TP AL ESIR2

Introduction à Maven et à l'intégration continue[1]

Objectifs du TP

Avant de travailler sur des TP de test logiciel, vous allez utiliser dans ce premier TP des outils liés au processus de développement logiciel, en l'occurrence Maven et des outils d'intégration continue souvent associés tels que Sonar et Jenkins. Le TP de test qui suivront utiliserons Maven. Le but de ce TP est donc de :

- **Comprendre le fonctionnement de Maven**
- **Utiliser les artefacts**
- **Configurer un projet IntelliJ avec Maven**
- **Créer son propre MOJO**
- **Générer des rapports Maven**
- **Utiliser git pour sauvegarder le code source de votre projet**
- **Utiliser des outils d'intégration continue**

Liens utiles

- Site de Maven : http://maven.apache.org/
- Plugin Checkstyle : http://maven.apache.org/plugins/maven-checkstyle-plugin/
- FAQ maven developpez.com : http://java.developpez.com/faq/maven/

Environnement

Selon le 3ième lien donné ci-dessus, Maven est essentiellement un outil de gestion et de compréhension de projet. Maven offre des fonctionnalités de :

- Construction, compilation
- Documentation
- Rapport
- Gestion des dépendances
- Gestion des sources
- Mise à jour de projet
- Déploiement

[1]Ce TP a été créé à l'origine par Olivier Barais (olivier.barais@irisa.fr), puis amélioré par Arnaud Blouin (arnaud.blouin@irisa.fr)

Utiliser Maven consiste à définir dans chaque projet à gérer un script Maven appelés POM : *pom.xml*. Nous allons voir dans ce TP qu'un POM permet de définir des dépendances, des configurations pour notamment construire, tester, mettre en paquet des artefacts logiciels (exécutables, tests, documentations, archives, etc.). Pour cela, Maven récupère sur des dépôts maven les outils dont il a besoin pour exécuter le POM. Utiliser Maven requière donc : une (bonne) connexion à Internet car il télécharge beaucoup de choses ; de l'espace disque pour la même raison. Les artefacts qu'il télécharge sont originellement stockés dans le dossier *.m2* dans votre home-dir. Ce dossier contient également le fichier de configuration Maven : *settings.xml*.
La première étape consiste donc à configurer Maven pour changer l'endroit où les artefacts téléchargés seront stockés afin d'éviter des problèmes d'espace disque. Pour ce faire, utilisez le fichier *settings.xml* suivant à mettre donc dans le dossier *.m2* :

```
<?xml version="1.0" encoding="UTF-8"?>
<settings>
    <localRepository>w:\mavenrepository</localRepository>
    <offline>false</offline>
</settings>
```

Partie 1 : Utilisation de maven

Pour initialiser un projet Java basique, vous pouvez utiliser l'archetype maven : *maven-archetype-quickstart.* Un archetype est une sorte de modèle dont le but est de faciliter la mise en place de projets. Vous avez juste à fournir un *groupid* et un *artefactid*. Un *groupid* est l'identifiant du groupe de projets dont le projet fait partie. Un *artefactid* est l'identifiant du projet au sein de son groupe.

```
mvn archetype:create -DgroupId=[your project's group id] -DartifactId=[your project's artifact id]
```

Vous obtenez alors la structure de projet suivante :

```
|-- src
|   |-- main
|   |   `-- java
|   |       `-- [your project's package]
|   |           `-- App.java
|   `-- test
|       `-- java
|           `-- [your project's package]
|               `-- AppTest.java
`-- pom.xml
```

Par exemple si vous exécutez la commande :

```
mvn archetype:create -DgroupId=fr.esir -DartifactId=tpmaven
```

Vous obtiendrez l'arborescence suivante.

```
tpmaven/
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── fr
    │           └── esir
    │               └── App.java
    └── test
        └── java
            └── fr
```

```
└── esir
    └── AppTest.java
```

Partie 2 : Configuration d'IntelliJ

Eclipse ne supporte pas Maven par défaut, et nécessite un plugin externe assez capricieux. Nous allons donc utiliser un autre IDE : IntelliJ. Normalement tout est déjà disponible sur les machines à votre disposition. Ouvrez IntelliJ, puis faites « File » et «Open... ». Choisissez le répertoire contenant le projet maven créé dans la Partie 1, et validez.

IntelliJ possède nativement un façon d'appeler Maven facilement. Pour cela, faites « View », « Tool Windows », « Maven Projects ». Un panneau s'affiche à droite avec la liste des projets maven détectés, et l'ensemble des actions Maven qu'il est possible de faire sur le projet. Effectuer ces actions en double cliquant dans ce panneau est équivalent à taper en ligne de commande « mvn <action> ».

Partie 3 : Génération de rapports

Générer la javadoc

Ajoutez des commentaires dans le code de votre projet. Ajoutez le code suivant dans le *pom.xml* de votre projet.

```
<reporting>
  <plugins>
   <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-javadoc-plugin</artifactId>
      <version>2.9.1</version>
   </plugin>
  </plugins>
 </reporting>
```

Puis lancez : *mvn site*. Ce goal crée un site Web pour votre projet. Par défaut, les goals maven générant des fichiers travaillent dans le dossier *target* se trouvant au même niveau que le fichier *pom.xml*. Allez dans le dossier *target/site* et ouvrez le fichier *index.html*. Vous pouvez regarder la Javadoc générée en cliquant sur *Project reports*.

Valider la qualité du code avec le plugin *checkstyle*

Ajoutez à la section <plugins> dans <reporting> le plugin *checkstyle* :

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-checkstyle-plugin</artifactId>
   <version>2.11</version>
</plugin>
```

Lancez *mvn clean site* (le goal *clean* vide le dossier *target*). Une nouvelle section *Checkstyle* a été ajouté dans *Project reports*. Quelle est la norme de codage à laquelle se réfère le rapport par défaut ? Modifier le pom pour utiliser la norme de codage maven. Liens utiles (notamment le second) pour répondre à ces questions :

Site de de l'outil CheckStyle : http://checkstyle.sourceforge.net/

Site du plugin Maven : http://maven.apache.org/plugins/maven-checkstyle-plugin/

Rapport croisé de source

```
<reporting>
      <plugins>
            <plugin>
                  <groupId>org.apache.maven.plugins</groupId>
                  <artifactId>maven-jxr-plugin</artifactId>
                  <version>2.3</version>
             </plugin>
      </plugins>
 </reporting>
```

Lien utile : http://maven.apache.org/plugins/maven-jxr-plugin/. Quelle est la valeur ajoutée de ce plugin ? En particulier, montrez sa complémentarité avec *CheckStyle*. Désormais vous pouvez passer du rapport *CheckStyle* au code source en cliquant sur le numéro de ligne associé au commentaire *CheckStyle*.

Connaître l'activité du projet

But : avoir des informations sur les fichiers modifiés et leurs auteurs. Pour cela vous allez versionner en local votre projet avec *git.* Aller dans le dossier de votre projet, puis :

- créez un dépôt git : *git init* ;
- le dossier *target* ne doit pas être commité, de même que les fichiers créés par IntelliJ : créez un fichier « .gitignore », et mettez y :

```
target
*.iml
.idea
```

- versionnez tous les autres fichiers : *git add --all* . Notez qu'on se permet de faire ça uniquement parce qu'on a bien pris soin de créer un *.gitignore* qui empêche de versionner les fichiers inutiles ! Plus généralement, on utilise très peu le *–all*, à part éventuellement pour un premier commit, comme ici.
- commitez : *git commit -m "first commit"*.

Ajoutez ensuite dans la section <plugins> de <reporting> du pom le plugin *changelog* :

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-changelog-plugin</artifactId>
  <version>2.2</version>
</plugin>
```

Ajoutez dans le bloc <project> le lien vers votre projet :

```
<scm>
  <connection>scm:git:file:///home/cheminVersProjet</connection>
</scm>
```

Lancez *mvn clean site*. Le dossier */target/site* contient maintenant trois rapports d'activité :

- *changelog* : rapport indiquant toutes les activités sur le SCM.
- *dev-activity* : rapport indiquant par développeur le nombre de commits, de fichiers modifiés. Pour que ce rapport fonctionne, il faut déclarer les développeurs du projet dans le bloc <project>, de la sorte :

```
<developers>
    <developer>
        <name>Firstname Name</name>
        <email>email@address.org</email>
        <roles>
            <role>admin</role>
            <role>developer</role>
        </roles>
    </developer>
</developers>
```

- *file-activity* : rapport indiquant les fichiers qui ont été révisés.

Partie 4 : Intégration Continue

Vous allez étudier les outils d'intégration continue Sonar et Jenkins. La différence entre ces deux outils est simple : Sonar est un outil d'assurance qualité tandis que Jenkins est un outils de « release engineering ». Les deux sont évidemment complémentaire.

Intégration avec l'outil Sonar

Téléchargez Sonar : http://dist.sonar.codehaus.org/sonarqube-5.0.zip

De-compressez-le dans */tmp* (ou autre répertoire temporaire – attention à ne pas avoir le moindre espace ou caractère spécial dans le chemin complet du répertoire extrait !) et lancez-le :

```
(linux) sh /tmp/sonarqube-4.0/bin/linux-x86-64/sonar.sh start
(windows) bin\windows-x86-64\StartSonar.bat
```

Dans votre projet, lancez *mvn sonar:sonar.* Cela va faire appel à un plugin maven pour sonar, qui va contacter le serveur sonar lancé (par défaut, il va aller voir en local sur http://localhost:9000/) et lui donner les informations sur le projet.

Allez à l'adresse http://localhost:9000/. Loguez vous avec le login *admin* et le mot-de-passe *admin*. Allez dans *Quality Profiles* et changez les règles de qualités utilisées puis relancez *mvn sonar:sonar*. Baladez vous dans Sonar pour explorer ces différentes fonctionnalités. Vous pourrez ensuite arrêter sonar avec :

```
(linux) sh /tmp/sonarqube-4.0/bin/linux-x86-64/sonar.sh stop
(windows) ???
```

Intégration avec Jenkins

Sur http://jenkins-ci.org/, prenez la version **LTS** Java Web Archive (.war) pour la mettre dans */tmp* ou autre répertoire temporaire.

Vous pouvez également déplacer l'endroit où la configuration Jenkins sera stockée, afin de ne pas « polluer » votre homedir :

(linux) export JENKINS_HOME=/tmp/.jenkins

(windows) ???

Démarrez jenkins : *java -jar jenkins.war.* Allez dans votre navigateur : http://localhost:8080/.

Par défaut, jenkins ne contient pas le plugin pour gérer des repository Git, Il vous faut installer le plugin "*Git Plugin*". De plus, vous devez configurer Maven (voir *Configure System*).

Pour créer un job, vous aurez besoin d'indiquer à Jenkins où se trouve votre *repository* git. Il y a deux façons de procéder :

(a) Étant donné que git est un système décentralité, votre répertoire de projet un *repository* git fonctionnel, même s'il est seulement accessible localement. Vous pouvez donc indiquer à Jenkins un chemin local vers votre répertoire de projet.

(b) Ceci étant, dans le cadre d'un développement collaboratif, on utilise souvent un repository « central » par convention (sur un serveur géré par le groupe de développement, ou bien sur github, etc.). Dans ce cas, on indiquera à Jenkins d'utiliser ce repo principal pour l'intégration continue. Vous pouvez donc si vous le souhaitez mettre votre code sur github de cette manière :

- créez un nouveau *repository* via l'interface github
- liez votre dépôt local au distant : *git remote add origin git@github.com:login/nomRepo.git*
- mettez votre code sur ce dépôt : *git push origin master*

De retour dans Jenkins créez un job, et définissez le *repository* git à utiliser soit (a) en indiquant un chemin local (e.g. */home/toto/monProjet*), soit (b) en indiquant l'url du repository git que vous avez préalablement créé sur github (i.e. *https://github.com/login/nomRepo.git*) et enfin définissez les goals maven pour le build : pour commencer *clean package install*.

Une fois tout cela configuré, lancez un build.

Dans l'historique des builds, un icône bleu doit apparaître à la fin de la construction pour désigner la construction correcte de l'artefact (bleu car le développeur de Jenkins est Japonais et au Japon le bleu équivaut au vert chez nous, d'ailleurs un plugin Jenkins existe pour afficher des icône verte et non bleue...). Cliquez ensuite sur le lien sous « Construction du module », les artefacts créés par jenkins en utilisant le POM du projet sont visibles dont un jar. Ouvrez ce dernier, vous verrez que le *manifest* est vide. Dans les étapes suivantes vous allez compléter le POM pour obtenir un vrai jar exécutable.

Partie 5 : Packager des artefacts logiciels avec maven

Comme expliqué précédemment, ces artefacts logiciels peuvent être produits soit en utilisant directement maven en ligne de commande, soit en utilisant Jenkins. Nous allons dans cette dernière partie étudier différents plugins maven permet de réaliser de nombreuses actions de liées à la construction d'artefacts logiciels. Notez que par défaut, maven produit un artefact très simple, à savoir un jar contenant toutes les classes compilées. Nous allons voir dans cette partie comment personnaliser à la fois la compilation, et la création des artefacts.

Création d'un jar exécutable via maven

Le plugin *maven-compiler-plugin* est chargé par défaut par maven, avec n'importe pom.xml, afin d'avoir directement accès à la possiblité pour maven de compiler le projet (goal *compile*). Ceci étant, il est possible de le déclarer explicitement dans la liste des plugins du projet afin de le configurer.

Ces plugins ne sont pas liés à la documentation, donc au lieu de travailler dans la section <reporting>, vous allez ajouter un bloc <build> dans le bloc <project> de votre POM. Dans ce nouveau bloc vont être ajouté les plugins utilisés/configurés dans cette partie comme le suivant :

```
<build>
   <plugins>
      <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
```

```
            <version>3.1</version>
            <configuration>
                  <source>1.7</source>
                  <target>1.7</target>
            </configuration>
      </plugin>
   </plugins>
</build>
```

Comme vous pouvez le constater, on a ici ajouté un bloc <configuration> pour indiquer au plugin de compiler du code et du binaire compatibles Java 1.7. La documentation expliquer de nombreuses autres options de configuration pour la compilation.

Cf. : http://maven.apache.org/plugins/maven-compiler-plugin/

Le plugin *maven-jar-plugin* est lui aussi chargé par défaut par maven, afin cette fois d'avoir directement accès à la possiblité pour maven de packager le projet (goal *package*, qui lancera par lui même le goal *compile* au préalable). Il est là aussi possible de le déclarer explicitement dans la liste des plugins du projet afin de le configurer.

Cette fois-ci, on ne vous donne pas le bloc xml à ajouter : trouvez comment déclarer ce plugin dans votre POM, et comment le configurer afin de créer automatiquement un manifest dans le jar dans lequel est indiqué quelle est la classe contenant la méthode *main* à utiliser. Vous aurez besoin pour cela de configurer le paramètre « archive ».

http://maven.apache.org/plugins/maven-jar-plugin/

Lancez *mvn clean install* et exécutez le nouveau jar généré.

Exécution de test via maven

Le plugin *maven-surefire-plugin* est lui aussi chargé par défaut par maven. Il a pour rôle d'exécuter les tests du projet à chaque fois que c'est demandé (goal test), mais également à chaque fois que le projet est compilé (goals compile pour package. Ainsi, par défaut avec maven, la compilation échoue si des tests ne passent pas ! Rien à faire de particulier dans cette section, c'est simplement à titre informatif.

Création d'archives des sources et des exécutables

Le plugin *maven-assembly-plugin* permet de créer des archives. Ce plugin est notamment très utile pour créer des archives des sources ou des fichiers exécutables, cf :

http://maven.apache.org/plugins/maven-assembly-plugin/

Étudiez et adaptez l'utilisation de ce plugin dans le projet suivant :

https://github.com/arnobl/latexdraw/blob/master/latexdraw-core/net.sf.latexdraw/pom.xml

pour l'utiliser dans votre projet afin de créer un zip des sources et un autre contenant le jar exécutable.

Commitez les modifications sur github et relancez un build sur Jenkins afin d'observer les évolutions apportées.