

intégration continue

(avec
« *git* »
+ « *maven et nexus* »
.
+ « *sonar* »
+ « *jenkins* »)

Table des matières

I - Intégration continue: présentation.....	4
1. Intégration continue.....	4
2. Maven et l'intégration continue.....	5
3. Principaux objectifs de l'intégration continue.....	6
4. Chaîne d'intégration continue.....	7

II - Rôle du système de gestion des sources.....	9
1. Rôle « scm » dans l'intégration continue.....	9
III - GIT.....	12
1. Présentation de GIT.....	12
2. Principales commandes de GIT (en mode local).....	16
3. initialisation d'un projet GIT (en local).....	19
4. Index (staging area).....	20
5. indispensable .gitignore.....	22
6. Commit et tags.....	23
7. Commandes de GIT pour le mode distant.....	25
8. Gérer plusieurs référentiels distants.....	26
9. initialiser git en mode distant.....	30
10. pistage (track) entre branches locales et distantes.....	31
11. Git fetch et git pull.....	33
12. Plugin eclipse pour GIT (EGIT).....	35
IV - Rôle du « builder » (ant, maven) et scripts.....	36
1. Constructions et déclenchement des tests (ic).....	36
2. Différents types de "builds".....	38
3. Exemple de script ant pour de l'intégration continue.....	39
V - Essentiel Maven pour intégration continue.....	40
1. Présentation de Maven.....	40
2. Principales fonctionnalités de maven.....	40
3. Intérêts de Maven.....	41
4. Evolutions, versions.....	41
5. Fonctionnement de maven.....	42
6. Variables d'environnement à bien régler.....	44
7. Mise en oeuvre de Maven.....	45
Buts (goals) et phases.....	47
8. Quelques options importantes.....	51
9. POM (Project Object Model).....	51
10. Structure & syntaxes (pom.xml).....	55
11. Configuration multi-modules (avec sous projet(s)).....	59
12. Héritage entre projets "maven" (<parent>).....	62
13. Archetypes.....	65
14. Utilisation d'un (nouvel) archetype maven.....	65

15. Création d'un nouvel archetype maven.....	66
16. Tests unitaires avec maven.....	67
17. Lien entre maven et eclipse (m2e).....	68
VI - Référentiel maven , Nexus , profils maven.....	75
1. Mise en place d'un référentiel "Maven"	75
2. Repository Manager (Nexus ou ...).	80
3. Profils "maven".....	85
4. Filtrage des ressources.....	89
5. Ajout (et éventuel filtrage) de ressources "web" externes.....	90
VII - Tests d'intégration avec maven (et selenium).....	92
1. Tests d'intégration avec maven.....	92
2. Tests web (http/html) via selenium.....	96
VIII - Lien maven / scm, gestion des releases.....	101
1. Plugins "scm" et "release" de maven.....	101
IX - Intégration continue avec Hudson / Jenkins.....	105
1. Premiers pas avec Hudson/Jenkins.....	105
2. Notifications des résultats (rss , email , ...).	109
3. Différents types de "builds" (avec Jenklins).....	114
X - Rapports qualimétriques / miniAudit (Sonar).....	117
1. Rapport / métriques (avec Sonar).....	117
XI - Annexe – Compléments sur Maven.....	123
1. Variables et versions (maven).....	123
2. Bonnes pratiques.....	125
3. Déploiement Jee avec CARGO (via maven).....	125
4. Configuration "maven" pour applications "JEE"	128
5. Gestion avancée des dépendances (BOM).....	135
6. Génération et publication d'une documentation.....	138
7. Javadoc (via maven).....	140
8. Rapports avec maven.....	141

I - Intégration continue: présentation

1. Intégration continue

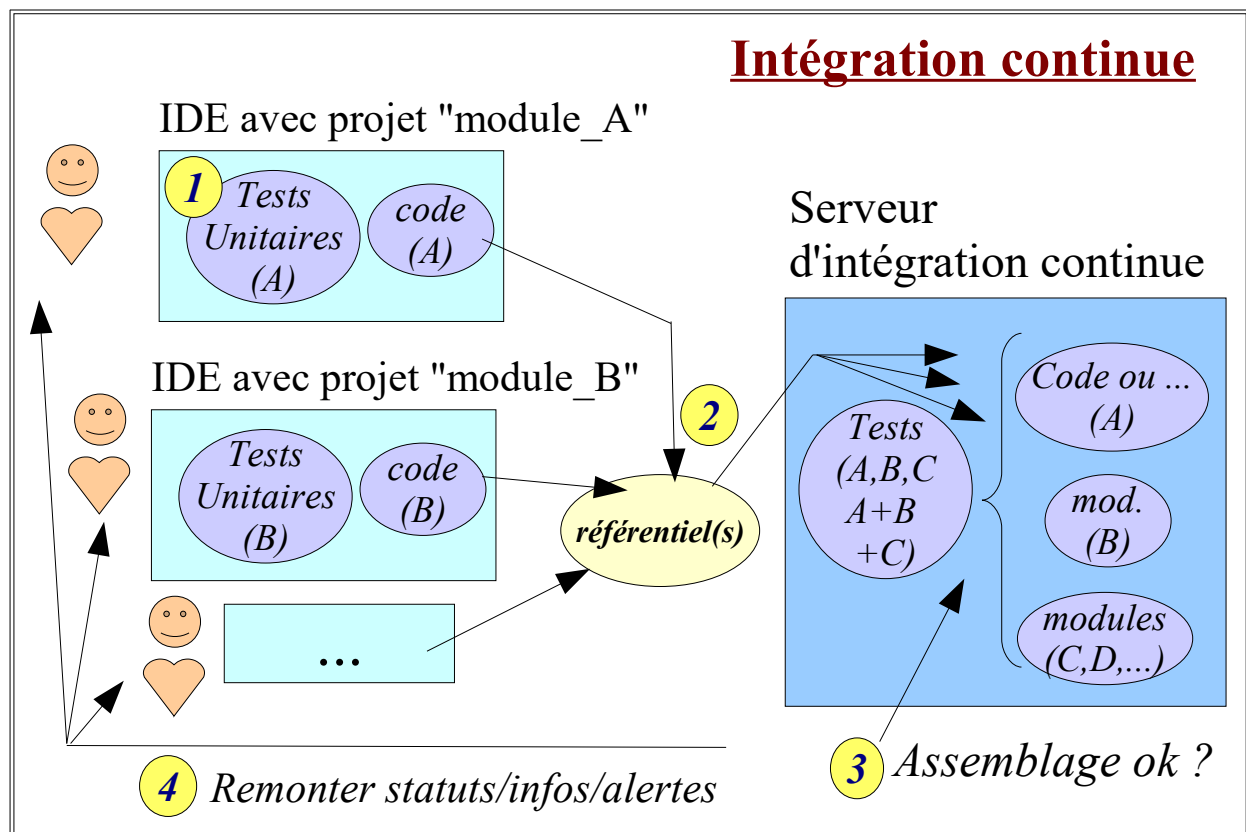
Intégration continue

Principes (intégration continue):

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

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

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



2. Maven et l'intégration continue

Maven et intégration continue

Principaux logiciels d'intégration continue (monde Java):

- **CruiseControl** (le pionnier)
- **Hudson** (évolution récente: **Jenkins**)
- **Continuum** (apache)

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

....

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

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

NB: Le serveur **Jenkins** reste aujourd'hui l'un des plus utilisé dans un contexte interne d'entreprise.

Ceci dit , l'entreprise **GitLab** (concurrente de GitHub) propose des fonctionnalités d'intégration continue d'un bon niveau (semblable au pipeline de jenkins) et gérant bien le contexte du cloud computing.

En gros, avec **Jenkins** on fait un peu ce que l'on veut (via une multitude de plugins) mais il faut configurer son serveur Jenkins.

Avec **GitLab CI** on dispose d'une solution un peu plus clef en main avec avec certaines limitations ou dépendances.

3. Principaux objectifs de l'intégration continue

Sur un gros (ou moyen) projet, chaque développeur se concentre une partie bien précise et génère des simples composants devant ultérieurement être assemblés entre eux pour produire l'application complète.

Sans automatisme, il faut alors manuellement:

- vérifier que les différents composants soient bien compatibles (mêmes versions, prévus pour s'interfacer entre eux)
- rassembler/packager les composants dans des modules exécutables (.exe, .dll, .jar, .ear, ...)
- déployer le tout sur un serveur d'application
- lancer des tests globaux

Ce qui peut prendre beaucoup trop de temps !!!!!

Pour rester concurrentielle, une SSII/ESN ou une maîtrise d'œuvre interne doit s'appuyer sur un système automatisant la plupart des points précédents.

Un tel serveur système dit "d'intégration continue" va (à peu près):

- récupérer le code source du composant dans un référentiel (SVN ou GIT ou ...) dans une structure neutre (indépendante de l'ide)
- recompiler ce code source (pour bien contrôler la version du compilateur utilisé)
- relancer (dans un contexte contrôlé) des jeux de tests unitaires
- packager le code compilé d'un composant ou d'un module dans une archive adéquate (.jar, .war, .ear)
- déployer éventuellement l'ensemble sur un environnement spécifique de tests (JVM, serveur d'application, ...)
- lancer éventuellement des tests d'intégrations (ou globaux) qui ont été préalablement préparés
- remonter des messages et des statistiques vers les développeurs
- générer et stocker une nouvelle version du logiciel si les tests ont réussi.

Dans le monde java, la plupart des environnements d'intégrations continues sont basés sur l'une et/ou l'autre des trois technologies fondamentales suivantes:

- **ANT** (sorte de makefile en XML et donc indépendants de la plate-forme) *10 % des projets*
- **MAVEN** (gestionnaire de projet indépendant de l'IDE) *70 % des projets java*
- **GRADLE** (plus flexible et moderne que maven, pas XML mais DSL) *20 % des projets*

3.1. Tester très régulièrement une application complète (fruit d'un assemblage de modules)

L'intégration continue est tout à fait dans l'esprit des **méthodes agiles (XP, Scrum, ...)** et du **développement piloté par les tests (TDD : Test Driven Development)**.

Le fait que le logiciel satisfasse les tests est un **indicateur concret** permettant de suivre l'évolution du projet.

Une grande partie de l'intérêt de l'intégration continue tient dans le fait de pouvoir relancer très régulièrement et fréquemment toute une série de tests même si le logiciel à tester est un assemblage de modules fabriqués à partir d'environnement de développement (IDE) très variés.

Un module peut être développé avec VsCode sous Windows ; un autre avec IntelliJ sous Macintosh et encore un autre avec Eclipse sous linux.

(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 (bibliothèques, ...) 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** .

Ce code source (idéalement basé sur une structure maven) contiendra généralement :

- les packages et les classes java (code du module / de l'application)
- les ressources de configuration (ex : fichiers de config pour Spring+Hibernate+cxfr)
- les ressources WEB (images , web.xml , faces-config.xml, pages HTML et JSP , ...)
- les classes java et les ressources (données/configurations) pour les tests unitaires
- des variantes dans la configuration exprimées sous forme de profils maven .
- liste des dépendances (bibliothèques) utiles pour le module (expression dans pom.xml)

Le gestionnaire d'intégration continue (exemple : **Hudson / jenkins**) 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
-

Autrement dit , mettre en œuvre une plate-forme d'intégration continue consiste à

- installer et configurer de façon cohérente les différents logiciels (Nexus , Sonar , SVN ou GIT , Hudson, ...)
- bien configurer les fichiers maven (pom.xml) des modules de l'application.

La configuration maven est essentielle car elle comporte sous forme de profils/variantes les paramètres utiles pour les tests , les bonnes reconstructions et les analyses de sonar.

4.2. Evolution récente/moderne de l'intégration continue:

- de plus en plus intégré dans des conteneurs "**dockers**"
- configuration **jenkins** de plus en plus en mode **pipeline**
- multi-langages (java et javascript et autres)
- dans un cadre "**DevOps**" pas que de l'intégration continue mais aussi de la **livraison ou déploiement continu**.

En anglais : **CI** : Continuous Integration

CI/CD : CI et Continuous **D**elivery or **D**eployment

Métaphore classique: **usine logicielle**

II - Rôle du système de gestion des sources

1. Rôle « scm » dans l'intégration continue

SCM = Source Control Manager (ex : CVS , SVN , GIT , Mercurial , ...)

1.1. Skocker (de façon versionnée) le code source produit

Le gestionnaire de code source sert essentiellement à stocker l'intégralité du code source (avec toute la configuration maven et les jeux de tests) .

Ce référentiel (versionné) de stockage est accessible depuis toutes les parties concernées par le développement du projet :

- accès depuis les IDE des développeurs (workspaces "eclipse" , "intelliJ" , "vsCode" , ...)
- accès depuis la configuration maven (plugin "scm" et "release" de maven)
- accès depuis le logiciel d'intégration continue "Hudson /jenkins" ou GitLab-CI

Dans certains cas (fréquents), on sera amené à paramétrer différents accès au référentiel :

- un accès restreint en lecture seule
- un accès en lecture/écriture (url accompagnée d'une authentification à paramétrer)
- ...

1.2. Référentiel (code de référence)

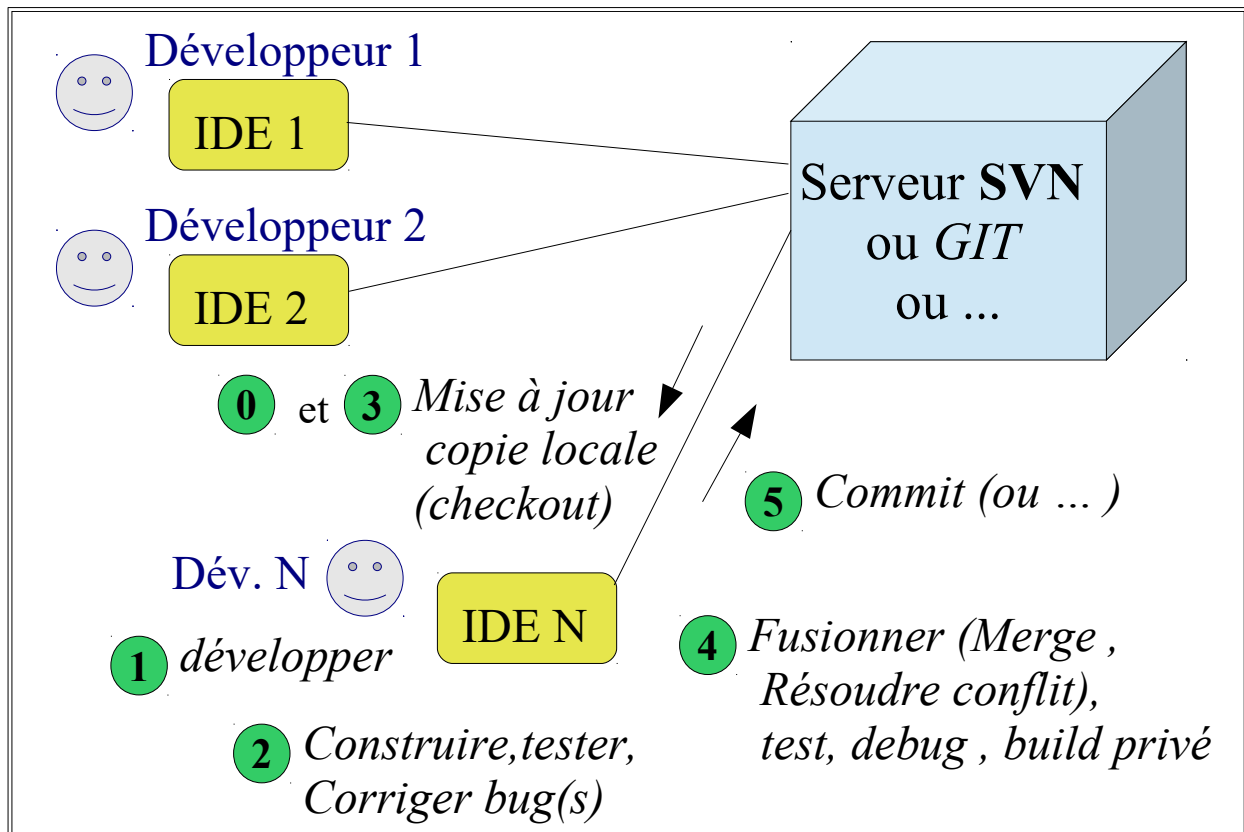
Un référentiel "SVN" ou "GIT" permet de partager des versions de référence pour le code d'un projet qui sans cela serait anarchiquement réparti en plusieurs parties potentiellement incohérentes sur les différents postes de développement.

1.3. Alimenter l'intégration continue

Le gestionnaire de code source ("GIT" , "SVN" ou "Mercurial") alimente le logiciel d'intégration continue d'une des différentes façons suivantes :

- pull périodique effectuer depuis Hudson (ex : toutes les heures ou tous les jours)
- pull sur demande (depuis la console d'administration de Jenkins)
- pull événementiel déclenché par la détection d'un commit (ou autre) sur le référentiel
-

1.4. Bonne utilisation de SVN (ou ...) pour l'intégration continue



Processus de développement dans un contexte d'intégration continue :

- 0) Le développeur récupère une copie du dernier code source en date à partir du référentiel SVN ou git (checkout).
- 1). Il effectue tout un tas de modifications au niveau du code dont pour accomplir sa tâche (développement d'une nouvelle partie, refactoring, évolution diverse, ...).
- 2) le développeur lance un build privé (avec des tests unitaires) sur sa machine.
Il corrige évidemment les bugs en cas de problème.
- 3) Lorsque le build réussit, le développeur peut commencer à songer à commiter.
Le problème est que d'autres développeurs auront peut-être commité leurs codes avant lui (ce qui est fréquemment le cas). Pour le vérifier, le développeur doit mettre à jour sa copie locale (via checkout ou git pull) pour récupérer les dernières modifications.
- 4) Il faut ensuite (au cas par cas) adapter le code (merge,) et reprendre le processus à l'étape "2 – lancer un build privé "pour tester le bon fonctionnement local.
- 5) Une fois que le build réussit avec le code synchronisé (par itération du sous cycle formé par les étapes "2,3,4"), le développeur pourra enfin commiter ses changements.
- 6) le serveur d'intégration détecte ensuite le code commité et lance automatiquement un build d'intégration continue et le développeur reçoit un mail quelques secondes plus tard lui indiquant le statut du build (échec ou réussite de l'assemblage global et des tests d'intégration).

1.5. Bonnes pratiques de développement (intégration continue)

<i>Bonnes pratiques</i>	<i>Commentaires/considérations</i>
Petits "Commit" fréquents (<i>early/often commit</i>)	Pour obtenir une bonne synchronisation entre les morceaux développés par différents développeurs et pour détecter et corriger au plus tôt l'impact d'un bug non détecté par un test unitaire mais ayant un effet de bord indirect sur les autres parties
Ne pas commiter du code erroné (<i>never commit broken broken code</i>) <i>et</i> Lancer régulièrement des builds privés/locaux	À prendre au sens "bug personnel" ou "incompatibilité avec code d'un collègue" .
Corriger rapidement les builds ratés <i>et</i> Garder si possible les builds d'intégration dans le vert tout au long de la journée (en collaborant avec les autres développeurs).	Ceci doit être considéré comme la priorité principale de chacun des membres de l'équipe
Ecrire des tests automatisés (sans interaction utilisateur) . <u>Exemple</u> : <i>test unitaire JUnit4_5</i> <i>et</i> Tous les tests (et les rapports qualimétriques) doivent réussir/passé .	<i>"tester c'est douter" est une blague .</i> <i>L'intégration continue ce n'est pas que de la compilation continue !!!</i>
Eviter de récupérer du mauvais code	Quand le build est dans le rouge , il ne faut pas récupérer le code récemment modifié par d'autres développeurs(avec des bugs) mais plutôt aider ces développeur à corriger leurs bugs.

Evidemment , ces conseils généraux doivent être adaptés aux spécificités de chaque projet. Par exemple, dans le cas d'une expérimentation, on peut imaginer aucun commit tant que pas bien au point localement.

III - GIT

1. Présentation de GIT

1.1. Fonctionnement distribué de GIT

Présentation de GIT

GIT est un **système de gestion du code source** (avec prise en charge des différentes **versions**) qui fonctionne en **mode distribué** .

GIT est moins centralisé que SVN . Il existe deux niveaux de référentiel GIT (local et distant).

Un référentiel GIT est plus compact qu'un référentiel SVN.

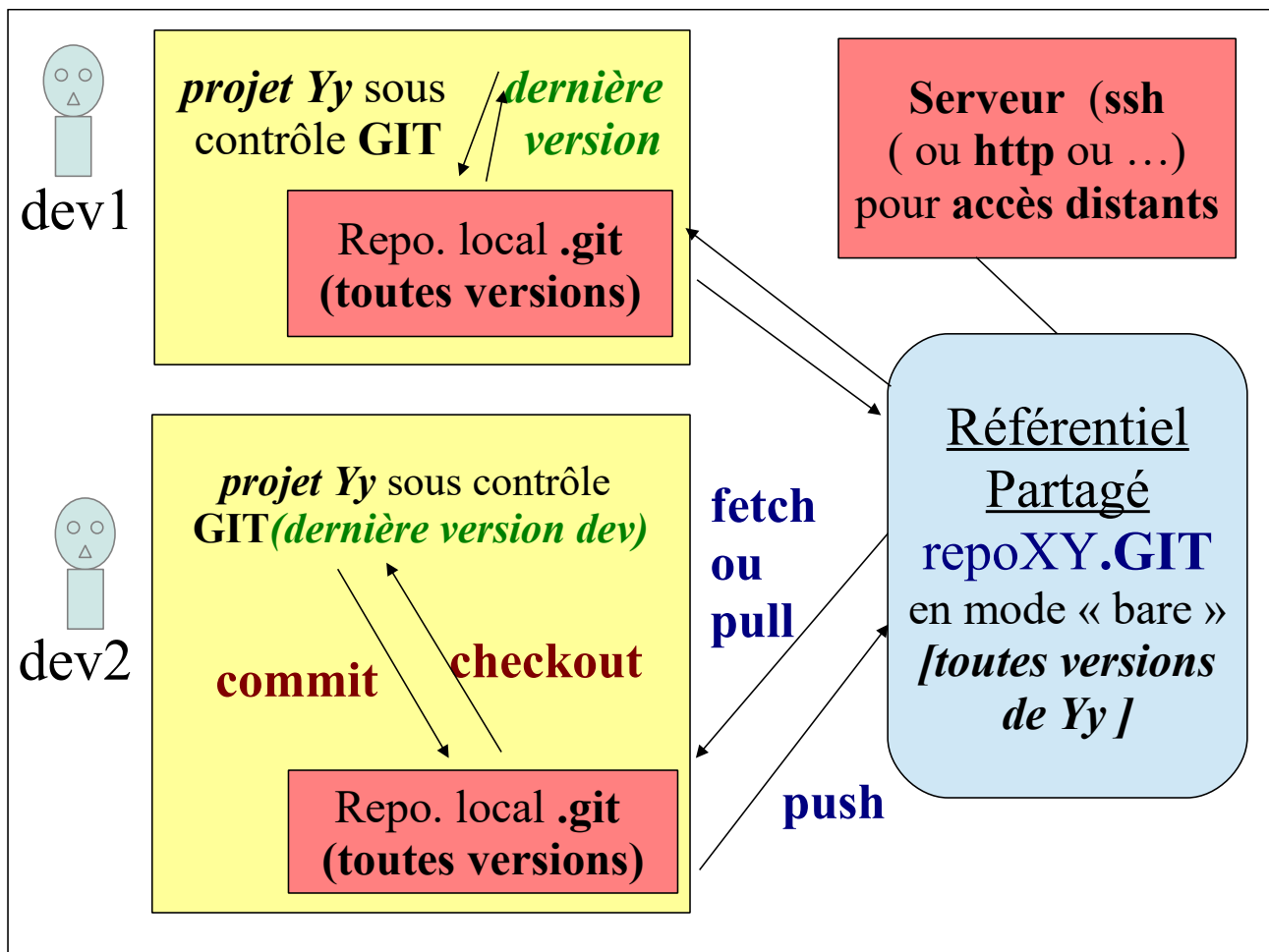
GIT a été conçu par **Linus Torvalds** (l'inventeur de **linux**) .

Un produit concurrent de GIT s'appelle « **Mercurial** » et offre à peu près les mêmes fonctionnalités.

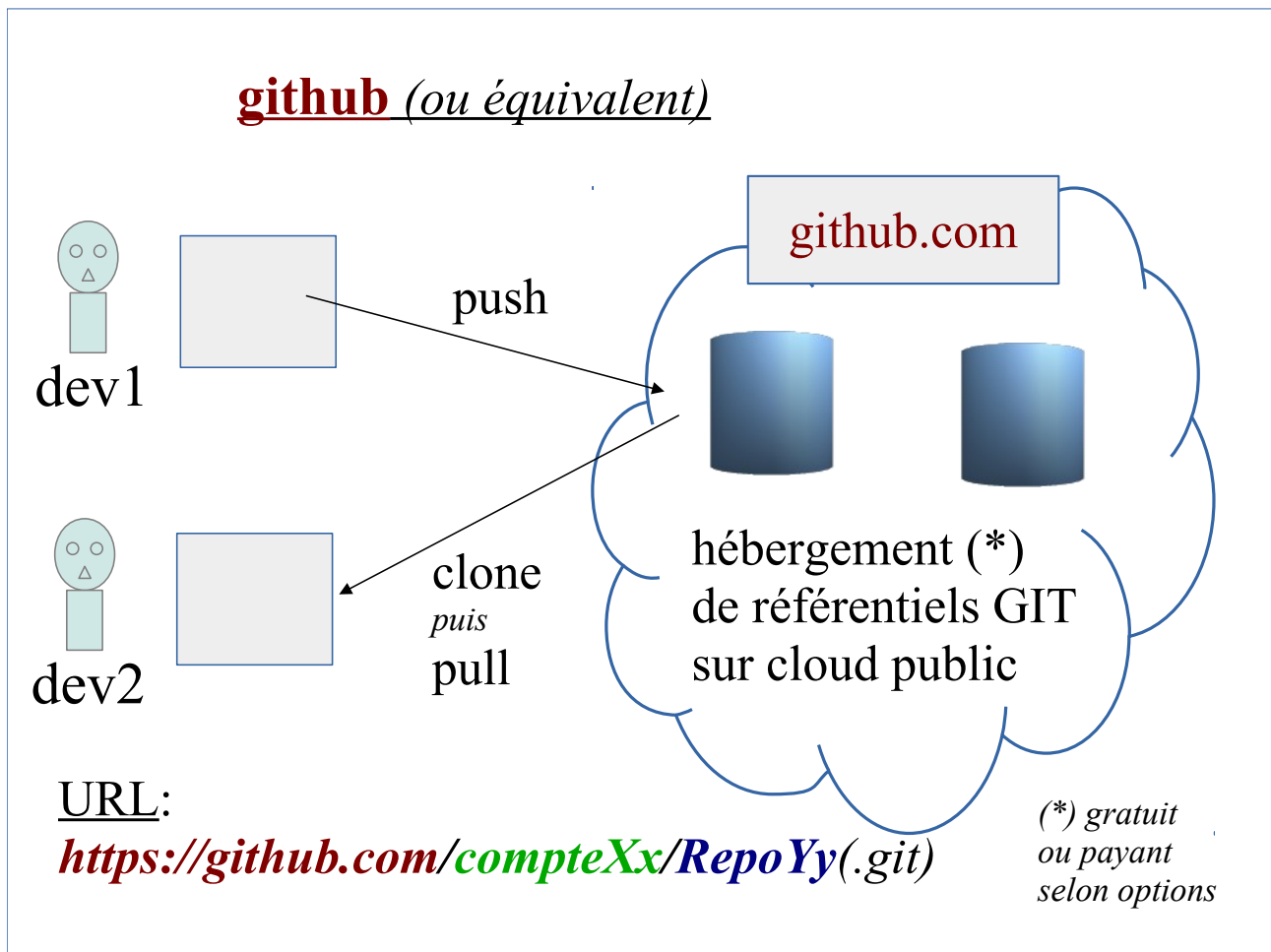
Mode distribué de GIT

Dans un système « scm » centralisé (tel que CVS ou SVN) , le référentiel central comporte toutes les versions des fichiers et chaque développeur n'a (en général) sur son poste que les dernières versions des fichiers.

Dans un **système « scm » distribué** (tel que **GIT** ou **Mercurial**) , le référentiel central ne sert que pour échanger les modifications et chaque développeur a (potentiellement) sur son poste toutes les versions des fichiers.



1.2. Hébergement de référentiel GIT



Principaux sites d'hébergement de référentiels GIT :

- **github**
- **gitlab**
- **bitbucket**
-

1.3. Installation et configuration minimum :

- 1) installer "*Git for windows*" ou bien "*Git sur linux*" (via **yum** ou **apt-get** ou autre) .
- 2) **git --help**
- 3) **git config**

*En bref, les commandes «**commit**» et «**checkout**» de **GIT** permettent de gérer le référentiel **local** (propre à un certain développeur) et les commandes «**push**» et «**fetch / pull**» de **GIT** permettent d'effectuer des **synchronisations** avec le **référentiel partagé distant** .*

Configuration locale de GIT

Installation de GIT sous linux (debian/ubuntu) :

```
sudo apt-get install git-core
```

Configuration locale:

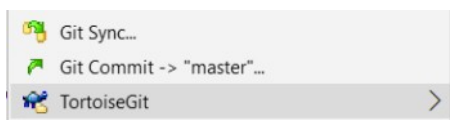
```
git config --global user.name "Nom Prénom"  
git config --global user.email "poweruser@ici_ou_la.fr"  
#...
```

pour voir ce qui est configuré :

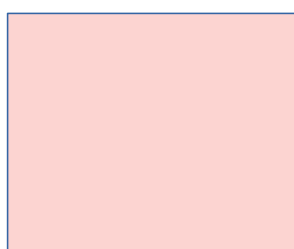
```
git config --list
```

1.4. GIT , Tortoise-GIT et intégration IDE

Utilisation directe ou indirecte de GIT



Ajoute des menus contextuels à l'explorateur de fichiers



...



Menu "team" et perspective "GIT"

GIT (en ligne de commande)

O.S. (linux ou windows ou ...)

2. Principales commandes de GIT (en mode local)

Commandes GIT (locales)	Utilités
git init	Initialise un référentiel local git (sous répertoire caché « .git ») au sein d'un projet neuf/originel.
git clone <i>url_referentiel_git</i>	Récupère une copie locale (sous le contrôle de GIT et avec toutes les versions des fichiers) d'un référentiel git existant (souvent distant)
git status	Affiche la liste des fichiers avec des changements (pas encore enregistrés par un commit) et git diff affiche les détails (lignes en + ou -) dans un certain fichier.
git diff <i>fichier</i>	
git add <i>liste_de_fichiers</i>	Ajoute un répertoire ou un fichier dans la liste des éléments qui seront pris en charge par git (lors du prochain commit).
git commit <i>-m message [-a]</i>	Enregistre les derniers fichiers modifiés ou ajoutés dans le référentiel git local (ceux

	préalablement précisés par <i>add</i> et affichés par <i>status</i>) . si option -a tous les fichiers modifiés (ou supprimés) qui étaient déjà pris en charge par git seront enregistrés
git checkout <i>idCommit</i> (ou <i>branche</i>)	Récupère les (dernières ou) versions depuis le référentiel local
git --help git cmde --help	Obtention d'une aide (liste des commandes ou bien aide précise sur une commande)
git log --stat ou git log -p	Affiche l'historique des mises à jour -p : avec détails , --stat : résumé
git branch , git checkout <i>nomBranche</i> , git merge	Travailler (localement et ...) sur des branches
git grep <i>texte</i> <i>a_rechercher</i>	Recherche la liste des fichiers contenant un texte
git tag <i>NomTag</i> <i>IdCommit</i>	Associer un tag parlant(ex: v1.3) à un id de commit .
git tag -l	Visualiser la liste des tags existants
git checkout tags/ <i>NomTag</i>	Récupère la version identifiée par un tag

Exemples :

#initialisation

cd p1; **git init**

#affichage des éléments non enregistrés

cd p1; **git status**

→ affiche:

On branch master

Changes not staged for commit:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

modified: src/f1.txt

modified: src/f3-renamed.txt

no changes added to commit (use "git add" and/or "git commit -a")

commit all already tracked/added :

cd p1

-a pour tous les fichiers listés dans git status

git commit -a -m "my commit message"

#commit all (with all new and deleted) :

cd p1

git add *pom.xml.txt* *src/****git status**

git commit gère tous les fichiers ajoutés (et supprimera de l'index ceux qui

n'existent plus si option -a)

```
git commit -m "my commit message" -a
```

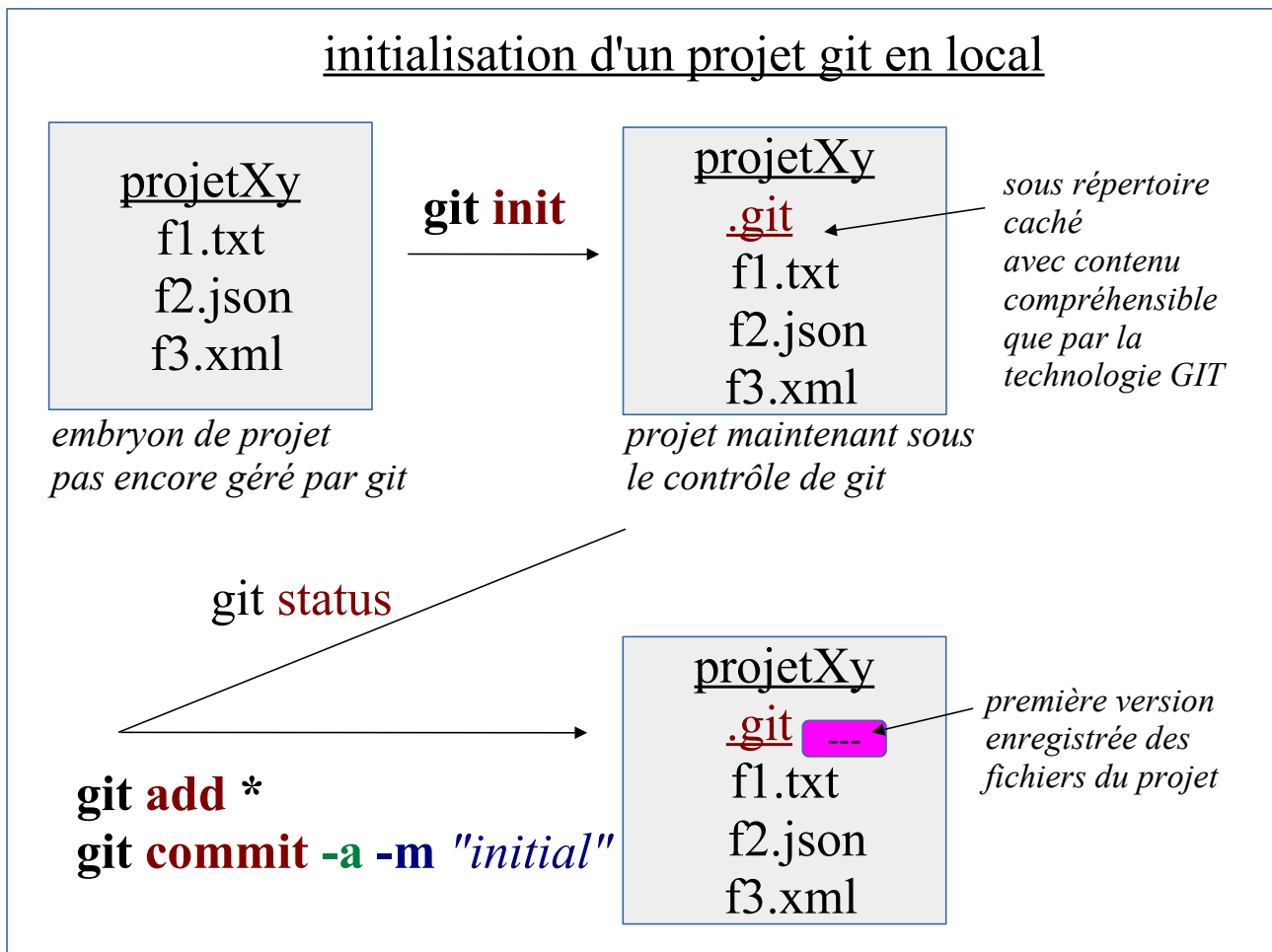
#historique des dernières mises à jour :

```
cd p1; git log -stat
```

---> affiche:

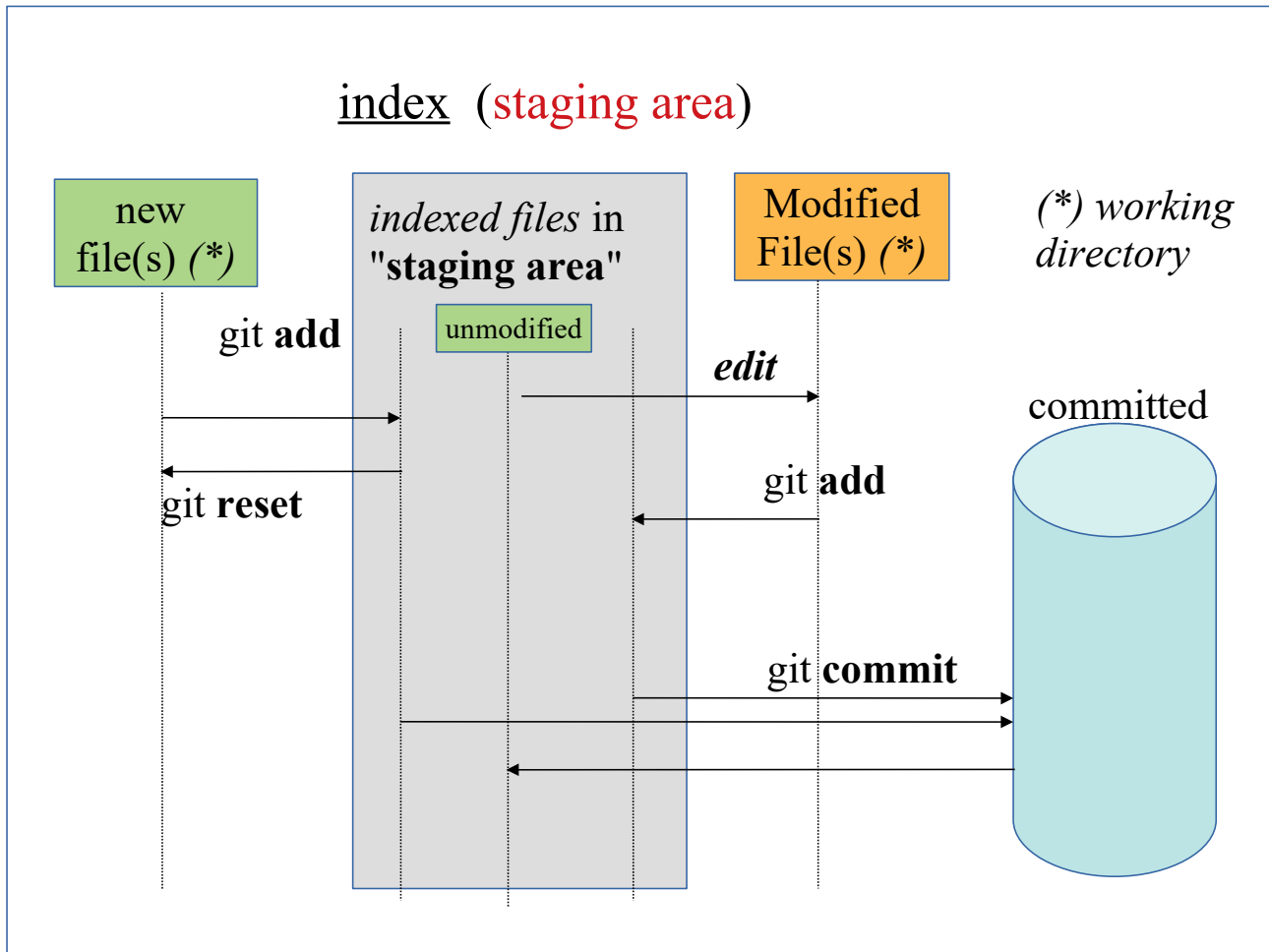
```
commit 93446a0f2194089d83c941a63768f212eb96e0f8
Author: developpeur fou <moi@ici_ou_la.everywhere>
Date:   Wed Dec 12 18:36:40 2012 +0100
    my commit message
    pom.xml.txt      | 2 ++
    src/f1.txt       | 1 +
    src/f3-renamed.txt | 1 +
    src/f4-renamed.txt | 1 +
    src/p/pf2-renamed.txt | 2 ++
    5 files changed, 7 insertions(+)
```

3. initialisation d'un projet GIT (en local)



4. Index (staging area)

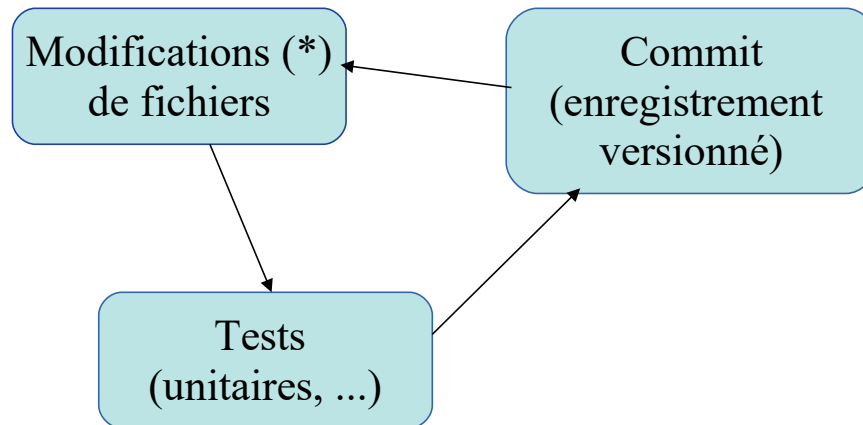
4.1. fichiers indexés par GIT



Si besoin , **git reset f1.txt** permet d'annuler **git add f1.txt**

4.2. cycle des mises à jour

Cycle de mises à jour



(*) Un fichier existant déjà lors du dernier commit et re-modifié depuis fait d'office partie de la "*staging area*" et sera par défaut re-commité. Les commandes `git add ...`, `git rm ...`, `git rm --cached ...` permettent d'ajouter ou retirer des fichiers à commiter ultérieurement avec l'option `-a`.

`git commit --amend -m nouveauMessage` permet si besoin de remplacer le message de commit

5. indispensable .gitignore

Le fichier caché .gitignore (à placer à la racine d'un référentiel git) est indispensable pour préciser la liste des fichiers à ne pas stocker dans le référentiel git (ex : fichiers temporaires , spécifiques à un IDE , fichiers binaires générés ,) .

Exemple de fichier ".gitignore" pour java / maven /eclipse :

.gitignore

```
target/  
*.class  
*/.settings/  
.settings/*  
.settings  
*.jar  
*.war  
*.ear
```

Exemple de fichier ".gitignore" pour npm/javascript :

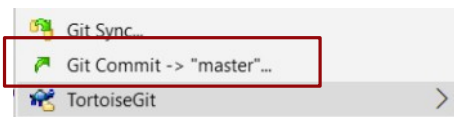
.gitignore

```
node_modules  
node_modules/*  
dist/*  
dist
```

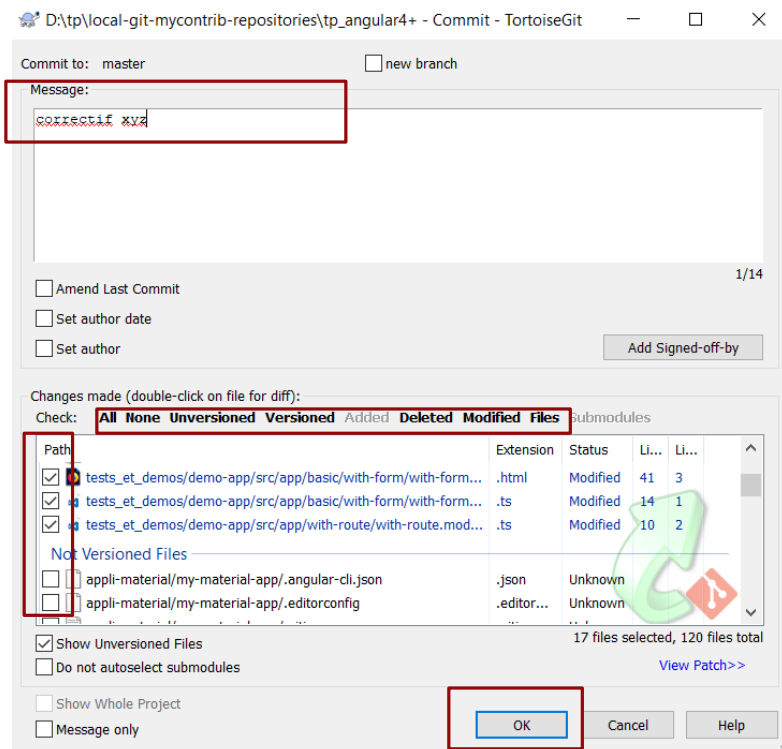
6. Commit et tags

commit via tortoiseGit

dans l'explorateur de fichiers à partir du répertoire du projet déclencher le menu contextuel **commit** --> ...



choix des fichiers à "commiter"



commits et tags



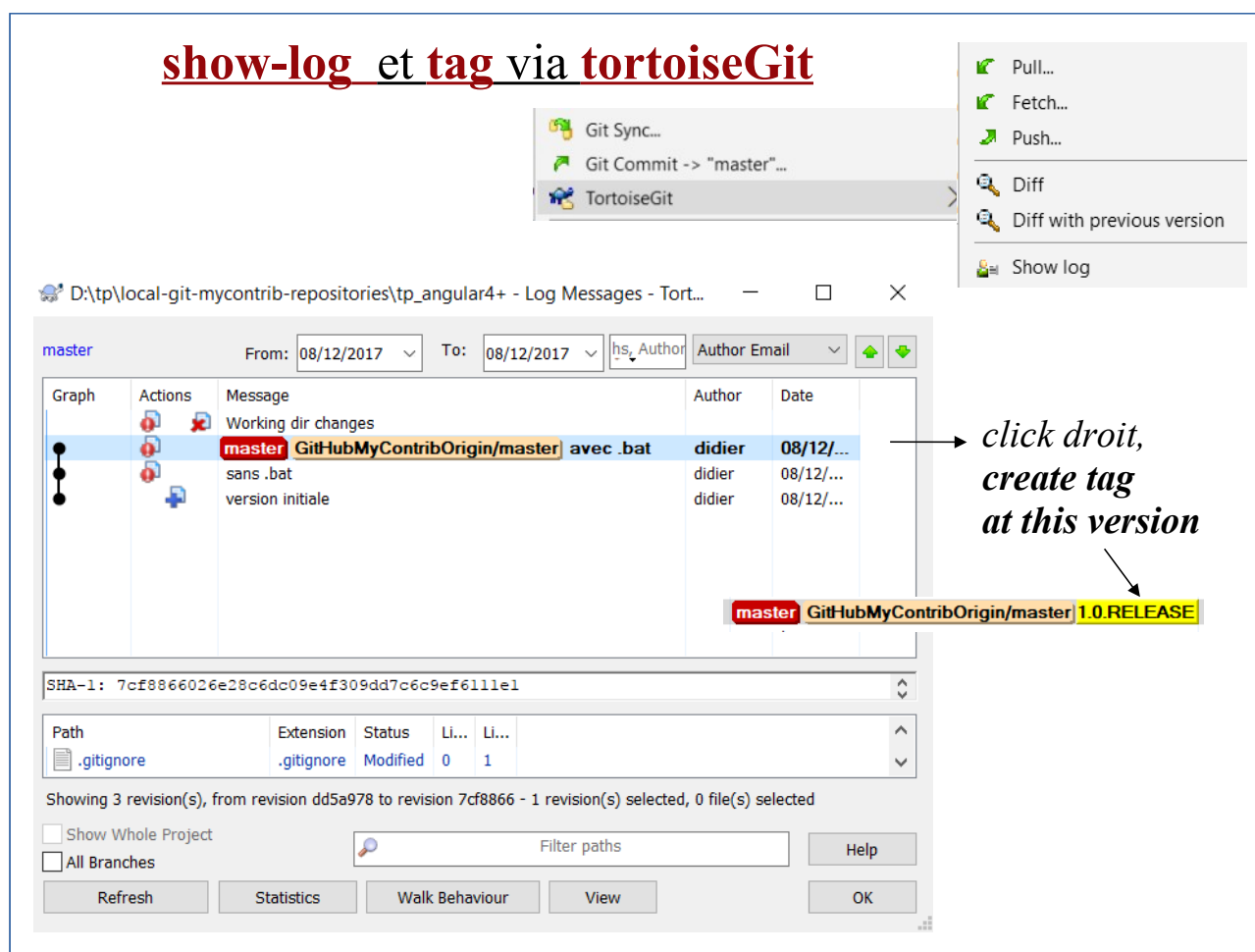
NB :

Il est éventuellement possible de visualiser l'état du projet dans une révision (numéro de commit bien précis) via la commande **checkout** *numero_de_commit_precis* .

Ceci dit , il ne vaut mieux pas éditer et enregistrer des fichiers juste après car on serait alors en mode "detached HEAD" (détaché d'une tête de branche) ce qui pose plein de problème par la suite .

---> pour effectuer des modifications à un niveau antérieur de l'historique , il faudra idéalement créer une nouvelle branche démarrant au niveau d'un commit_bien_precis .

show-log et tag via tortoiseGit



....

7. Commandes de GIT pour le mode distant

Commandes GIT (mode distant)	Utilités
git init --bare	Initialisation d'un nouveau référentiel vide de type «nu» ou «serveur». (à alimenter par un push depuis un projet originel)
git clone --bare url_repo_existant	Idem mais via un clonage d'un référentiel existant
git clone url_repo_sur_serveur.git	Création d'une copie du projet sur un poste de développement (c'est à ce moment qu'est mémorisée l'url du référentiel « serveur » pour les futurs push et pull)
git pull	Rapatrie les dernières mises à jour du serveur distant (de référence) vers le référentiel local. (NB: <i>git pull</i> revient à déclencher les deux sous commandes <i>git fetch</i> et <i>git merge</i>)
git push	Envoie les dernières mises à jour vers le serveur distant (de référence) <u>Attention</u> : le <i>push</i> est irréversible et <i>personne</i>

	<i>ne doit avoir effectué un push depuis votre dernier pull !</i>
...	

Exemples:

#script de création d'un nouveau référentiel GIT (coté serveur) dans /var/scm/git ou ailleurs:

```
mkdir p0.git
cd p0.git
git init --bare
git update-server-info
mv hooks/post-update.sample hooks/post-update

#nb www-data est le groupe de apache2
cd ..
sudo chgrp -R www-data p0.git

# ce repository initial et vide pourra être alimenté par un push depuis un projet "original"
# depuis ce projet original , on pourra lancer git config remote.p0.url http://localhost/git/p0.git
#           puis git push p0 master
echo "fin ?"; read fin
```

ou bien

```
# construira p1.git
git clone --bare file:///home/formation/Bureau/tp/tmp-test-git/original/p1
cd p1.git
git update-server-info
```

#récupération d'une copie du projet sur un poste de développement

```
git clone http://localhost/git/p1.git
```

#pull from serv:

```
cd p1
git pull
```

#push to serv:

```
cd p1
git push
```

8. Gérer plusieurs référentiels distants

Commandes GIT (mode distant)	Utilités
git remote -v	Affiche la liste des origines distantes (URL des référentiels distant)
git remote add originXy url_repoXy	ajoute une origine distante (alias associé à URL)

git push -u originXy	Effectue un push vers l'origine (upstream) précisé (alias associé à l'URL du référentiel distant).
git push --set-upstream originXy master	push en précisant la branche remote à pister (track) ceci est particulièrement utile pour un push initial vers référentiel distant vide (pas encore initialisé)
...	

Exemples :

s_list_remote_git_url.bat

```
git remote -v
pause
```

s_set_git_remote_origin.bat

```
git remote set-url origin Z:\TP\tp_angular1.git
git remote -v
pause
```

s_push_to_remote_origin.bat

```
git push -u origin master
pause
```

s_push_to_github.bat

```
git remote add GitHubMyContribOrigin https://github.com/didier-mycontrib/tp_angular.git
REM didier-mycontrib / gh14.....sm..x / didier@d-defrance.fr
git push -u GitHubMyContribOrigin master
pause
```

commit_and_push.bat

```
cd /d "%~dp0"
git add *
git commit -a -m "nouvelle version"
git push -u GitHubMyContribOrigin master
pause
```

Bien qu'il soit possible de placer username et password dans l'URL ceci est une mauvaise pratique

d'un point de vue sécurité .

Il vaut mieux configurer GIT pour qu'il retienne le mot de passe saisi durant une certaine période (exemple : 3600 secondes = 1 heure) :

```
git config [ --global ] credential.helper 'cache --timeout=3600'
```

Et si nécessaire (pour ancienne version de git) :

```
git credential-cache exit
```

pour que GIT oublie l'ancien mot de passe et que l'on puisse de ré-authentifier .

ou pour version récente de git :

```
git config --global --unset credential.helper
```

et/ou

```
git config --system --unset credential.helper
```

pour désactiver (temporairement ou pas) le "credential.helper" mémorisant les username/password .

Sur une machine windows d'entreprise, on peut également choisir "manager" comme type de "credential.helper" via la commande :

```
git config --global credential.helper manager ou wincred
```

puis en lançant la commande suivante pour vérifier :

```
git config --system --list
```

Ceci permet de configurer les informations d'authentification via la partie "Comptes d'utilisateurs → Gestionnaire des informations d'identification → Gérer les informations d'identification Windows " du panneau de configuration de windows d'entreprise (pas "windows famille") .

NB: l'exemple configuration ci dessus doit être adaptée au cas par cas (selon version de git , selon OS linux ou windows , selon github ou gitlab, selon l'année ,)


Token pour opérations git distantes sans mot de passe

De manière à accéder en lecture/écriture à un référentiel distant géré de façon récente par GitHub ou bien GitLab on a besoin de générer et enregistrer un "**token développeur**" (via la page d'administration d'un compte GitHub ou GitLab) .

Settings / Developer settings

 GitHub Apps

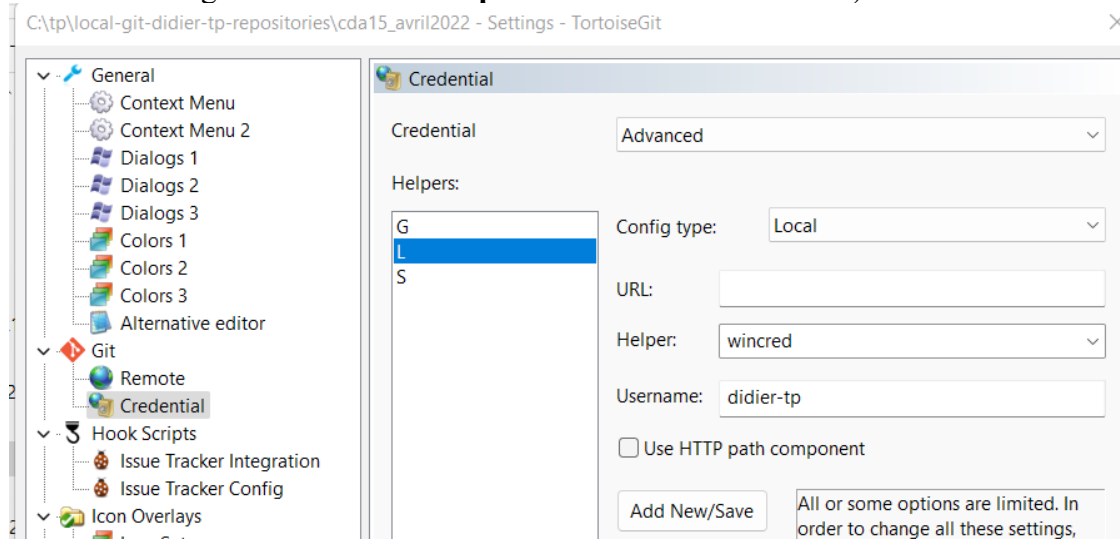
 OAuth Apps

 Personal access tokens

Une fois généré et mémorisé dans un fichier temporaire , ce "token développeur" (ressemblant à `gtNVm7wH8P9ohp_u5jIVxzs2AWx6zT2E5p4AYlsF` et valable pour une durée à choisir telle que 30 jours par exemple) devra être enregistré auprès d'un "**credential.helper**" du poste du développeur (ex : linux ou windows) et sera utilisé en tant qu'élément d'authentification .

Cette opération un peut technique peut éventuellement être effectuée graphiquement via une interface graphique telle que tortoise git ou autre .

Avec tortoise git (git settings , partie git/credentials ,
add local config with **credential-helper=wincred** and **username**)



et spécifier le token via un copier/coller lorsque ce sera demandé (lors d'un git push par exemple). Ce token (lié à votre username) sera ensuite automatiquement conservé lors des futures requêtes .

9. initialiser git en mode distant

initialiser git en mode "remote"

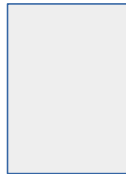
1 *préparation d'un projet*

git en mode local (java ou ...)

git init , .gitignore , ... , commit



dev1



3 *préciser url et premier push :*

git remote add origin url_repo

git push --set-upstream origin master

2 *préparation*

du référentiel

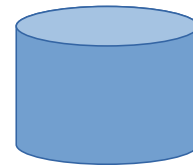
git distant vide

(git init --bare)

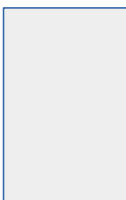
ou équivalent via

console github

ou autre



dev2,



4 *clonage(s) (avec "origin" fixée automatiquement) :*

git clone url_repo

...

Ensuite , git pull et git push de chaque côté .

10. pistage (track) entre branches locales et distantes

pistage (tracking) de branche

- Lorsqu'une branche locale est paramétrée pour pister (**track**) une branche distante, celles ci pourront ensuite être synchronisées via git push et git pull (et autres).
- Pour initialiser ce processus, il faut la première fois lancer la commande **git push** avec l'option **--set-upstream origin**
- Exemple: **git push --set-upstream origin m-dev**
[new branch] m-dev -> m-dev
Branch m-dev set up to track remote branch m-dev from origin.
git branch -vv
** m-dev 9f0d962 [origin/m-dev] ...*
master d41dc86 [origin/master] ...

pistage (tracking) de branche (sur clones)

- **git branch -a** permet de visualiser toutes les branches dont les branches distantes existantes :

* *master*

remotes/origin/HEAD -> origin/master

remotes/origin/m-dev

remotes/origin/master

- Au niveau d'un clone secondaire , pour initialiser une branche locale pistant une branche distante existante on peut lancer la commande suivante :

git checkout -b m-dev origin/m-dev

Branch m-dev set up to track remote branch m-dev from origin.

Switched to a new branch 'm-dev'

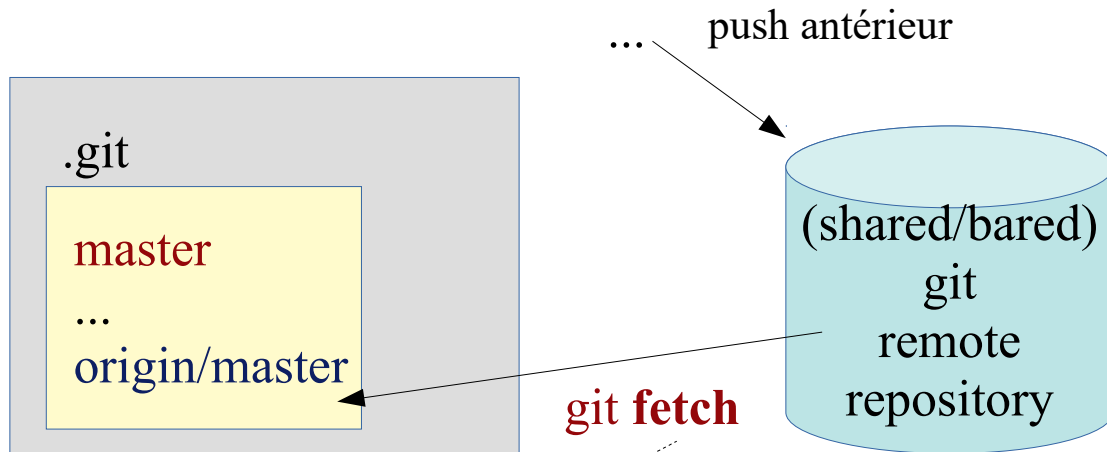
git branch -vv

* *m-dev 439df8e [origin/m-dev] ...*

master d41dc86 [origin/master] ...

11. Git fetch et git pull

git fetch



② on peut ensuite , en étant placé sur la branche locale `master` voir les différences via `git branch -vv` et via `git diff origin/master`

① rapatrie (réplique) les mises à jour du référentiel distant dans la branche `origin/master` du référentiel local. Rien ne change sur la branche locale "master" ni dans le répertoire de travail (du projet).

git pull

- En étant placé sur la branche master pistant la branche distante origin/master, **git pull** est équivalent à **git fetch** suivit de **git merge origin/master**
- il faut quelquefois résoudre certains conflits
- Les mises à jours distantes sont alors vues / répercutées dans le répertoire de travail (du projet)

NB: Dans certains cas , on peut avoir envie de mettre de coté rapidement certaines opérations temporaires et expérimentales sur la branche courante pour y revenir après sachant qu'une opération plus importante est nécessaire (ex : rapatrier des modifications distantes via **git pull**) . On peut éventuellement utiliser la commande **git stash** pour cela .

NB: après un "git stash" , les modifications temporaires (non commitées) sont stockées dans un espace local de type "brouillon temporaire réactivable" et ne seront jamais propoagées lors d'un git push .

Si l'on souhaite réactiver les modifs temporaires enregistrée via git stash , on peut lancer la commande **git stash pop**

L'option **-a** existe sur **git stash** de la même façon que sur git commit .

12. Plugin eclipse pour GIT (EGIT)

Le plugin eclipse pour GIT s'appelle **EGIT**.

12.1. Actions basiques (commit , checkout , pull , push)

→ Se laisser guider par la perspective "GIT" et via le menu "Team"

12.2. Résolution de conflits

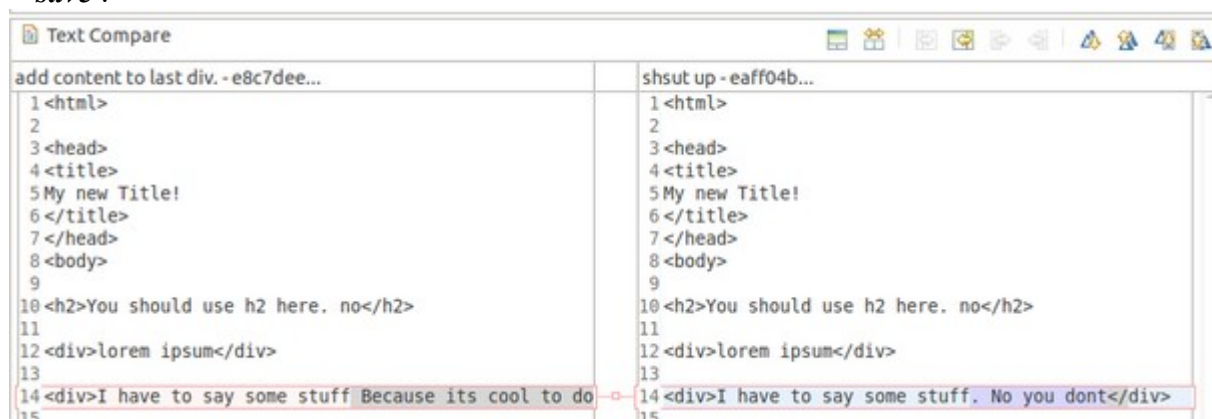
- 1) déclencher "**Team / pull**" pour récupérer (en tâche de fond) la dernière version (partagée / de l'équipe). Le plugin EGIT va alors tenter un "**auto-merge**" ("git fetch FETCH_HEAD" suivi par "git merge").
- 2) En cas de conflit (non résoluble automatiquement), les fichiers en conflit seront marqués d'un point rouge.

```
<<<<<< HEAD
<div>I have to say some stuff Because its cool to do so.</div>
=====
<div>I have to say some stuff. No you dont</div>
>>>>>> branch 'master' of /var/data/merge-issue.git
```

Sur chacun des fichiers en conflit, on pourra déclencher le *menu contextuel*

"Team / Merge tool". (laisser par défaut la configuration de "Merge Tool" : use HEAD).

- 3) **Saisir, changer ou supprimer alors au moins un caractère dans la zone locale (à gauche) + save :**



... au cas par cas

- 4) Déclencher le menu contextuel "**Team / add to index**" pour ajouter le fichier modifié dans la liste de ceux à gérer (staging).
- 5) Effectuer un "**Team / commit**" local.
- 6) Effectuer un "**Team / push to upstream ...**" pour mettre à jour le référentiel distant/partagé.

IV - Rôle du « builder » (ant, maven) et scripts

1. Constructions et déclenchement des tests (ic)

NB : Un logiciel d'intégration continue se paramètre généralement avec une IHM Web. Cependant le déclenchement des "builds" avec des tests s'effectue de façon complètement automatisée (par scripts , sans aucune interaction d'un utilisateur) .

1.1. Reconstructions (compilation, assemblage) du code

La première étape à effectuer consiste à recompiler tout le code (avec une version bien contrôlée du compilateur java) pour s'assurer que tous les modules sont bien compatibles avec une certaine version du jdk (ex : 8 ou 11 ou 17).

D'autre part, un logiciel sérieux est généralement constitué de plusieurs modules complémentaires devant rester cohérents.

Le logiciel d'intégration continue va donc alors permettre de tester si un module client fraîchement reconstruit avec la dernière (ou ...) version du code s'assemble bien avec les autres modules "backends" également reconstruits depuis la dernière version du code déposé dans SVN ou GIT.

Tout éventuel problème d'incompatibilité fonctionnelle (décalage de version une des interfaces ou dans une des implémentations) sera alors détecté et remonté vers les développeurs.

1.2. Tests automatisés , scripts d'intégration

L'autre tâche importante du logiciel d'intégration continue consiste à lancer toute une batterie de tests fonctionnels pour s'assurer que la dernière version construite fonctionne bien.

Dans le monde java , les "builds" et le lancement des tests s'effectuent généralement via les technologies "ant" ou "maven" ou "gradle".

L'utilisation de "ant" nécessite une configuration assez longue devant expliciter tout un tas de chemin (classpath ,) et d'instructions (ordre de compilation , de packaging ,) .

A l'inverse, en utilisant "maven" , la configuration est beaucoup plus simple (car basée sur des conventions structurelles et des ordres/instructions implicites).

En pratique, la plupart des solutions d'intégration continue s'appuient aujourd'hui sur maven (ou bien sur gradle).

1.3. Bien utiliser la notion de « profils mavens »

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 8 ou 11 ou 17 ou ...)
- les ressources (url et type de base de données , annuaire ldap , ...)
- les jeux de données à charger pour les tests (fichiers xml pour DbUnit ,)
- ..

2. 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 développeurs fonctionnent</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 Hudson/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 .

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)

Idée : éventuel blog (ou ...) pour que les développeurs puissent déposer des billets/messages/commentaires sur les statuts des builds .

Lors de la construction des builds, une compilation complète ("clean build") est quelquefois préférable à une compilation incrémentale de façon à mieux détecter certains problèmes.

3. Exemple de script ant pour de l'intégration continue

Bien que la plupart des projets java soient maintenant structurés autour de la technologie maven , il est néanmoins possible d'utiliser ANT au niveau de la partie "script" de construction pour les besoins de l'intégration continue :

build.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="project" default="default">
  <description> petit calculateur java </description>

  <target name="default" depends="compile,test,package" description="compile and test">
  </target>

  <target name="clean" description="clean ant-build directory and my-java-app2.jar">
    <delete dir="ant-build" />
    <delete file="my-java-app2.jar"/>
  </target>

  <!------- target: compile ----->
  <target name="compile" description="compile source de src dans ant-build">
    <mkdir dir="ant-build"/>
    <javac srcdir="src" destdir="ant-build">
      <classpath location="lib/junit-3.8.2.jar"/>
    </javac>
  </target>

  <!-- - - - target: test (unitaires) / junit3 test suite ----->
  <target name="test" description="tests unitaires junit3">
    <junit printsummary="withOutAndErr" showoutput="true"
      haltonerror="true" haltonfailure="true">
      <classpath>
        <pathelement location="lib/junit-3.8.2.jar"/>
        <pathelement location="ant-build"/>
      </classpath>
      <test name="com.mycompany.myapp.test.AllTests" />
    </junit>
  </target>

  <!------- target: package (construit my-java-app2.jar sans les tests) ----->
  <target name="package" description="package appli sous forme de .jar">
    <jar destfile="my-java-app2.jar">
      <fileset dir="ant-build">
        <exclude name="**/*Test.class"/>
        <exclude name="**/test/**"/>
      </fileset>
    </jar>
  </target>
</project>
```

V - Essentiel Maven pour intégration continue

1. Présentation de Maven

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

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

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

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

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

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

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

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

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

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

+

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

2. Principales fonctionnalités de maven

Maven (principales fonctionnalités)

- **Centré sur la notion de projet** (api java, module applicatif)
--> toutes les caractéristiques d'un projet sont déclarées dans un fichier "**pom.xml**" (Project **O**bject **M**odel).
- **Configuration "déclarative"** (basée sur des conventions) plutôt qu'explicite. Pas de commandes et chemins précis à renseigner (contrairement à ANT).
- **Gestion distribuée** (sur le web) des **dépendances** inter-projets.
---> identification et **téléchargement automatique des ".jar"** nécessaires selon les API déclarées en dépendances .
- Gère toutes les phases (compile/build , tests unitaires , packaging , stockage dans le référentiel , éventuel lien avec SVN, ...) .

3. Intérêts de Maven

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

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

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

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

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

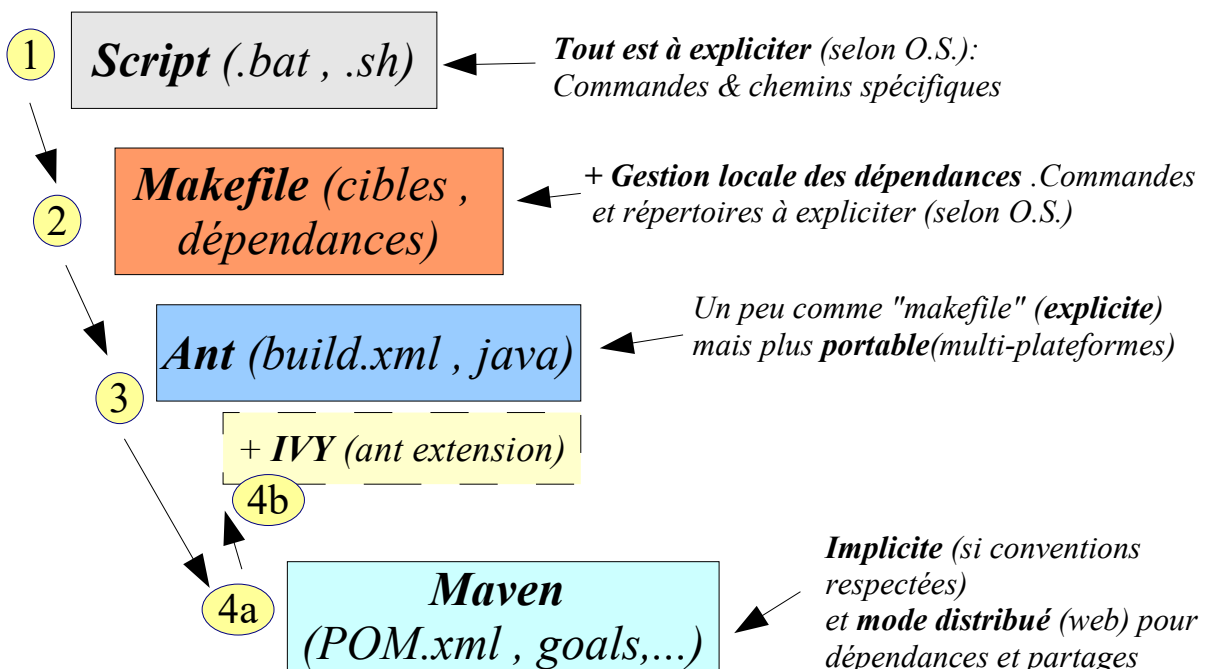
NB: il existe maintenant une extension pour ANT appelée "IVY" qui gère également les dépendances de librairies (".jar") , la syntaxe des fichiers de configuration d'IVY est cependant légèrement différente de celle de "maven". Maven est plus orienté "projet" et va plus loin.

4. Evolutions, versions

Versions de maven

- *Maven 1* (version assez ancienne , aujourd'hui obsolète)
- **Maven 2** (beaucoup d'améliorations , changements en profondeur)
 - > version mature (avec pleins de plugins disponibles)
 - > plugin eclipse "m2e" maintenant au point.
- **Maven 3** (depuis début 2011 , dans la continuité de la V2)
 - > même configuration que la V2 (syntaxe inchangée)
 - > restructuration interne permettant d'obtenir une **nette amélioration des performances**.
 - > quelques ajouts (parallélisme , ...)

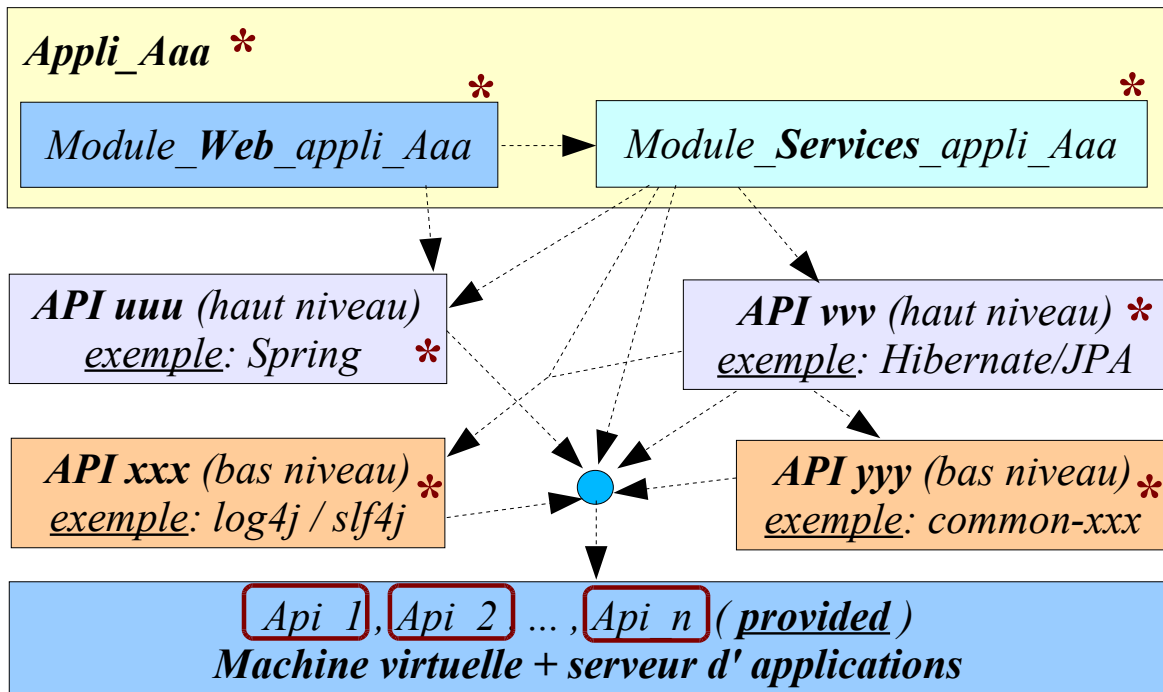
évolutions (du script à maven)



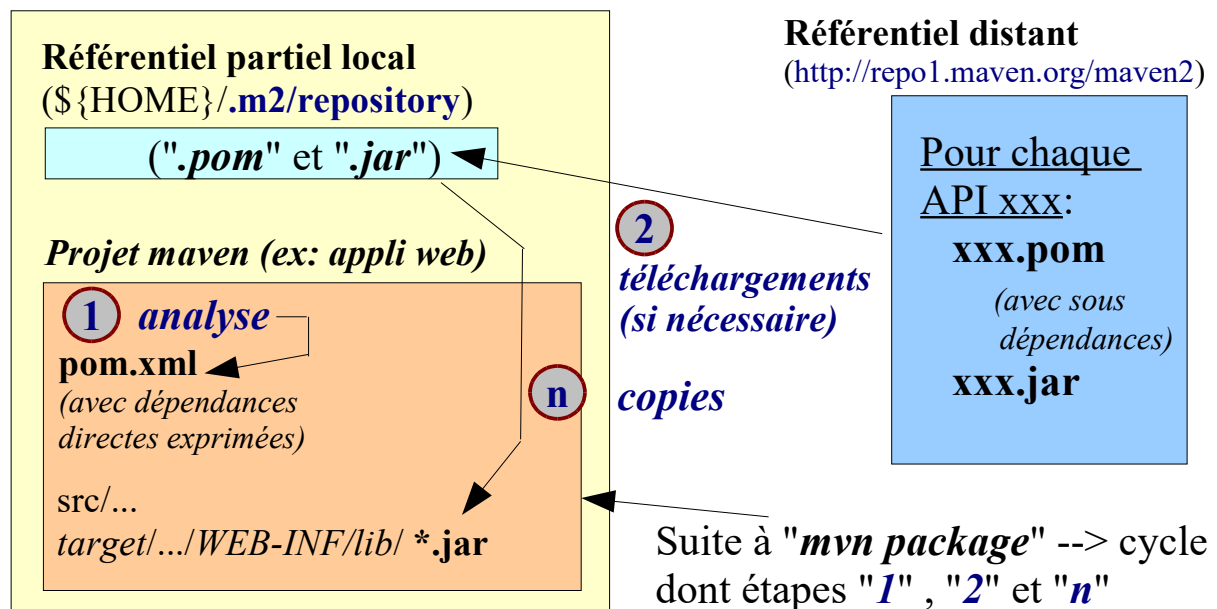
5. Fonctionnement de maven

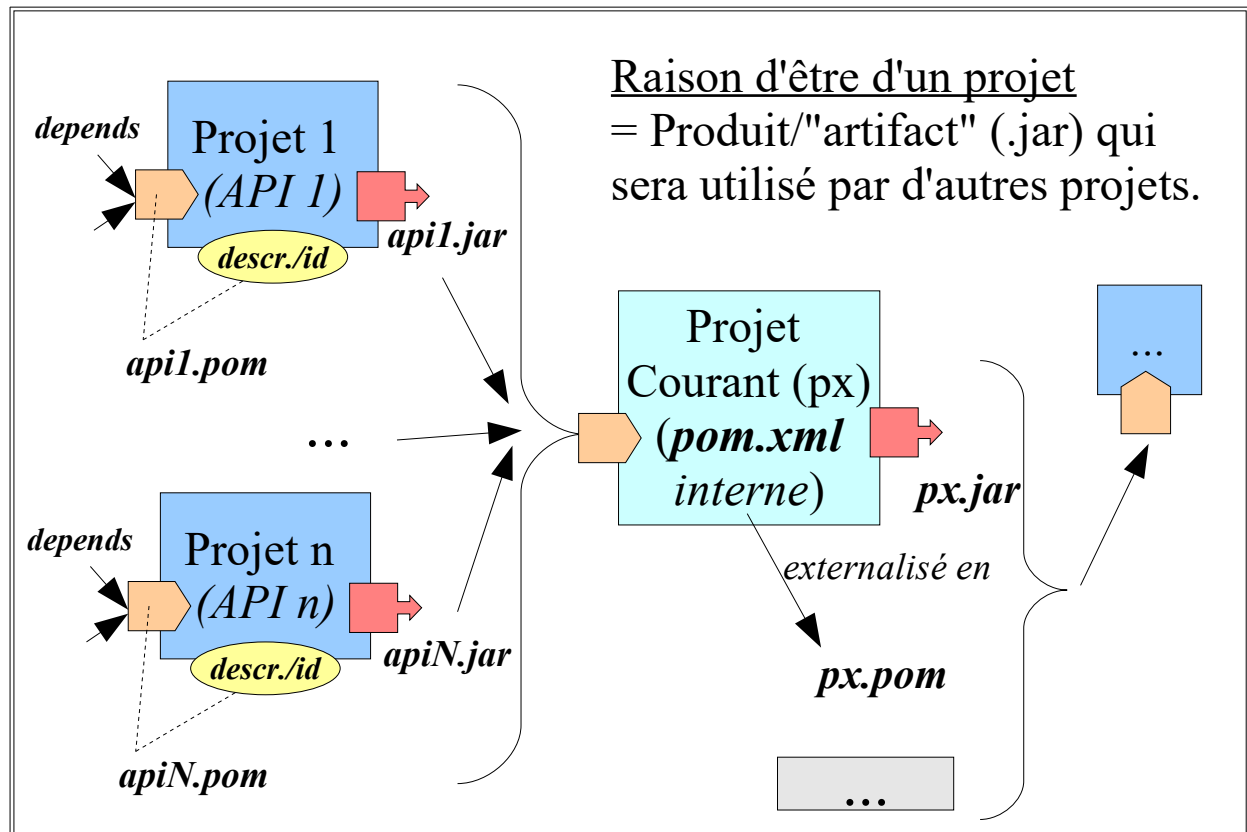
Exemple (classique) de dépendances

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



Téléchargement automatique des librairies nécessaires (avec prise en compte des dépendances indirectes)

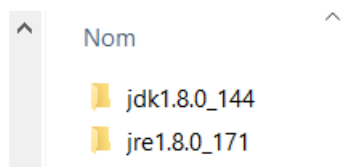




6. Variables d'environnement à bien régler

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

s (C:) > Programmes > Java



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

Modifier la variable système

Nom de la variable :

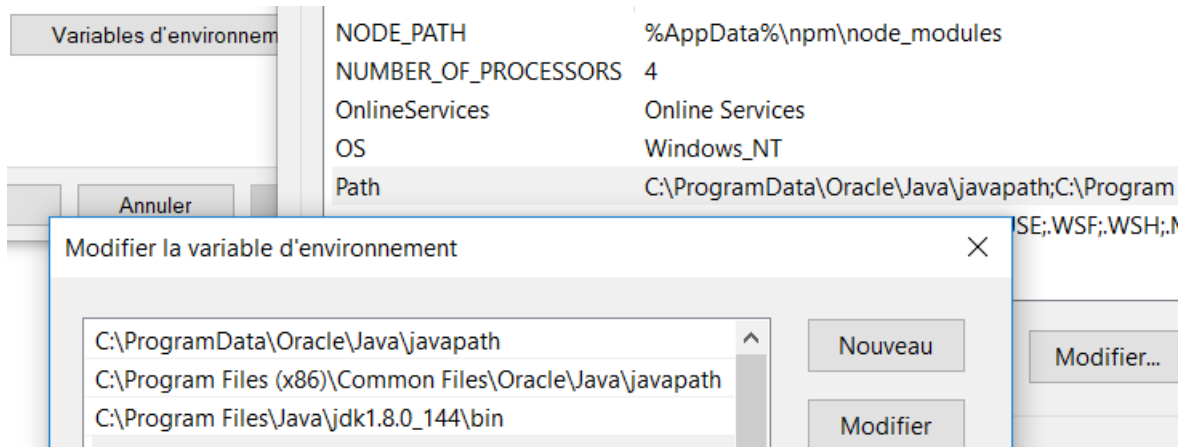
Valeur de la variable :

Parcourir le répertoire.. Parcourir le fichier.. OK Annuler

D'autre part, une fois que maven.3.5....zip aura été recopié et "dezipé/décompressé" sur le poste du développeur, il faudra que le sous répertoire **bin** de **apache-maven-3.5...** soit **présent** dans le

PATH pour que la commande "**mvn**" soit trouvée dans le cadre d'un déclenchement en ligne de commande (depuis une fenêtre "CMD" par exemple sous windows").

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



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

7. Mise en oeuvre de Maven

Utilisation pratique de maven

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

1. télécharger (<http://maven.apache.org/download.html>) et installer le produit "maven" en "dézipant" le contenu de "**apache-maven-2-ou-3.zip**" et en ajoutant c:\...\maven...\bin dans le path du système (os).

Tester l'installation via la commande "**mvn --version**"

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

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

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

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

Ligne de commande pour créer un nouveau projet "maven"

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

Arborescence de répertoires et fichiers créée :

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

pom.xml

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

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

  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

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

Ligne de commande pour construire le projet:

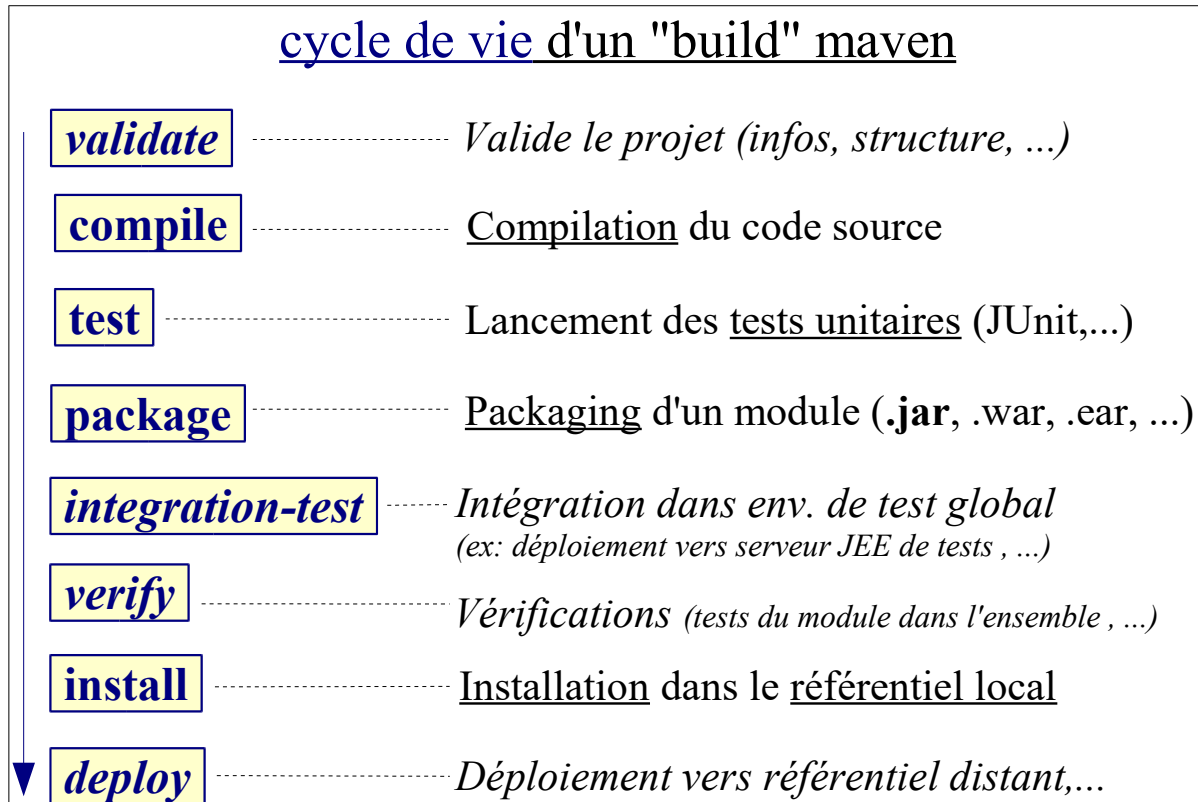
```
mvn package
```

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

Buts (goals) et phases

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

Principaux buts (goals):



Phases du cycle de construction

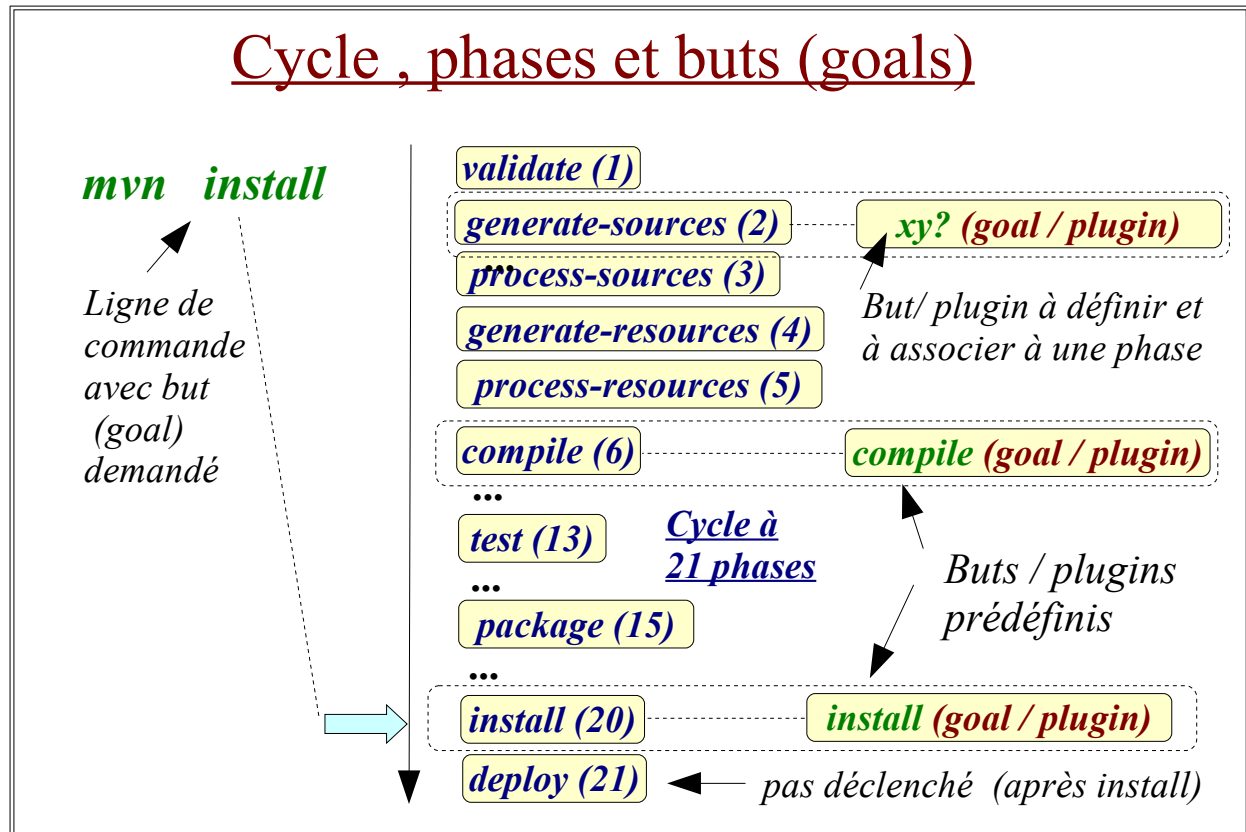
Lorsque l'on déclenche une *ligne de commande* "mvn <goal>" (ex: mvn install) , on *demande explicitement à atteindre un but* .

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

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

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

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

lien entre phases et buts (goals):Phases du cycle de construction par défaut:

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

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

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

NB:

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

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

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

(*) selon version de maven

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

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

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

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

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

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

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

Cycle spécifique au "clean":

Clean Lifecycle

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

Autres buts:
selon plugins (et documentation associée)

8. Quelques options importantes

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

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

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

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

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

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

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

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

9. POM (Project Object Model)

9.1. groupId & artifactId

groupId & artifactId (quelques exemples)

↑
*Éditeur,
organisation/entreprise
(+éventuelle sous branche)*

↑
Produit (application, sous module, Api)

org.apache.cxf ————— cxf-api , cxf-rt-core , ...

org.springframework ——— spring-core , spring-orm , ...

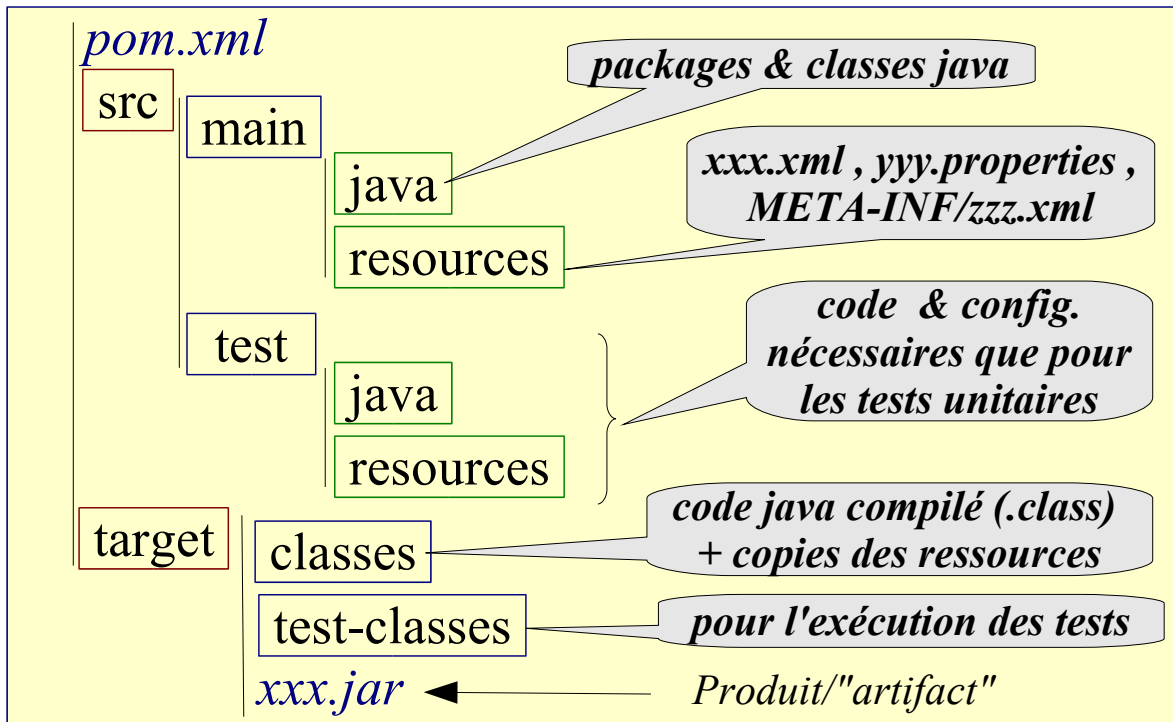
org.hibernate ————— hibernate-core, hib...-annotations, ...

javax.persistence ——— persistence-api (JPA)

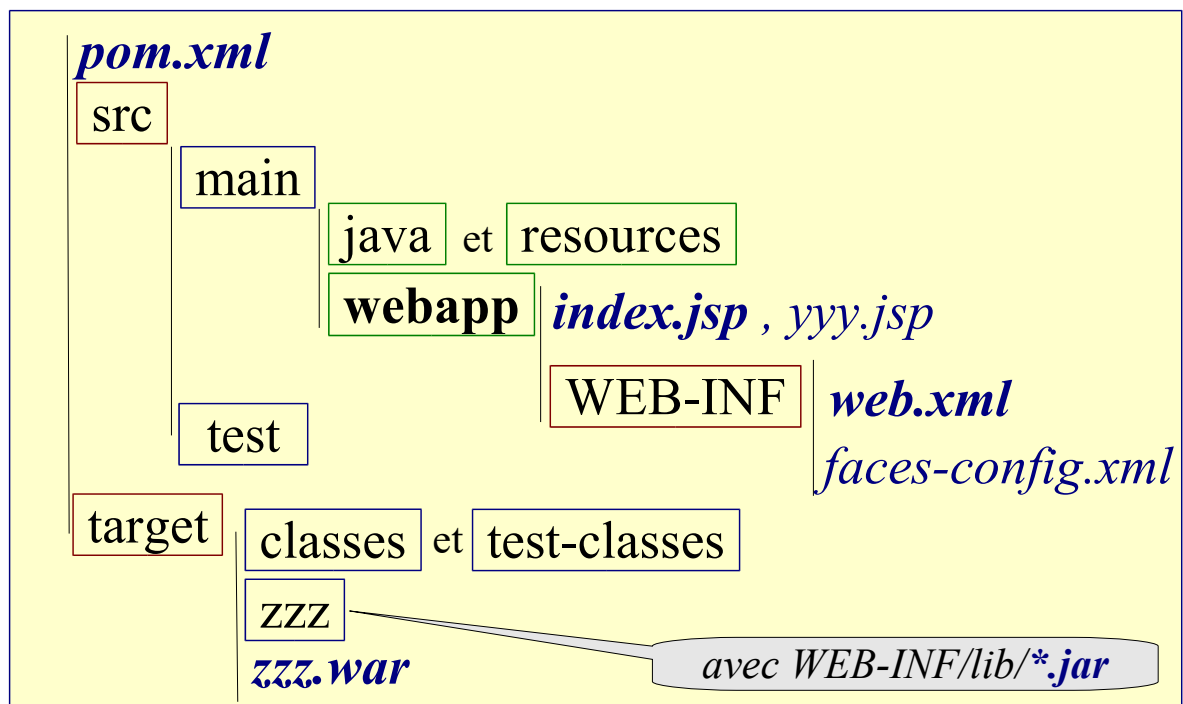
javax.servlet ————— servlet-api

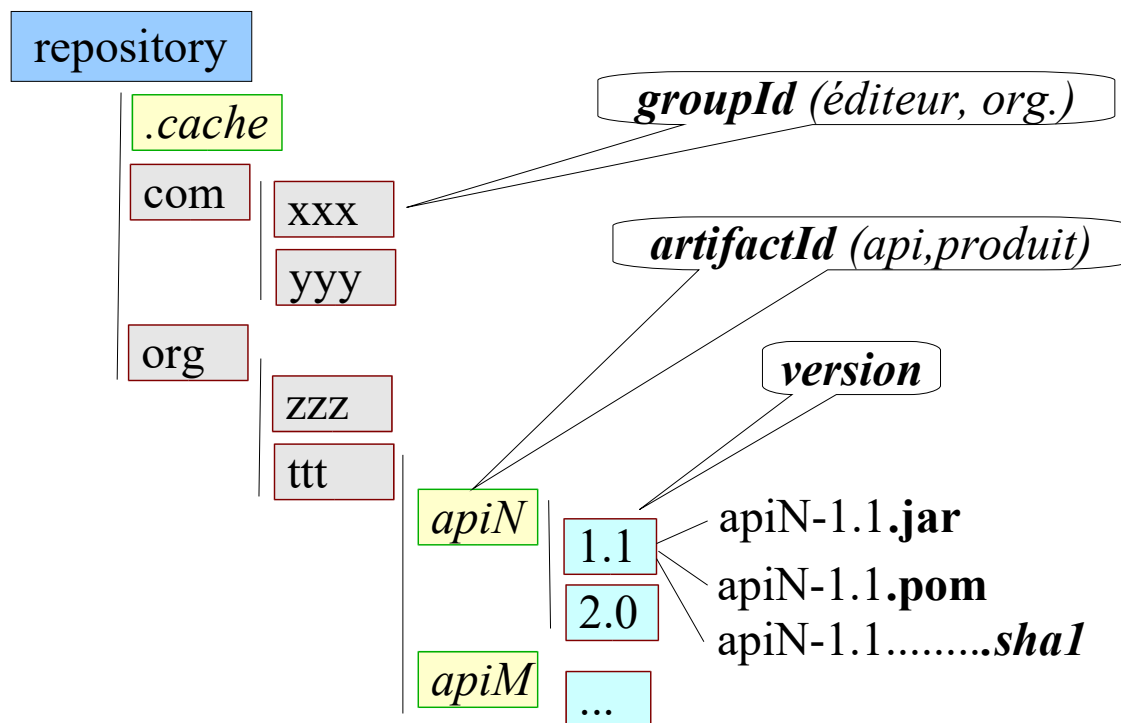
9.2. Arborescences conventionnelles

Structure (quasi-imposée) par conventions "maven"



Structure supplémentaire pour module "java/web"



Structure d'un référentiel "maven" (local ".m2" ou distant)**9.3. portées ("scope") des dépendances:**Principaux types de dépendances "maven" (scope)

- . **compile** (par défaut)
 - > *nécessaire pour l'exécution et la compilation* (dépendance directe puis transitive) [diffusé dans tous les "classpath"].
- . **runtime**
 - > *nécessaire à l'exécution* (dépendance indirecte **transitive**)
- . **provided**
 - > *nécessaire à la compilation* mais **fourni par l'environnement d'exécution (JVM + Serveur JEE)** [diffusé uniquement dans les "classpath" de compilation et de test, dépendance non transitive]
- . **test**
 - > uniquement nécessaire pour les **tests** (ex: *spring-test.jar* , *junit4.jar*)

Diffusé dans quel(s) "classpath" ?

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

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

10. Structure & syntaxes (pom.xml)

Fichier "POM" (éléments essentiels)

servlet-api-2.3.pom (exemple)

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

← Version du modèle
interne de maven

biblio-web-....pom

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion> <parent> ... </parent>
  <groupId>tp</groupId> <artifactId>biblio-web</artifactId>
  <packaging>war</packaging> <version>0.0.1-SNAPSHOT</version>
  <name>biblio-web JEE5 Webapp</name>
  .... <dependencies> ... </dependencies> <build>....</build>
</project>
```

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

```

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

```

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

```

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

```

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

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

Mise au point des dépendances (partie 3)

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

Fichier "POM" (partie "build")

```

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

```

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

```

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

```

Propriétés classiques :

```

<properties>
  <failOnMissingWebXml>false</failOnMissingWebXml>
  <java.version>11</java.version>
  <spring.version>5.3.21</spring.version>
  <junit.jupiter.version>5.8.1</junit.jupiter.version>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <!-- windows/preferences/general/workspace / UTF8 avec eclipse coherent -->
</properties>

```

et

```

<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>${junit.jupiter.version}</version> <scope>test</scope>
</dependency>

```

et

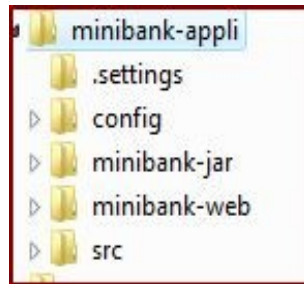
```

  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.10.1</version>
    <configuration>
      <source>${java.version}</source>
      <target>${java.version}</target>
    </configuration>
  </plugin>

```

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

Mode "multi-projets" [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>...
  </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>
```

Mode "multi-modules" (suite)

Arborescence globale conseillée:

my-global-app

pom.xml

minibank-jar (module de services)

...

pom.xml

mywebapp

pom.xml

...

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

Principal intérêt du mode multi-module :

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

pom.xml (de niveau projet global)

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-global-app</artifactId>
  <packaging>pom</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>....</name>
  <modules>
    <module>services</module> <!-- ou <module>../services</module> -->
    <module>mywebapp</module> <!-- ou <module>../mywebapp</module> -->
  </modules>
</project>
```

....

pom.xml (de niveau sous projet / module)

```

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

```

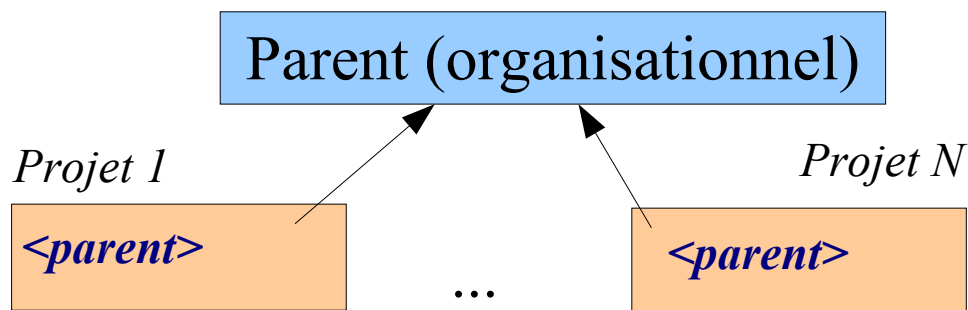
12. Héritage entre projets "maven" (<parent>)

Héritage "organisationnel" au niveau de "maven"

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

Une certaine partie de la configuration du projet parent est ainsi héritée (sans devoir être répétée).

--> intérêt du projet parent: **factoriser** une **configuration commune** entre différents projets "fils".

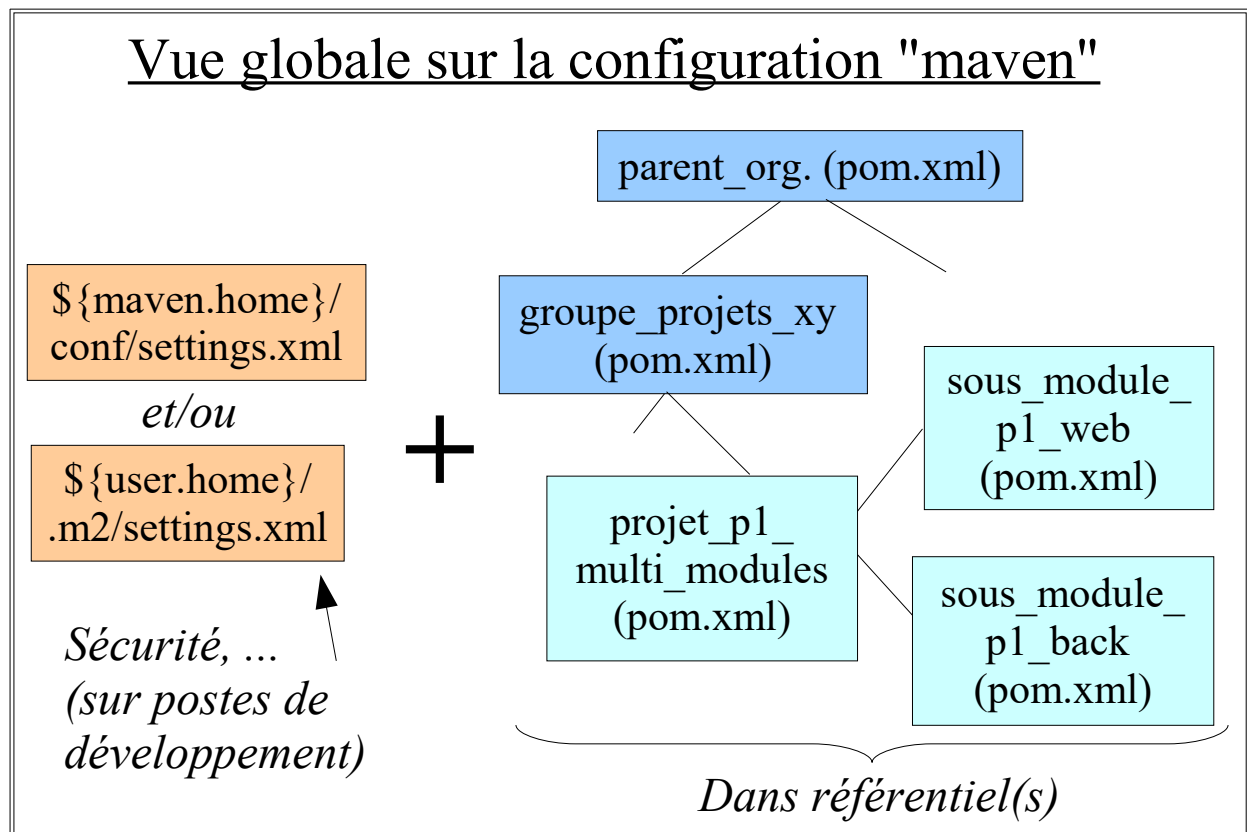


Éléments couramment hérités (pom.xml)

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

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

Conseil: plusieurs niveaux de parents (parent intermédiaire).

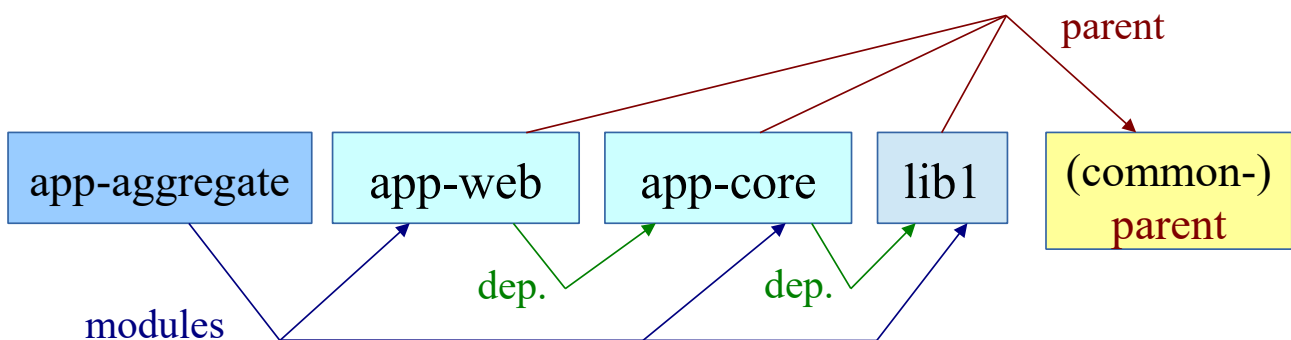


Notion de projet "aggregate" (depuis maven 3)

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

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

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



app-aggregate/pom.xml

```

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

```

app-core/pom.xml

```

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

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

```

app-web/pom.xml

```

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

```


13. Archetypes

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

- Un "*archetype*" est un *modèle (template) de configuration de projet* qui est accessible depuis un référentiel "maven" et qui permet de rapidement définir un nouveau fichier "pom.xml" sans devoir tout préciser en partant de "zéro".
--> ce "*point de départ*" est évidemment à personnaliser au cas par cas selon les spécificités de chaque projet.
- Quelques exemples ("archetypes" / "projet type"):
 - * "j2ee-web"
 - * "jee5-..."
 - * "java5"
 - * "spring3+jpa2+jsf2+cxfr_pour_tc6" *à mettre au point*
 - * ...

14. 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 , ...) .

15. Création d'un nouvel archetype maven

Archetype (structure)

```

pom.xml
src
  --- main
    --- resources
      |-- META-INF
      |   --- maven
      |       --- archetype.xml
    --- archetype-resources
      |-- pom.xml
      --- src
        |-- main
        |   --- java
        |       --- App.java
        --- test
          --- java
              --- AppTest.java
  
```

Fichier listant tous les éléments qui seront initialement générés (par copies de prototypes)

Fichier pom.xml qui sera ultérieurement recopié (et adapté) dans tous les futurs projets "maven" qui seront basés sur cet archetype

Packaging: jar

15.1. 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 .

16. Tests unitaires avec maven

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

src/test/java

Test Junit 4:

```
package package com.mycompany.app1;

import org.junit.Assert;

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

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

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

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

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

Test Junit 5:

```

package package com.mycompany.app1;

import org.junit.jupiter.api.Assertions;

import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.BeforeEach;

/**
 * Unit test for simple Calculateur. (JUnit 5 with annotations)
 */
public class CalculateurTest
{
    private Calculateur c;

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

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

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

```

16.2. Lancement des tests unitaires

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

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

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

17. 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

17.1. 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*]

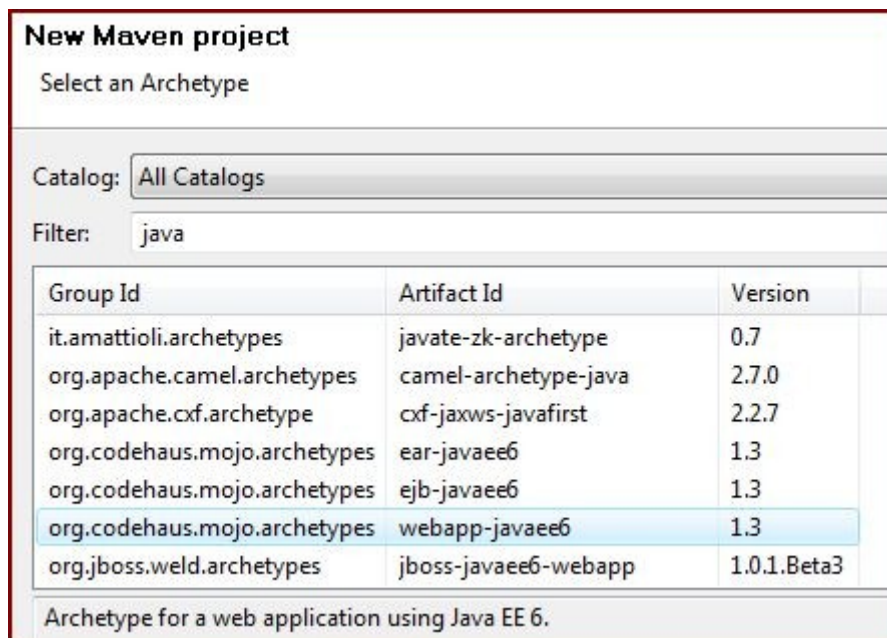
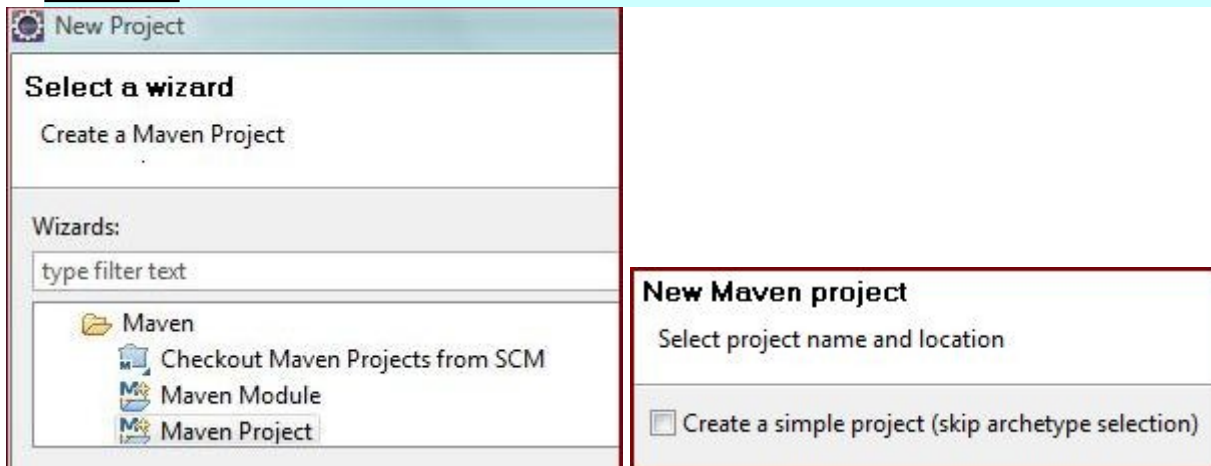
+ nouveau projet "maven"

+ éditer manuellement le fichier "pom.xml"

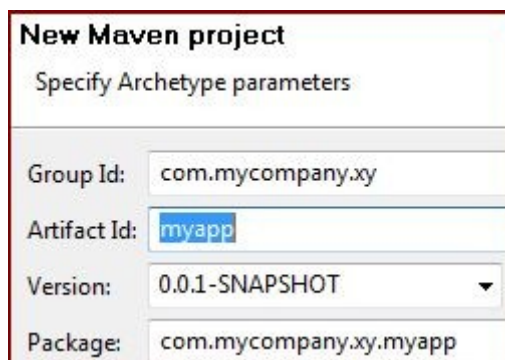
+ éditer le code au bon endroit (dans src/main/java,)

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

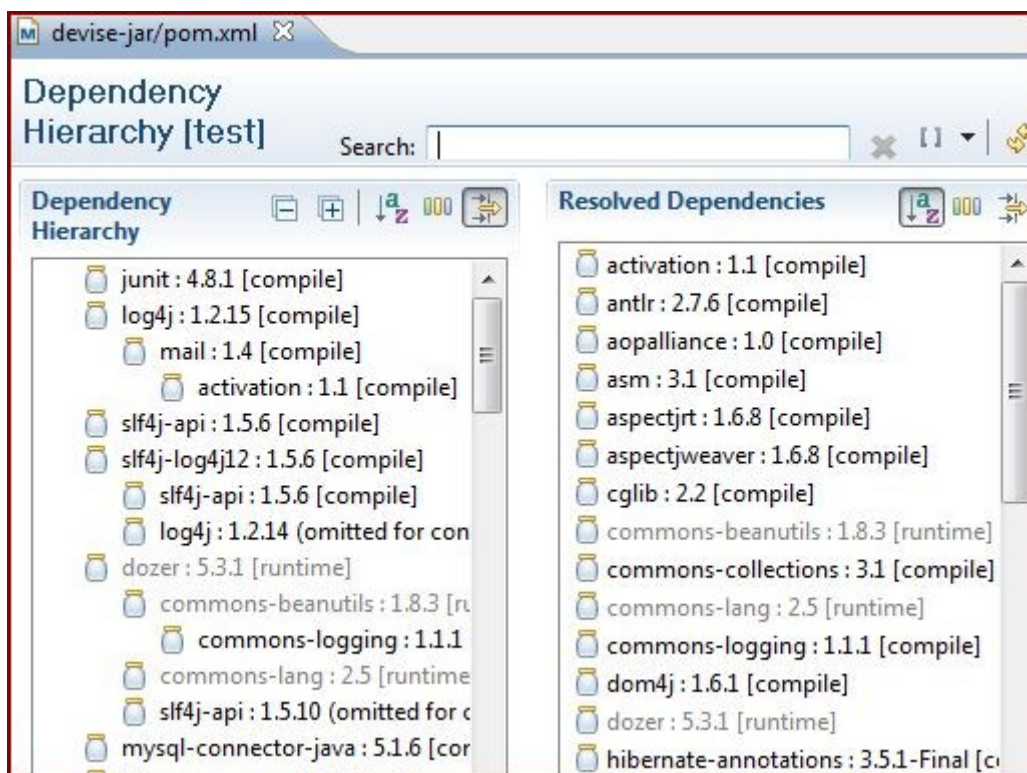
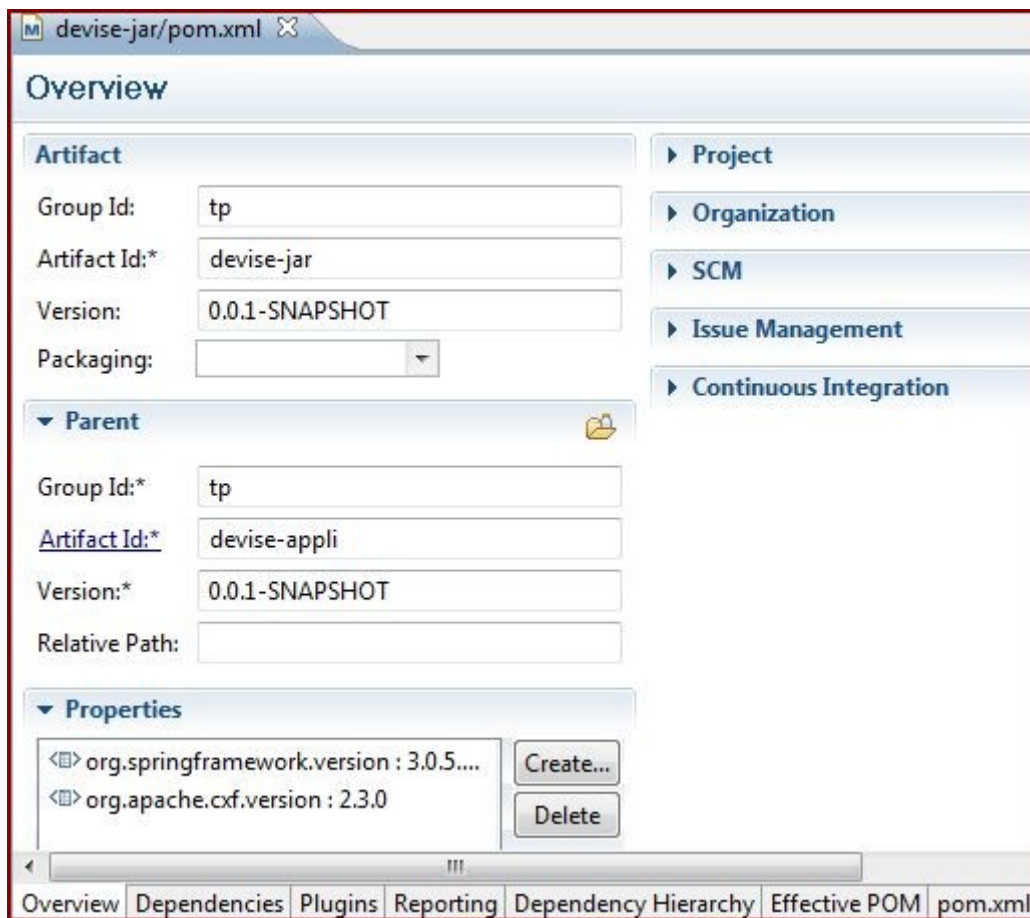
17.2. Assistants "eclipse/m2e" pour créer un nouveau projet maven



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



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



NB :

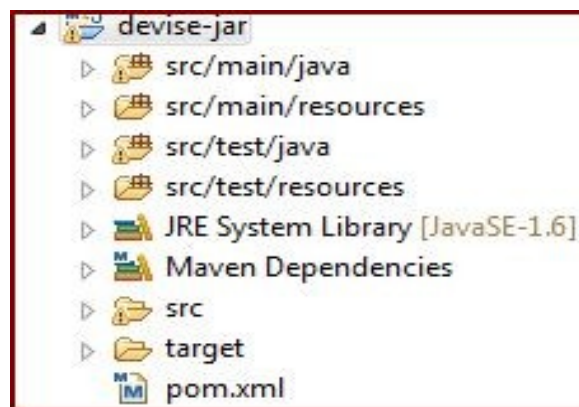
- l'onglet "**pom.xml**" correspond au contenu exact du fichier "pom.xml" du projet courant.
- l'onglet "**Effective POM**" correspond à **la somme du contenu de pom.xml et de toute la configuration héritée** (projet parent + config par défaut) .
- l'onglet "**Dependency Hierarchy**" permet de bien visualiser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

17.4. Structure des projets "eclipse maven"

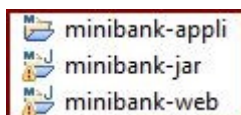
Plugin eclipse "m2e" pour maven (partie 2)

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

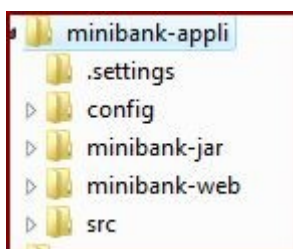
- * ont une structure "maven" classique (*src/main/java, ...*)
- * n'ont pas la structure classique d'un projet java (*src,bin*)
ni la structure d'un "dynamic web projet" (*webContent,...*)



Cas d'une structure multi-modules:



(structure vue à plat dans eclipse)



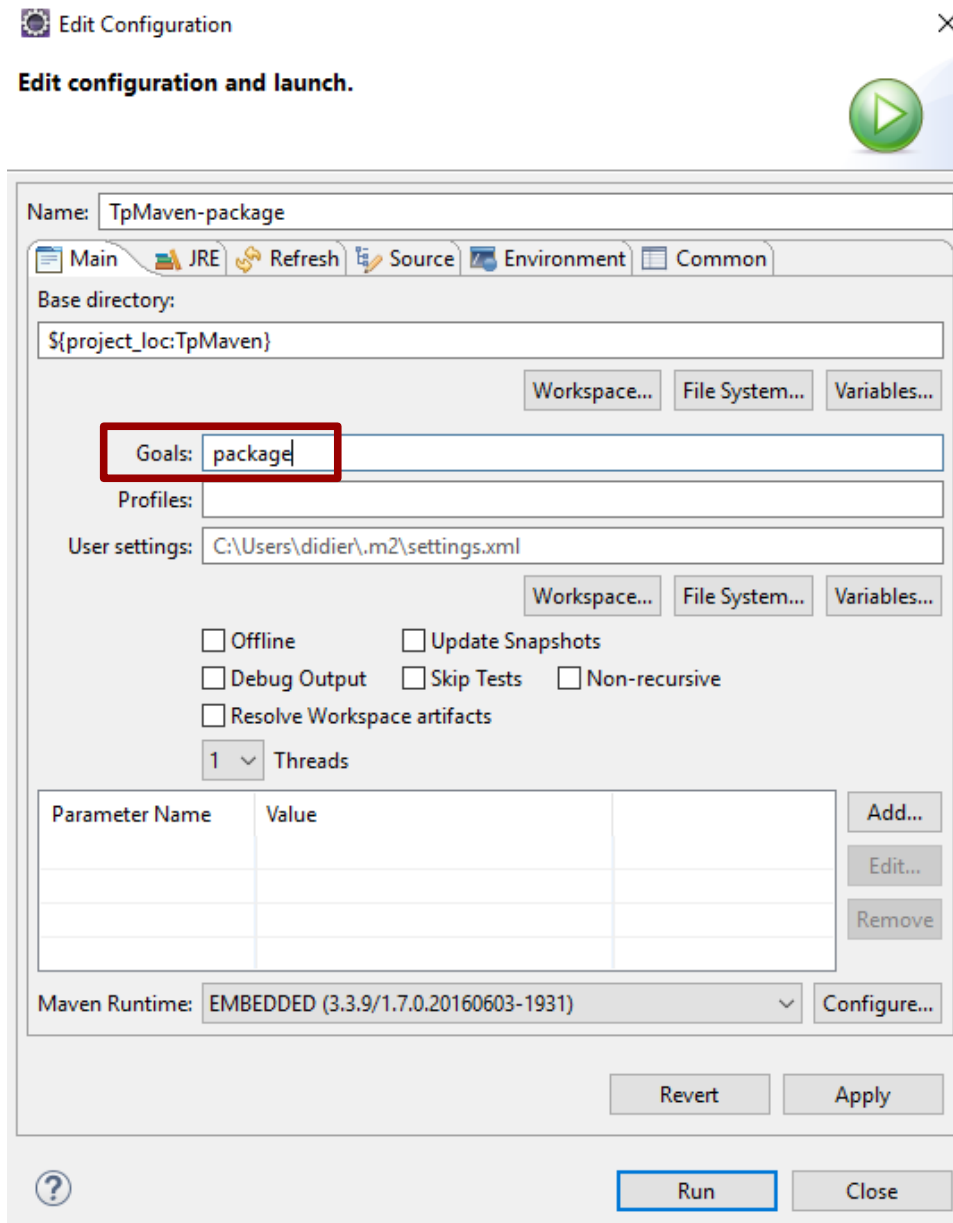
(structure arborescente sur le file system hôte).

17.5. Lancement d'un build maven depuis eclipse

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

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

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



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

NB : il est possible de retoucher certains paramètres via le menu "project / run configurations ..."

17.6. Diverses astuces "eclipse/maven" :

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

- Le plugin eclipse "m2e " nécessite absolument le jdk complet (plutôt que le JRE) pour fonctionner. Il faut donc pointer vers le répertoire "jdk ..." plutôt que "jre ..." au niveau du menu "**Windows préférences / Java / Installed JREs ...**"
- En cas de désynchronisation "maven-eclipse" , on peut tenter la séquence suivante :
run as / maven clean
run as / maven install
project / clean (eclipse)
server / tomcat / clean (si projet web)
run as / run on server (si projet web)
refresh navigateur (si projet web)
- En cas de bug inexpliqué et non identifié , il peut quelquefois être utile de redémarrer entièrement eclipse (comportant quelquefois quelques bugs ou bien rendu instable suite à des manipulations erronées) .
- Quelques fois , certains ".jar" sont mal téléchargés (mal recopiés , fichiers corrompus) et il faut manuellement **supprimer certains sous répertoires de \$HOME/.m2/repository** en fonction des groupId/artifactId du message d'erreur maven .

VI - Référentiel maven , Nexus , profils maven

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

Configuration des référentiels "maven" (partie 1)

Un nouveau référentiel maven (interne à une entreprise/organisation) est simplement structuré comme le référentiel local (.m2/repository).

--> même contenu (à recopier ou alimenter)

même structure arborescente (selon groupId , artifactId et version)

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

<http://repo1.maven.org/maven2>

nouveau référentiel distant interne entreprise/organisation

<http://myserver/repo>

(*) Proxy-ing (avec copies proches)

Configuration des référentiels "maven" (partie 3)

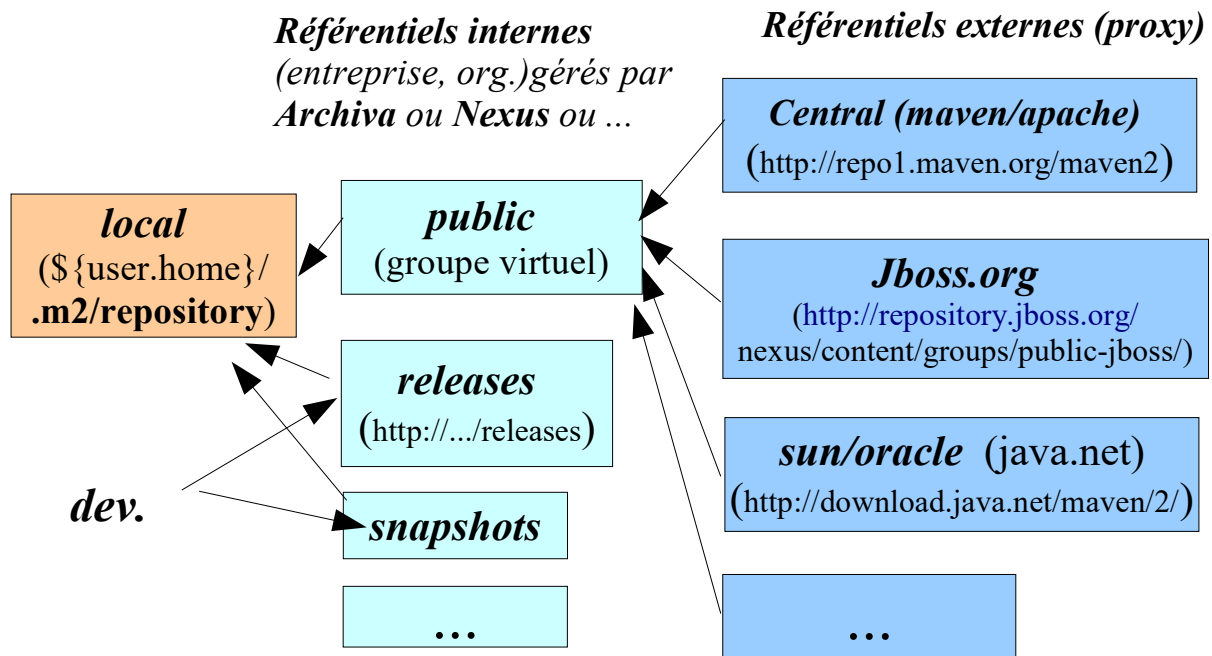
Un référentiel maven est très souvent géré par un logiciel spécialisé de type "**repository manager**".

Les produits les plus connus sont *Proximity* , *Nexus* et *Archiva* .

Archiva gère par exemple les fonctionnalités suivantes:

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

Vue globale sur les référentiels



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

Par défaut, lorsque rien n'est configuré , maven utilise le référentiel distant "<http://repo1.maven.org/maven2>" et le référentiel local **\$HOME/.m2/repository** où ".m2" est un répertoire caché placé dans le répertoire de l'utilisateur .

...

Le fichier "\$HOME/.m2/settings.xml" peut éventuellement être modifié pour configurer le référentiel local :

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

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

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

</settings>
```

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

\$HOME/.m2/settings.xml

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

1.3. Référentiel spécifique (interne à l'entreprise)

Il est possible de configurer de nouveaux référentiels au sein d'une organisation (entreprise).

...

Une fois en place , ce nouveau référentiel pourra être référencé/utilisé de la façon suivante:

pom.xml

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

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

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

1.4. Préciser le référentiel de distribution (produits)

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

pom.xml

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

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

settings.xml

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <servers>
    <server>
      <id>mycompany-repository</id>
      <username>jvanzyl</username>
      <!-- Default value is ~/.ssh/id_dsa -->
      <privateKey>/path/to/identity</privateKey> (default is ~/.ssh/id_dsa)
      <passphrase>my_key_passphrase</passphrase>
    </server>
  </servers>
  ...
</settings>
```

2. Repository Manager (Nexus ou ...)

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

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

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

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

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

2.1. Installation et configuration de "Sonatype Nexus 2"

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

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

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

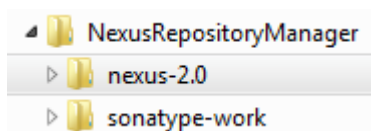
NB: sous windows , il est également possible de faire fonctionner **nexus** comme un *service windows*. Pour cela il faut lancer l'instruction "**nexus install**" au sein d'une fenêtre de commande lancée en tant qu'administrateur (depuis raccourci et clic droit approprié).

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) .

Arborescence du produit Nexus2:



2.2. Configuration d'un référentiel sous Nexus

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

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

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

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

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

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

The screenshot shows the 'add...' form for a repository in the Nexus web console. The form includes the following fields:

- 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/./../sona
- Override Local Storage Location**: (empty)
- Remote Repository Access**: (expanded section)
 - Remote Storage Location**: http://repository.jboss.org/nexus/content/groups/public-jboss/

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

The screenshot shows the configuration for the 'Public Repositories' group in the Nexus web console. The form includes the following fields:

- 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

2.3. Utilisation de Nexus en "lecture/téléchargement"

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

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

```

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

```

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

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

```

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

```

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

```

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

```

```

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

```

2.4. Utilisation de Nexus en "écriture/alimentation"

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

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

```

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

```

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

```

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

```

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

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

Il est éventuellement possible d'alimenter un référentiel à partir d'un artefact 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 sélectionné le référentiel "*3rd party*") :

3rd party

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

GAV Definition: ★ GAV Parameters ▼ ⓘ

Auto Guess: ☒ ⓘ

Group: ★

Artifact: ★

Version: ★

Packaging: ★ Select... ▼ ⓘ

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

2.5. Autres fonctionnalités de nexus

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

3. Profils "maven"

Profils "maven" (utilité , configuration)

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

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

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

Types de configuration possible des profils "maven":

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

3.1. Activation selon la version de java

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

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

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

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

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

3.3. Activation selon une propriété système java

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

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

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

Pour activer selon variable d'environnement:

-Dxxx.yyy=%VAR_XX_YY% dans .bat
ou -Dxxx.yyy=\${VAR_XX_YY} dans .sh

3.4. Activation selon un fichier manquant ou existant

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

3.5. Profils nommés à activer explicitement

```
<profiles>
  <profile>
    <id>profile-1</id>
    ...
  </profile>
  <profile>
    <id>profile-2</id>
    ...
  </profile>
</profiles>
```

et

mvn groupId:artifactId:goal -P profile-n

Eventuel profil nommé et activé "en dur":

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

ou encore

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

3.6. Exemple concret (switch de configuration)

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

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

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

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


4. Filtrage des ressources

Principe:

Certains fichiers de ressources (.properties , .xml) peuvent éventuellement comporter des valeurs basées sur des **variables qui seront renseignées lors de la construction via maven**.

Ceci permet d'obtenir *plusieurs configurations différentes* (pour les bases de données par exemple) *en fonction d'un choix de profile*.

Exemples:

exemple.properties (dans src/main/resources)

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

exemple.xml (dans src/main/resources)

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

Configuration maven:

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

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

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

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

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

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

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

Possibilités de <includes> , <excludes> :

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

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

WEB_INF) .

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

Exemple:

via le filtre

configurations/properties/**config.prop**

```
interpolated_property=some_config_value
```

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

configurations/**config.cfg**

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

pom.xml

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

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

VII - Tests d'intégration avec maven (et selenium)

1. Tests d'intégration avec maven

1.1. Positionnement des tests d'intégration avec maven

Dernières phases du cycle "build" de maven :

<i>Phases</i>	<i>Tâches potentiellement prévues</i>	<i>Exemples</i>
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"

1.2. 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** .

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
```

Le tableau suivant montre les principales différences entre les plugins "**maven-failsafe-plugin**" et "**maven-surefire-plugin**" :

	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 (interruption du cycle)	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

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>
```

1.3. Exemples de tests d'intégration

Un des tests d'intégration très classique consiste à :

- démarrer un serveur d'application (ex : Jetty) au niveau de la phase "pre-integration-test"
- démarrer l'agent "selenium-server" au niveau de la phase "pre-integration-test"
- 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 ,) .
- arrêter l'agent "selenium-server" au niveau de la phase "post-integration-test"
- stopper un serveur d'application (ex : Jetty) au niveau de la phase "post-integration-test"
- constater l'échec (ou la réussite) des tests (et du cycle maven) au niveau de la phase "verify".

Autre variantes assez classique(sans selenium):

- Tester un service web SOAP via l'api JAX-WS
- Tester un service web REST via l'api HttpComponents/ HttpClient d'apache .
- Tester un accès distant (RMI) vers un EJB
-

NB :

- L'annexe sur l'essentiel de "selenium" permet d'approfondir "selenium" et le lien entre maven et "selenium".
- L'annexe sur le déploiement JEE avec le plugin cargo donne un bon aperçu de la configuration maven nécessaire et des possibilités de "cargo" .

Exemple de configuration "cargo" dans le cadre des tests d'intégration :

```
... <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>
        <dependency>
          <groupId>org.apache.tomcat</groupId>
          <artifactId>el-api</artifactId><!-- doit faire reference a
                                une dependance existante du projet -->
        </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> <!-- sans version ici (reference)-->
        </dependency>
      </dependencies>
    </container>
  </configuration>
</plugin>
```

```
        </dependencies>
      </container>
    </configuration>
  </plugin> ...
```

NB : Dans le cas de Jetty (rapide à démarrer), il faut ajouter les dépendances suivantes dans pom.xml :

```
<dependency>
  <groupId>javax.el</groupId>
  <artifactId>el-api</artifactId>
  <version>2.2</version>
  <scope>provided</scope> <!-- for jetty launched by cargo / integration-test-->
</dependency>

<dependency>
  <groupId>org.glassfish.web</groupId>
  <artifactId>el-impl</artifactId>
  <version>2.2</version>
  <scope>provided</scope> <!-- for jetty launched by cargo / integration-test-->
</dependency>
```

Variante de configuration pour tomcat7 ou 8 :

```
<configuration>
  <wait>>false</wait>
  <container>
    <containerId>tomcat7x</containerId>
    <type>installed</type>
    <home>/opt/apache-tomcat-7.0.30</home>
  </container>
  <configuration>
    <type>existing</type>
    <home>/opt/apache-tomcat-7.0.30</home>
  </configuration>
</configuration>
```

2. Tests web (http/html) via selenium

Attention ce chapitre écrit il y a longtemps est basé sur une vieille version de selenium .
Quelques adaptations sont à prévoir avec des versions plus modernes .

2.1. Présentation et installation de Selenium

Présentation et installation 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.

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>

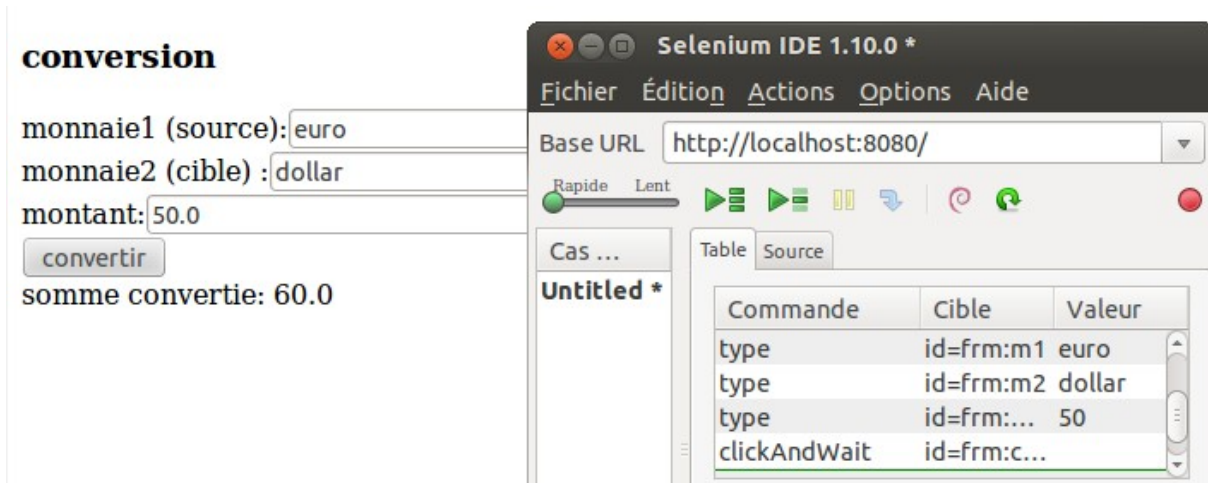
Attention (petit piège JSF et d'autres frameworks WEB) :

Il faut absolument définir des "id" explicites au niveau de chaque composant des pages ".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 :<h:inputText id="m1" value="#{conv.monnaie1}"/> ...
```


2.2. Enregistrement de séquence 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 "Selelium-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.

2.3. Selenium au sein de tests d'integration "maven"

Dépendance "maven" pour utiliser l'api "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;

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);
}
}
```

VIII - Lien maven / scm, gestion des releases

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

1.1. Liaison avec un référentiel de code source SVN

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

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

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

Ou bien url de type :

scm:git:file:///media/sf_ext/tp/local-git-repositories/env-ic-my-java-app1

ou encore scm:git:http://www.xy.com/my-repo.git

ou encore scm:cvss :... , 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

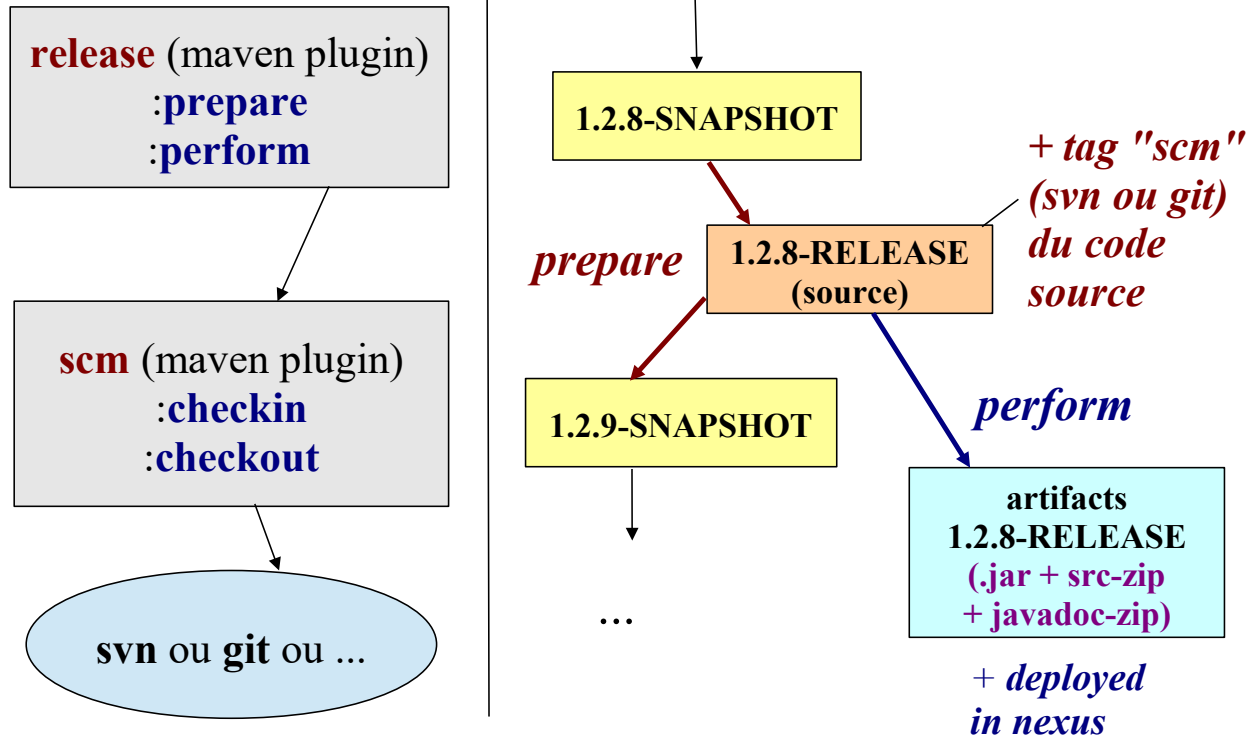
scm:tag - command for tagging a certain revision

==> 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 à son tour réutilisé par un plugin de plus haut niveau intitulé "release" .

1.2. Gestion des "releases" pilotées par maven

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



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

Le plugin "release" sert à changer de version "maven" (ex : 1.1-SNAPSHOT vers 1.1-RELEASE) de façon cohérente à tous les niveaux (pom.xml de l'application , des sous modules , dans SCM).

Préparation d'une version "released" via maven

(a) *mvn_scm_release_prepare_my_java_app1.sh*

```

cd my-java-app1
# vérifications effectuées par release:prepare :
# * pas de modifications locales (sans commit) : même en TP !!!
# * pas de dépendances vers des xxx-SNAPSHOT
# informations suggérées / demandées (que l'on peut saisir si pas -B):
# * new mvn/pom released 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**".

IX - Intégration continue avec Hudson / Jenkins

1. Premiers pas avec Hudson/Jenkins

1.1. Présentation et installation de kenkins

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



NB : il est également possible de démarrer une version récente de jenkins sans serveur tomcat via un script de ce genre :

startJenkins.bat

```
set JAVA_HOME=C:\Program Files\Java\jdk-11.0.12
set PATH="%JAVA_HOME%\bin";%PATH%
REM java -jar jenkins.jar -D"hudson.plugins.git.GitSCM.ALLOW_LOCAL_CHECKOUT=true"
java -jar jenkins.jar
```

et après un tel lancement l'url menant à la console jenkins sera simplement **http://localhost:8080**

1.2. Configuration nécessaire lors du premier démarrage

Lire le mot de passe temporaire à la console lors du premier démarrage (ex:
`d1223a3ba2a44d079ecb7deec0625de8`)
et reporter/recopier celui-ci dans la console de jenkins

Installer quelques plugins fondamentaux (ceux qui sont suggérés)

Configurer un compte principal (administrateur) pour les futurs démarrages :
par exemple `username=admin password=admin123`

1.3. Installation ou mise à jour de plugins pour Jenkins

Menu "tableau de bord" / "Administrer Jenkins" / "Gestion des plugins"

1.4. Configuration élémentaire d'une tâche "jenkins"

Menu "tableau de bord" / "Nouveau item"

puis :

- donner un nom (ex : `jobXy`)
- choisir souvent "`projet free-style`" pour les cas simples/ordinaires
- **OK**

Dans la partie "gestion du code source", choisir généralement :

- **GIT**

et préciser l'url du référentiel git (par exemple <https://github.com/.../repoXy>)

Attention: les versions récentes de Jenkins n'acceptent des URLS de type `file:///c:/xx/yy` qu'avec l'option `-D"hudson.plugins.git.GitSCM.ALLOW_LOCAL_CHECKOUT=true"` à fixer au démarrage et dans la partie "branch to build" on pourra par exemple choisir `/master` ou `*/main` .*

Dans la partie "**build**", choisir généralement :

- **Invoquer les cibles Maven de haut niveau**

et préciser la "*cible (ou goal)*" maven à déclencher (ex : `clean package`)

NB: si le projet maven à construire est dans un sous répertoire du référentiel git (cas pas très conseillé mais admis), alors au niveau du build on peut préciser un chemin menant au pom.xml de type `sous_rep1` ou bien `sous_rep1/sous_sous_rep2` dans config avancée .

Sauvegarder assez rapidement ces configurations essentielles.

Les configurations secondaires annexes pourront être ajoutées ultérieurement

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

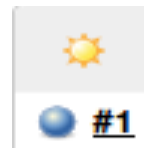

2. Notifications des résultats (rss , email , ...)

Notifier les développeurs du résultats des tests (état du projet)

Au niveau d'un logiciel d'intégration continue, on peut souvent paramétrer un mécanisme qui servira à avertir les développeurs sur :

- * **résultats des tests** (statistiques , erreurs , ...)
- * **état du projet** (**vert** , **rouge** , ...) et évolution
- * ...

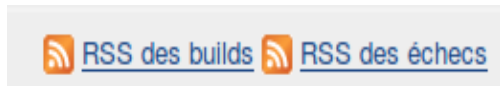
On parle souvent en terme imagé de **météo du projet** .



Les notifications peuvent être effectuées par :

- * des **envois d'e-mail** (à partir d' une liste des adresses e-mail des développeurs de l'équipe)
- * des "**flux RSS**"
- * une **simple consultation régulière du statut du projet dans jenkins**

Notifications élémentaires par flux rss (jenkins)



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 .

Configuration système "SMTP" nécessaire (jenkins)

Notification par email

Serveur SMTP	<input type="text" value="localhost"/>	
Suffixe par défaut des emails des utilisateurs	<input type="text"/>	
<input checked="" type="checkbox"/> Utiliser l'authentification par SMTP		
Nom d'utilisateur	<input type="text" value="root"/>	
Mot de passe	<input type="password" value="...."/>	
Utiliser SSL	<input type="checkbox"/>	
Port SMTP	<input type="text" value="25"/>	
Reply-To Address	<input type="text"/>	
Jeu de caractères	<input type="text" value="UTF-8"/>	
<input checked="" type="checkbox"/> Tester la configuration en envoyant un e-mail de test		
Destinataire du courriel de test	<input type="text" value="user1@localhost"/>	
Email was	<input type="button" value="Tester la configuration"/>	

James : petit serveur SMTP en java pour tests (tp, ...)

/opt/james-2.3.2/bin

lancer le server smtp via [sudo-]**run.sh** (dans terminal) .

mot de passe = *formation* , arrêt via *ctrl-c*

Deux comptes pré-crées: (**user1/pwd1** et **user2/pwd2**)

==> **user1@localhost** / pwd1 configuré dans ThunderBird .

si nécessaire (pour créer autres comptes):

lancer telnet localhost 4555

 root , root

 adduser user1 pwd1

 adduser user2 pwd2

 Quit

paramétrage smtp (pour connexions):

server: **localhost** , port : **25** , username/password : **root/root**

Notification par email (depuis Jenkins)

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

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

☒ Notification par email

Recipients

user1@localhost user2@localhost

☒ Send e-mail for every unstable build

e-mail reçu en cas d'échec. →

Sujet **Jenkins build became unstable: my-java-app1 #3**

Pour Moi☆, user2@localhost☆

See <http://localhost:8585/jenkins/job/my-java-app1/3/changes>

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

3. Différents types de "builds" (avec Jenkins)

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

X - Rapports qualimétriques / miniAudit (Sonar)

1. Rapport / métriques (avec Sonar)

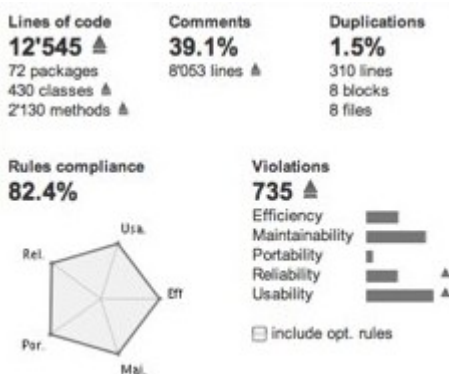
1.1. Présentation de sonar (issue de wikipedia)

Sonar est un [logiciel libre](#) permettant de mesurer la qualité du [code source](#) sur les projets de développement [java](#). Sonar est distribué selon les termes de la licence [LGPL](#) v3.

Le code source est analysé suivant sept axes :

- identification des [duplications de code](#) ;
- mesure du niveau de [documentation](#) ;
- respect des [règles de programmation](#) ;
- détection des [bugs](#) potentiels ;
- évaluation de la [couverture de code](#) par les [tests unitaires](#) ;
- analyse de la répartition de la complexité ;
- analyse du Design et de l'Architecture d'une application et en faire ressortir des métriques orientées objet.

L'ensemble de ces [métriques](#) qualité permettent d'évaluer rapidement la [dette technique](#) de chaque projet. Une interface Web permet à la fois d'administrer l'outil (exclusion de code source, activation des profils qualité, définition des seuils d'alertes, ...) et de consulter les résultats en croisant les indicateurs et en offrant plusieurs modes de restitution (*clouds*, *treemap*, *hotspots*, *timemachine*, ...).



Sonar s'appuie sur 3 composants :

- un plugin [Maven](#) où une tâche [Ant](#) a en charge l'[analyse du code source](#),
- une base de données dans laquelle sont stockés l'ensemble des résultats des analyses
- un site web pour la partie reporting et pilotage.

Cette architecture permet d'utiliser Sonar pour des audits de code ponctuels, mais également dans le cadre d'une démarche d'amélioration continue

Sonar s'appuie en interne sur d'autres technologies open source très classiques :

- Duplication de code : [CPD - PMD](#) ;
- Test unitaires et couverture de code : [Cobertura](#), Clover, [JUnit](#), [Surefire](#) ;
- Règles de programmation : [Checkstyle](#), [PMD](#) ;
- Bugs potentiels : [FindBugs](#).

Concernant le niveau de documentation et les métriques standards comme la complexité et le nombre de [lignes de code](#), Sonar utilise son propre moteur d'analyse.

1.2. Installation et configuration de sonarqube

Télécharger "[sonarqube-9.6.0.59041.zip](https://www.sonarqube.org/downloads/)" depuis le site <https://www.sonarqube.org/downloads/>
Extraire le contenu de cette archive sur le poste de développement (ex: `/opt/sonarqube-9.6`).

NB : une versions 2022 de sonarqube nécessite java 11 et H2 par défaut ou bien postgres 9,10,11,12 ou 13

Lancement et arrêt (sur demande) du serveur sonar :

si **StartSonar** ne fonctionne pas bien
on peut essayer l'alternative suivante:

bin/Windows..../MyStartSonar.bat

```
REM set JAVA_HOME=C:\Program Files\Java\jdk-11.0.12
set JAVA_HOME=C:\Program Files\Java\jdk-17
set PATH="%JAVA_HOME%\bin";%PATH%

echo Starting SonarQube...
java -Xms8m -Xmx32m^
-Djava.awt.headless=true^
--add-exports=java.base/jdk.internal.ref=ALL-UNNAMED^
--add-opens=java.base/java.lang=ALL-UNNAMED^
--add-opens=java.base/java.nio=ALL-UNNAMED^
--add-opens=java.base/sun.nio.ch=ALL-UNNAMED^
--add-opens=java.management/sun.management=ALL-UNNAMED^
--add-opens=jdk.management/com.sun.management.internal=ALL-UNNAMED^
-cp "%REALPATH%..\lib\sonar-application-9.6.0.59041.jar" "org.sonar.application.App"
```

puis en arrêtant ultérieurement le serveur via Ctrl-C dans la console du serveur démarré

URL de la console de sonar: <http://localhost:9000>

NB: cette console (connecté au serveur sonar) servira à visualiser des métriques enregistrées par le plugin maven sonar

NB: Au premier démarrage username=*admin* et password=*admin*
on est obligé de changer le mot de passe ---> par exemple *admin123*

Lancer l'analyse d'un projet java/maven :dans **pom.xml**

```

...
<build>

  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.sonarsource.scanner.maven</groupId>
        <artifactId>sonar-maven-plugin</artifactId>
        <version>3.4.0.905</version>
      </plugin>
    </plugins>
  </pluginManagement>
...
</build>
...

```

```

mvn clean install -DskipTests=true
mvn sonar:sonar -Dsonar.host.url=http://localhost:9000
-Dsonar.login=admin -Dsonar.password=admin123

```

→ lancement d'un scan de l'application et enregistrement des métriques dans le serveur sonar

Affichage des rapports de sonar :

URL (par défaut) de sonar : <http://localhost:9000>

NB: il existe éventuellement un plugin eclipse permettant de directement lier sonar à eclipse .

1.3. Paramétrages des qualimétries et interprétations

L'interprétation des rapports de Sonar peut être plus aisée en s'appuyant sur le tableau suivant :

<i>Problèmes potentiels (anti-patterns , défauts récurrents)</i>	<i>Mesures de Sonar (ou ...)</i>	<i>Valeurs recherchées / visées</i>
Duplication de code (pas assez de factorisation)		
Code localement trop complexe (<i>pas assez de décomposition</i> en sous	Complexité "cyclomatique" (nombre de niveaux avec des "if" , "while" , "for" dans une	De 1 à 3 : TB 4 à 6 : B

fonctions/méthodes)	méthode)	7 à 9 : Moyen 12 = grand maximum !!!
Mauvaise conception (ex : dépendances circulaires, ...)	Analyse des dépendances (idéalement concret--->abstrait , pas de circulaire ,)	
Pas (ou pas assez) de test unitaire	Couverture du code par les tests (via <i>Cobertura</i>)	% de couverture (idéalement élevé)
Pas de respects des standards (conventions syntaxiques non respectées, ...)	Erreurs de style (remontées par " <i>checkStyle</i> ") .	Pas trop de "violations de styles" (sauf si prévues par conventions "org" ou ...)
Bugs potentiels (non décelés / pas assez de protection) <u>Exemple</u> : NullPointerException si référence à "null" non testée	Nombre de bugs potentiels remontés par " <i>findBugs</i> "	Le moins possible !
Pas ou trop de commentaires dans le code (mieux vaut méthode avec nom "parlant" que commentaire de 5 lignes au dessus de fct1(a,b))		
...		

1.4. Quelques repère pour une bonne conception:

Quelques critères de qualité (métriques sur les packages):

Abstractness (A) ou degré d'abstraction.	Pourcentage de classes concrètes par rapport aux classes abstraites. Si proche de 0 : package concret . Si proche de 1 : package abstrait.	Le degré d'abstraction d'un package doit tendre vers l'une ou l'autre des deux borne : 0 ou 1. Une valeur proche de 0.5 montrerait une mauvaise écriture du code.
Afferent coupling (Ca) où couplage par dépendance descendante.	Le nombre de packages tiers qui utilisent les classes du package analysé/courant.	Si ce nombre est très grand, il est peut être nécessaire de fragmenter ce package "trop central" .
Efferent coupling (Ce) où couplage par dépendance ascendante.	Le nombre de packages utilisés par les classes du package analysé/courant	C'est un indicateur d'indépendance du code. Plus ce nombre est faible, mieux c'est.
Instability (I) où degré de stabilité	Indicateur de résilience du package : propriété de stabilité par rapport à la mise à jour d'autres packages. (cet indicateur est lié à Ca et Ce) .	Proche de 0 : package stable (cas idéal d'un package abstrait), si proche de 1 : package instable (normal pour un package concret d'implémentations)
Distance from the Main Sequence (D)	Distance normale (perpendiculaire) à la droite $A + I = 1$. Proche de 0 : le package coïncide avec la « Main séquence », proche de 1 : très éloigné de la « Main séquence ».	Dans le cas idéal ($D=0$), un package est soit complètement abstrait et stable ($A=1, I=0$) ou complètement concret et instable ($A=0, I=0$). Si D est proche de 1, c'est mauvais et est le signe d'un package mal géré.
Dépendance cyclique (Package Dependency Cycles)	Ce critère indique s'il existe des dépendances cycliques	Pas bien si il y en a!
...		

Analyse via *Metrics* , *JDepends* , ...

ANNEXES

XI - Annexe – Compléments sur Maven

1. Variables et versions (maven)

1.1. Variables prédéfinies de maven

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

D'autre part, tous les éléments (xml) présents dans le fichier *settings.xml*, peuvent être référencés via le préfixe "*settings.*"

Depuis Maven 3.0, toutes les propriétés "pom.*" sont "deprecated".

Il faut utiliser les propriétés en "project.*" à la place .

<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"
...	

1.2. Versions des artifacts "maven"

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

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

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

Exemples:

0.0.1-SNAPSHOT
1.0-SNAPSHOT
2.0-SNAPSHOT

Structure habituelle d'un numéro de version:

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

Incréments

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

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

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

2. Bonnes pratiques

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

Exemple : si groupId = "tp.myapp.xy"

alors packages

"tp.myapp.xy.itf.domain.dto" , "tp.myapp.xy.itf.domain.service" ,
 "tp.myapp.xy.impl.domain.service" , "tp.myapp.xy.impl.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. Déploiement Jee avec CARGO (via maven)

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

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

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

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

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

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

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

mvn cargo:start

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

3.1. 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>

```

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

```

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

<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.cargo</groupId>
      <artifactId>cargo-maven2-plugin</artifactId>
      <version>1.4.12</version>
      <configuration>
        <wait>true</wait>          <!-- waiting for Ctrl-C -->
        <container>
          <containerId>jetty8x</containerId>
          <type>embedded</type>
          <dependencies>
            <dependency>

```

```

        <groupId>javax.el</groupId>
        <artifactId>el-api</artifactId>      <!--reference sans version -->
    </dependency>
    <dependency>
        <groupId>org.glassfish.web</groupId>
        <artifactId>el-impl</artifactId>
    </dependency>
    <dependency>
        <groupId>javax.servlet.jsp</groupId>
        <!-- <artifactId>jsp-api</artifactId> ancienne version -->
        <artifactId>javax.servlet.jsp-api</artifactId>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
    </dependency>
</dependencies>
</container>
</configuration>
</plugin>
</plugins>
</build>

```

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

```

<!-- ..... dans le pom.xml du sous projet "....-ear" ..... -->
<build>
    <plugins><plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-ear-plugin</artifactId>
        <version>2.3.1</version>
        <configuration>
            <generateApplicationXml>true</generateApplicationXml>
            <includeJar>>false</includeJar>
            <defaultLibBundleDir>lib</defaultLibBundleDir>
            <modules>
                <webModule>
                    <groupId>com.mycompany.jee5app1</groupId>
                    <artifactId>my-jee5app1-web</artifactId>
                    <contextRoot>my-jee5app1-web</contextRoot>
                </webModule>
                <ejbModule>
                    <groupId>com.mycompany.jee5app1</groupId>
                    <artifactId>my-jee5app1-ejb</artifactId>
                </ejbModule>
            </modules>
        </configuration>
    </plugin>
    <plugin>
        <groupId>org.codehaus.cargo</groupId>
        <artifactId>cargo-maven2-plugin</artifactId>
        <version>1.4.12</version>
        <configuration>

```

```

<wait>true</wait>      <!-- waiting for Ctrl-C -->
<container>
  <containerId>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"

4. Configuration "maven" pour applications "JEE"

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

4.1. Application "web" avec services "Spring" pour tomcat6

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>
</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>

....

<dependencies>
  ...
  junit , log4j, slf4j-api , slf4j-log4j12 , dozer ,
  mysql-connector-java , hibernate-core, jta, hibernate-commons-annotations,
  hibernate-entitymanager, validation-api, hibernate-validator,
  spring-core , spring-context, spring-beans, spring-aop, javax.inject,
  aspectjrt, aspectjweaver, spring-jdbc, spring-orm, spring-tx, spring-test
  ...
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.3.2</version>
      <configuration>
        <source>1.6</source>    <target>1.6</target>
      </configuration>
    </plugin>
  </plugins>
  <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>

  <dependencies>

```



```

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

...
servlet-api , sp-api , log4j, slf4j-api, slf4j-log4j12,
validation-api, hibernate-validator , spring-core, spring-context,
spring-beans, spring-aop, javax.inject, spring-web,
jstl, myfaces-api, myfaces-impl,
richfaces-components-ui, richfaces-core-impl,
cxf-api , cxf-rt-frontend-jaxws, cxf-rt-transport-http
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId> <version>2.3.2</version>
      <configuration> <source>1.6</source> <target>1.6</target></configuration>
    </plugin>
  </plugins>
  <finalName>my-webapp1-web</finalName>
</build>
</project>

```

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

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

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

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

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <artifactId>my-jee5app1</artifactId>
        <groupId>com.mycompany.jee5app1</groupId>    <version>1.0-SNAPSHOT</version>
    </parent>
    <groupId>com.mycompany.jee5app1</groupId>
    <artifactId>my-jee5app1-ear</artifactId>
    <packaging>ear</b>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 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>
  </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>

  <repositories>
    <repository>
      <id>java.net2</id>
      <name>Java.Net Maven2 Repository, hosts the javaee-api dependency</name>
      <url>http://download.java.net/maven/2</url>
    </repository>
  </repositories>

  <dependencies>
    <dependency>
      <groupId>javaee</groupId>
      <artifactId>javaee-api</artifactId>
      <version>5</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.3.2</version>
        <configuration><source>1.6</source><target>1.6</target></configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-ejb-plugin</artifactId>
        <configuration> <ejbVersion>3.0</ejbVersion> </configuration>
      </plugin>
    </plugins>
    <finalName>my-jee5app1-ejb</finalName>
  </build>
</project>
```

sous module "web" (avec dépendances adéquates):

```
<?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>
  </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 richfaces 3.3 -->
    <repository>
      <id>jboss.org</id>
      <url>http://repository.jboss.org/maven2</url>
    </repository>

    <!-- 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>

    <dependency>
      <groupId>javax.servlet</groupId> <artifactId>servlet-api</artifactId>
      <version>2.5</version> <scope>provided</scope>
    </dependency>

    <dependency>
      <groupId>javax.servlet.jsp</groupId> <artifactId>jsp-api</artifactId>
      <version>2.1</version> <scope>provided</scope>
    </dependency>
    ...
  </dependencies>
```

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.3.2</version>
      <configuration> <source>1.6</source> <target>1.6</target> </configuration>
    </plugin>
  </plugins>
  <finalName>my-jee5app1-web</finalName>
</build>
</project>
```

5. Gestion avancée des dépendances (BOM)

5.1. 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"

5.2. 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>

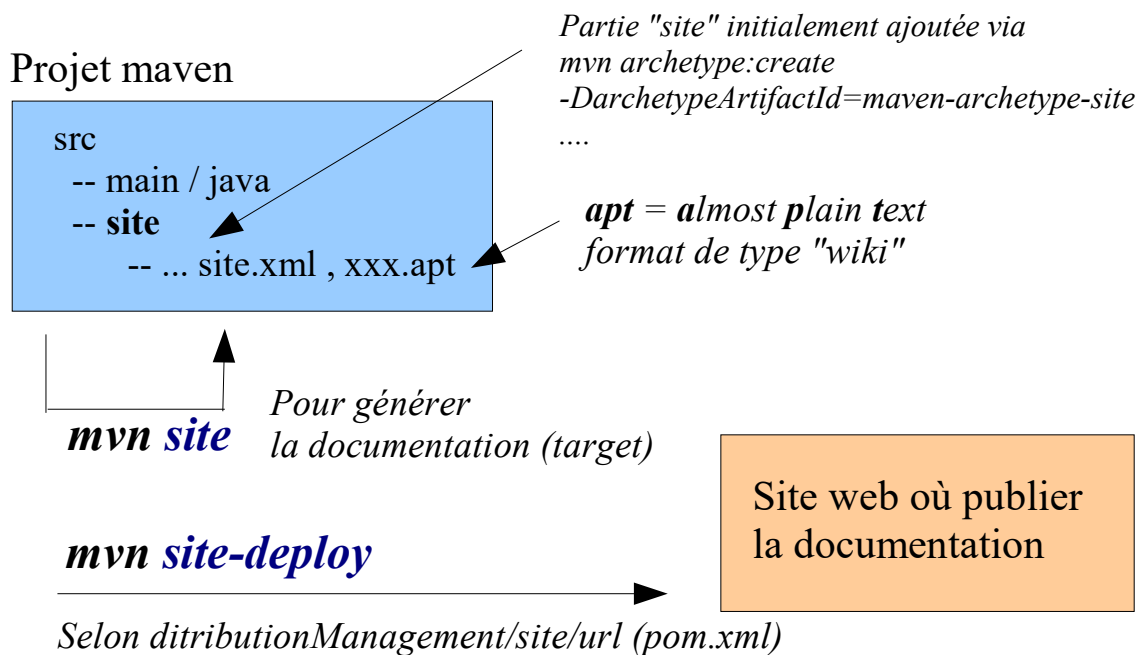
      <dependency>
        <artifactId>slf4j-log4j-bom</artifactId>
        <groupId>com.mycompany.bom.util</groupId>
        <version>1.0-SNAPSHOT</version>
        <scope>import</scope>
        <type>pom</type>
      </dependency>
    <dependency> ... </dependency>
  </dependencies>
</dependencyManagement>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


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>
-->
```

6. Génération et publication d'une documentation

Génération de documentation avec "maven"



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)

7. Javadoc (via maven)

Générer javadoc avec "maven"

... (dans pom.xml)

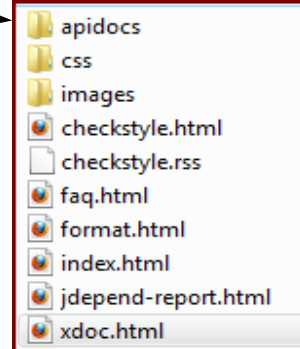
```

<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-javadoc-plugin</artifactId>
      <version>2.7</version>
      <!-- <configuration>
        <minmemory>128m</minmemory>
        <maxmemory>512m</maxmemory>
      </configuration> -->
    </plugin>
  </reporting>

```

mvn javadoc:javadoc

target/site/



8. 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/

