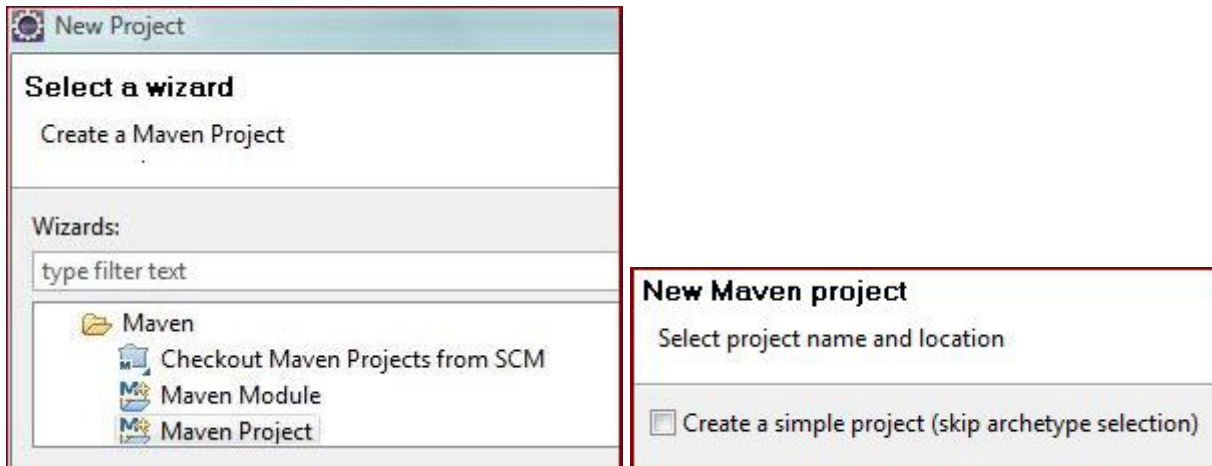
Installer si nécessaire dans eclipse 3.x ou 4.2 le plugin eclipse "m2e" via le "update site" suivant:

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

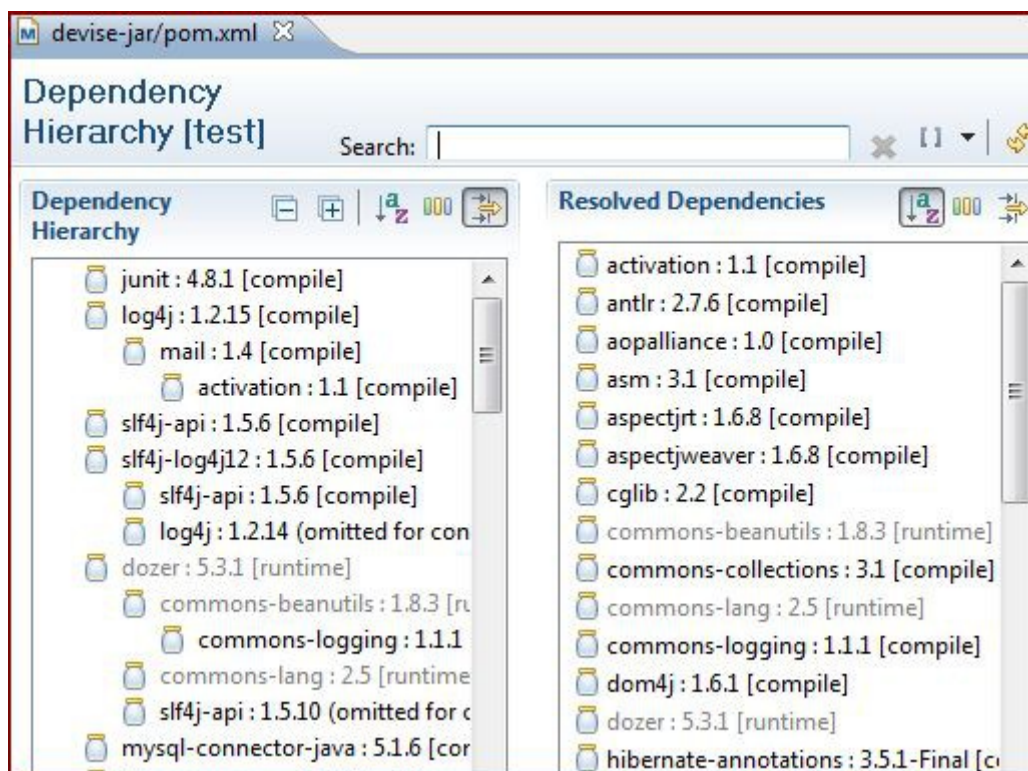
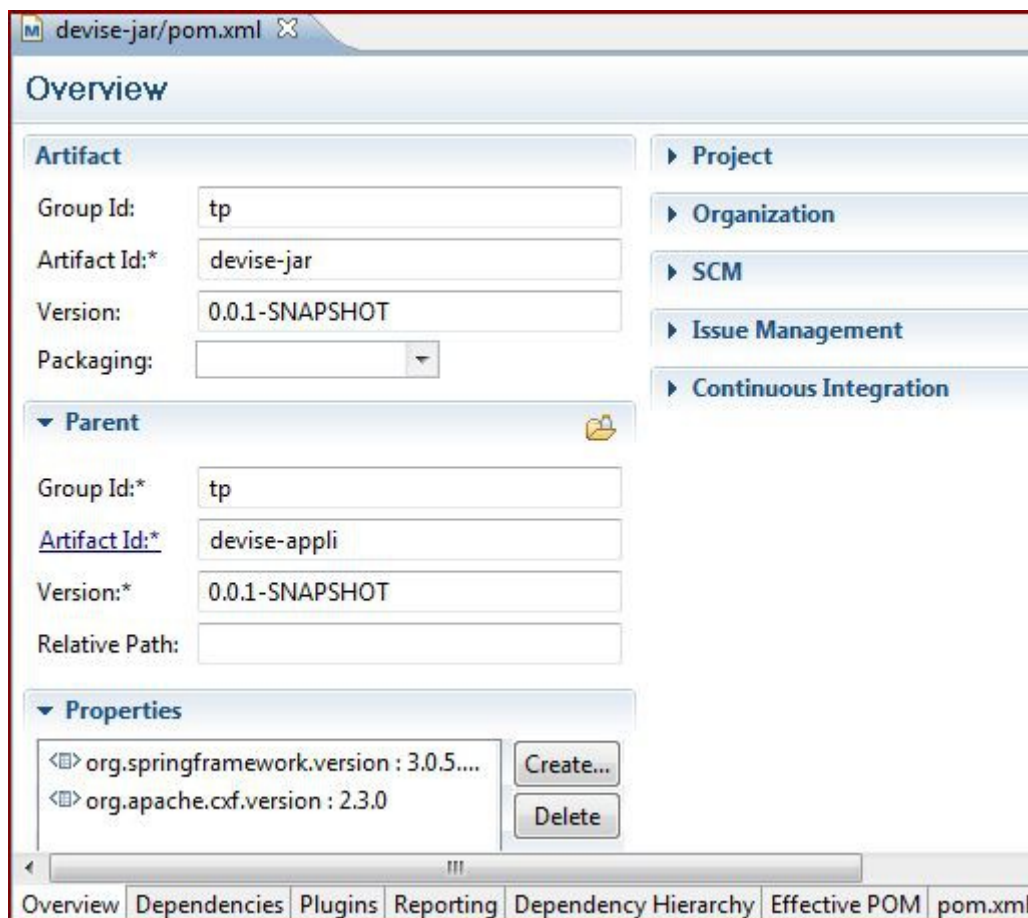
- + nouveau projet "maven"
- + éditer manuellement le fichier "pom.xml"
- + éditer le code au bon endroit (dans src/main/java,)

- 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



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

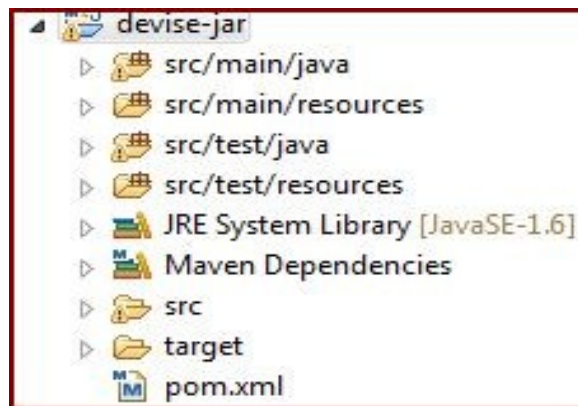


1.4. Structure des projets "eclipse maven"

Plugin eclipse "m2e" pour maven (partie 2)

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

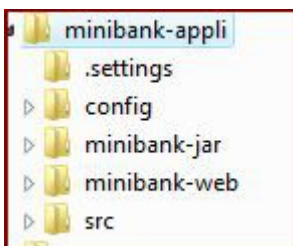
- * ont une structure "maven" classique (*src/main/java, ...*)
- * n'ont pas la structure classique d'un projet java (*src, bin*)
ni la structure d'un "dynamic web projet" (*webContent, ...*)



Cas d'une structure multi-modules:



(structure vue à plat dans eclipse)



(structure arborescente sur le file system hôte).

Remarque importante:

Le **plugin eclipse "m2e"** gère un **cache** dans le répertoire **.m2/repository**.

Activer le menu "**project / maven / update dependencies**" permet normalement de **remettre à jour le cache (ré-indexation)** et permet de faire en sorte de la config eclipse s'ajuste à la config maven.