TP Usine Logicielle

ENVOL 2012 – Biarritz
21-25 janvier 2013

1. Corriger l'environnement

Editer le fichier .bashrc et ajouter les lignes :
export MVN_HOME=/home/stagiaire/TP_UsineLogicielle/apache-maven-3.0.4
export JAVA_HOME=/home/stagiaire/TP_UsineLogicielle/jdk1.7.0
export PATH=$MVN_HOME/bin:$JAVA_HOME/bin:$PATH
Relancer le terminal
Vérifier la bonne prise en compte des variables d'environnement en exécutant :
echo $JAVA_HOME
Le chemin vers la jdk7 doit s'afficher.

aller dans ~/TP_UsineLogicielle/
exécuter sudo chown -R www-data:www-data svn

2. Démarrer les services

exécuter archiva start dans ~/TP_UsineLogicielle/apache-archiva-1.3.5/bin
exécuter startup.sh dans ~/TP_UsineLogicielle/apache-tomcat-7.0.34/bin
exécuter sonar start dans ~/TP_UsineLogicielle/sonar-3.4.1/bin/linux-x86-32

Vérifier le bon démarrage
Avec un navigateur, accéder à :
Sonar : http://localhost:9000/
Archiva : http://localhost:8080/archiva
Jenkins : http://localhost:9080/jenkins

3. Ouvrir Eclipse

exécuter eclipse dans ~/TP_UsineLogicielle/eclipse

Partie 1 : Découverte de l'environnement et 1er programme en intégration continue

Dans le projet exo1-usine-logicielle, nous allons :

- Créer une classe simple et un test unitaire simple.
- Vérifier son exécution en local.

Ensuite, nous allons mettre en place un premier processus simple d'intégration continue. Pour cela, nous allons :

- Créer le hook SVN sur l'action post-commit
- Créer un job Jenkins exécutant le build de ce programme automatiquement à chaque commit
- Améliorer le job de manière à déclencher une analyse Sonar
- Améliorer le job de manière à livrer l'artefact sur Archiva

exo1-usine-logicielle est un projet Maven. Aussi, Maven impose (par convention), un fichier et une arborescence :

- pom.xml : à la racine du projet le POM (Project Object Model) qui décrit le projet, ses dépendances et son build.

- src/main/java : Sources Java du code principal
- src/main/resources : Ressources (configuration, images, etc...) utilisées pour l'exécution du code principal
- src/test/java : Sources des Tests
- src/test/resources : ressources utilisées pour l'exécution des tests

- target/classes : répertoire de génération des classes contenues dans src/main/java
- target/resources : répertoire contenant les ressources pour l'exécution principale
- target/test-classes : répertoire de génération des classes contenues dans src/test/java
- target/test-resources : répertoire contenant les ressources pour l'exécution des tests

NB : Les projets Maven peuvent avoir une arborescence plus complexe selon l'archetype Maven du projet. Par exemple, un projet Maven d'une application Web Java, contiendra un répertoire src/main/webapp.

1/ Créer une classe Personne dans un package org.plume.envol2012.usine.exo1 :

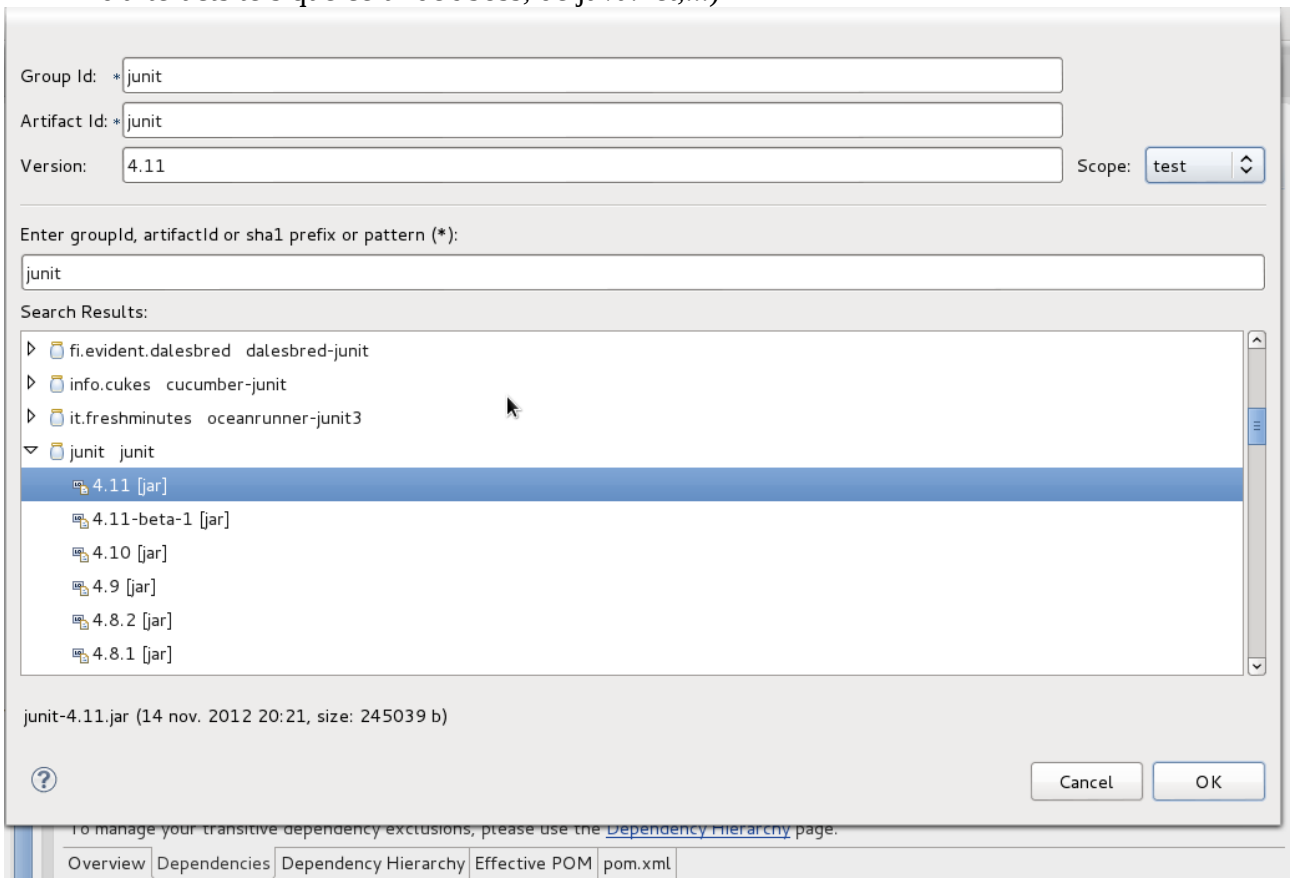
```
package org.plume.envol2012.usine.exo1;

public class Personne {

	private String nom;

	private String prenom;

	public Personne(String nom, String prenom) {
		this.nom = nom;
		this.prenom = prenom;
	}

	public String getNom() {
		return nom;
	}

	public void setNom(String nom) {
```

```
		this.nom = nom;
	}

	public String getPrenom() {
		return prenom;
	}

	public void setPrenom(String prenom) {
		this.prenom = prenom;
	}

}
```

2/ Ecrire un test unitaire sur le constructeur de cette classe.

Avant tout, nous avons besoin d'un framework de test unitaire : prenons JUnit. Il s'agit d'une dépendance. Inutile d'aller la télécharger et de l'inclure manuellement dans le classpath du projet ; déclarons simplement cette dépendance dans le POM et laissons Maven faire le travail !

- Ouvrir le fichier pom.xml
- Aller sur l'onglet « dependencies »
- Cliquer sur le bouton « Add » du tableau dépendancies
- Taper junit... (Maven va chercher, par défaut, parmi tous les artéfacts existants sur http://search.maven.org/ ; dans le POM, il est possible d'ajouter d'autres repositories d'artefacts tels que celui de Jboss, de java.net,...)



- Sélectionner l'artefact portant le groupId « junit » et l'artefactId « junit » en version 4.11
- Choisir le Scope « Test ». Il existe principalement 3 scopes :
  - compile : indique que cette dépendance est nécessaire à la compilation du code principal et à son exécution

    - provided : indique que cette dépendance est nécessaire à la compilation du code principal mais qu'elle sera fournie par un autre composant (un serveur d'application par exemple) pour son exécution ; cette dépendance ne sera donc pas incluse dans le packaging
    - test : indique que cette dépendance est nécessaire à la compilation et à l'exécution des tests ; cette dépendance ne sera donc pas incluse dans le packaging.
- Cliquer sur OK et sauvegarder
- 2 dépendances ont du apparaître dans Maven Dependencies : junit-4.11 qui est la dépendance déclarative que nous avons ajouté et hamcrest-core-1.3 qui est une dépendance de junit-4.11 (et donc une dépendance transitive pour nous). On peut également voir l'ensemble des dépendances dans l'onglet Dependency Hierarchy du POM.

Ecrire la classe de test.

- Dans src/test/java, créer la classe PersonneTest dans le package org.plume.envol2012.usine.exo1 :

```
package org.plume.envol2012.usine.exo1;

import static org.junit.Assert.*;
import org.junit.Test;

public class PersonneTest {

	@Test
	public void testConstructeurPersonne() {
		final String NOM = "ROUSSE";
		final String PRENOM = "David";
		Personne personne = new Personne(NOM, PRENOM);
		assertEquals(NOM, personne.getNom());
		assertEquals(PRENOM, personne.getPrenom());
	}

}
```

Exécuter le test en cliquant droit sur la classe de test, puis Run-As, Junit Test.

3/ Exécuter le build en local.
Même si nous n'avons encore rien précisé sur la manière de « builder » ce projet, Maven gère déjà dans son cycle par défaut : la compilation, l'exécution des tests, le packaging, le déploiement dans un repository.

Dans un terminal, se positionner dans ~/TP_UsineLogicielle/workspace/exo1-usine-logicielle :

- Compilation : exécuter mvn clean compile
- Tests : exécuter mvn clean test
- Packaging : exécuter mvn clean package
- Deploiement dans le repository local (~/.m2/repository) : exécuter mvn clean install. Ce repository contient tous les artefacts téléchargés précédemment et tous les artefacts construits en local par la tâche « install ». Tous les autres projets Maven locaux peuvent donc maintenant inclure cet artefact comme dépendance.

Remarquez le message de Warning « Using platform encoding (UTF-8 actually) to copy filtered ressources i.e. Build is platform dependant ! »
En effet, une bonne pratique de Maven est que le build soit indépendant de la plateforme, ce qui

n'est pas le cas. Il indique ici que, s'il y avait des fichiers dans les répertoires resources et qu'il avait du « parser » ces fichiers (par exemple pour subsituer des variables, cas d'utilisation le plus fréquent), il ne lui a pas été précisé quel encodage utilisé. Dans notre exemple simple, ce n'est pas gênant, mais ce n'est pas « propre ».

- Ajoutons donc dans le POM cette information en incluant :

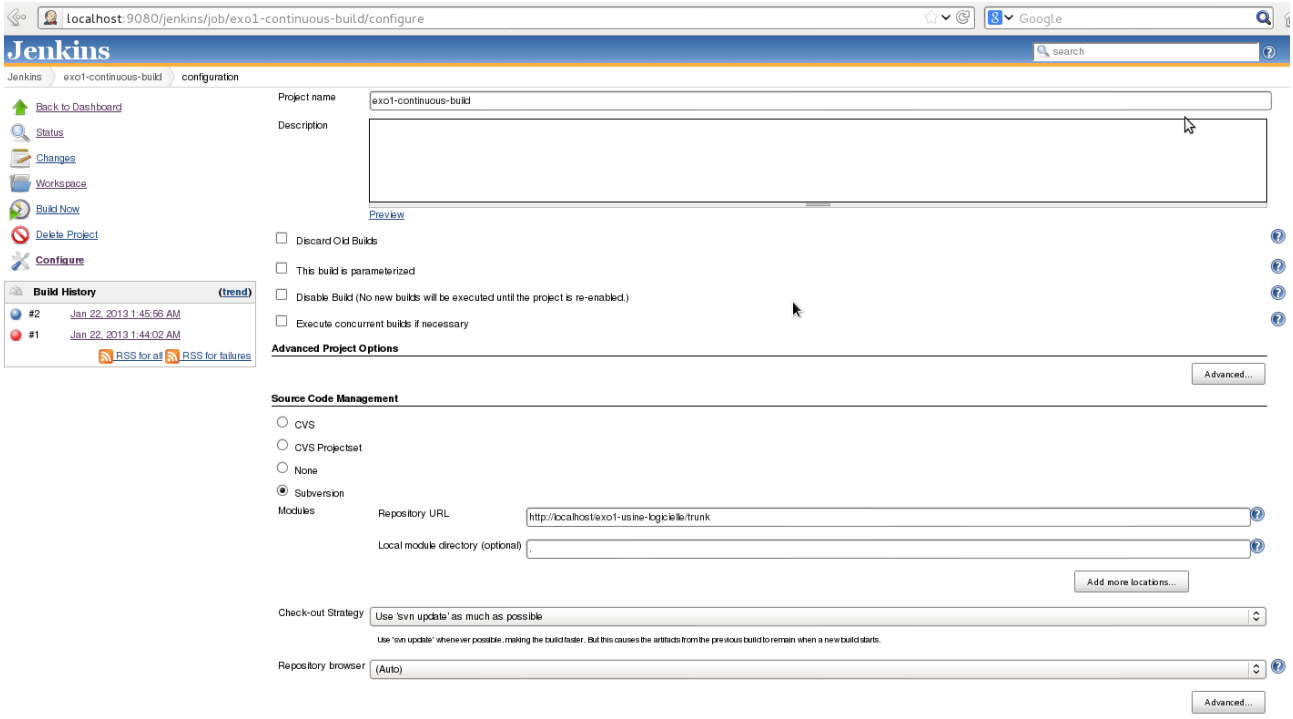
```
<properties>
            <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
```

NB : les tâches du build peuvent aussi être exécutées depuis Eclipse sur un clic droit sur le projet puis Run As → Maven... ou, avec toutes les options possibles, dans le menu Run → Maven Build...

- Bien, notre projet compile et se package en local, nous pouvons le commiter !
- Effectuer un clic droit sur le projet puis Team → Commit
- Saisir un message de commit puis cliquer sur OK.

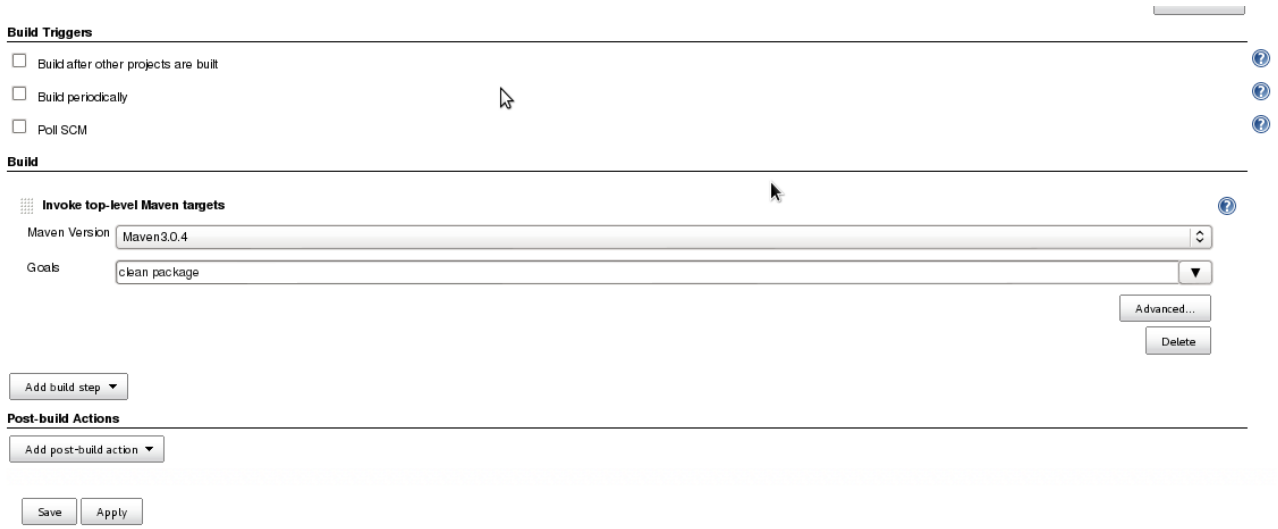
4/ Créer un job Jenkins qui exécute le build sur le serveur d'intégration continue

- Accéder à http://localhost:9080/jenkins
- En haut à gauche, cliquer sur New Job
- Saisissez le nom de job « exo1-continuous-build »
- Sélectionner « Build a free-style project software »
- Cliquer sur OK



Dans la configuration :

- sélectionner Subversion comme gestionnaire de sources et sélectionner l'URL du trunk dans le dépôt http://localhost/exo1-usine-logicielle/trunk
- dans la section « Build », cliquer sur « Add build step » et sélectionner « Add top-level Maven targets »
- sélectionner « Maven 3.0.4 » dans la liste Maven version
- saisir « clean package » dans le champ goals et cliquer sur « Save »



Le job est créé ! Il va simplement récupérer les sources dans le trunk SVN et exécuter mvn clean package. Vérifions :

- Sur la page principale, cliquer sur « Schedule a build » sur la ligne du job « exo1-continuous-build »... un joli soleil devrait apparaître !
- Cliquer sur le job, dans le tableau (à gauche) « Build history », sélectionner la dernière exécution
- Remarquer que les binaires sont disponibles en cliquant sur Module Build
- Cliquer également sur « Test result » pour consulter le rapport de test du build

Notre serveur d'intégration continue est donc capable maintenant de builder (compiler, exécuter les tests, packager) le trunk du projet, d'archiver chaque exécution, chaque rapport de tests unitaires et chaque binaire construit !

5/ Le job est créé certes mais pour le moment nous le lançons manuellement, ce qui a assez peu d'intérêt ! Automatisons son déclenchement à chaque modification de code, c'est-à-dire à chaque commit !

- Sur jenkins, reprendre la configuration du Job et dans la section « Build triggers », cocher la case « Poll SCM » (il est possible ici de saisir une expression cron pour indiquer à Jenkins de scruter le dépôt SVN à une fréquence donnée. Cependant, la bonne pratique est que ce soit le commit le déclencheur. Ne rien saisir dans ce champ).
- Cliquer sur Save

La manière de déclencher un build sur chaque commit est relativement simple : elle se base sur le hook post-commit de SVN qui contiendra une requête qui sera transmise à Jenkins.

Dans ~/TP_UsineLogicielle/svn/exo1-usine-logicielle/hooks, éditer le fichier post-commit et saisir les lignes suivantes :

```
#!/bin/sh

REPOS="$1"
REV="$2"
UUID=`svnlook uuid $REPOS`
/usr/bin/wget \
  --header "Content-Type:text/plain;charset=UTF-8" \
  --post-data "`svnlook changed --revision $REV $REPOS`" \
  --output-document "-" \
```

```
--timeout=2 \
http://localhost:9080/jenkins/subversion/${UUID}/notifyCommit?rev=$REV
```

Concrètement, chaque commit Subversion effectuera une commande wget sur Jenkins avec en paramètre l'UUID du dépôt et le numéro de révision ; Jenkins déclenchera alors automatiquement tous les jobs qui sont configurer pour « écouter » les commits de ces dépôts.

Vérifions cela... :

- Dans eclipse, saisir une modification sans impact (un saut de ligne par exemple) puis enregistrer
- Effectuer un commit
- … s'assurer que le Job Jenkins s'est bien déclenché « tout seul » !

Note : une action post-build à ajouter dans la configuration est la notification par mail à l'équipe en cas d'instabilité ou d'erreur du build.

6/ Déclencher une analyse Sonar.
Nous avons maintenant un serveur d'intégration continue qui sait déclencher des jobs sur commit. Ces jobs peuvent exécuter énormément de tâches différentes, un build (Maven, Ant, Make) n'est qu'une partie de ce qu'il peut faire. Naviguer dans Manage Jenkins → Manage Plugins → Avalaible pour avoir une idée des différentes tâches automatisable (déploiement sur un serveur ou un cloud, mail, publications, analyses, rapports, copier/transfert de fichiers, messagerie instantanée, archivage sur un repository...)

- Dans Jenkins, reprendre le job exo1-continuous-build et dans sa configuration, ajouter une action post-build (Add post-build action) et sélectionner Sonar puis Sauver. (L'installation du plugin nécessaire et la configuration initiale du lien Jenkins → Sonar a déjà été faite préalablement dans Manage Jenkins → Manager System)
- Relancer un build.
- Accéder à Sonar sur http://localhost:9000 pour voir l'analyse du programme
- Que remarque t-on sur la qualité sur ces quelques lignes de codes pourtant très simples ?

7/ Faire en sorte que le build publie les artefacts sur les repositories Maven

Maven, dans son cycle, gère aussi le déploiement des artefacts sur un repository distant (pas uniquement le repository local dans ~/.m2/repository). Cela se réalise par la commande mvn clean deploy. Néanmoins, l'upload d'artefact sur un repository est soumis à une authentification. La commande « deploy » a donc besoin des URLs des repositories où archiver les artefacts et d'un moyen de s'y authentifier. Nous allons fournir ces informations.

- Dans Eclipse, éditer le POM en allant dans l'onglet pom.xml et ajouter les lignes :

```
<distributionManagement>
        <repository>
            <id>internal</id>
            <name>Repository Maven des Releases</name>
            <url>http://localhost:8080/archiva/repository/internal</url>
        </repository>
        <snapshotRepository>
            <id>snapshots</id>
            <name>Repository Maven des Snapshots</name>
            <url>http://localhost:8080/archiva/repository/snapshots</url>
        </snapshotRepository>
```

```
    </distributionManagement>
```

<distributionManagement> indique là où doivent être archivés les artefacts du projet. <repository> indique le repository devant inclure les versions « released » et <snapshotRepository> le repository devant inclure les versions snapshots.

Grâce à ces informations, Maven sait maintenant vers quelles URLs (ie quels repositories) il doit envoyer les artefacts lors d'un appel à la tâche deploy. Par contre, il faut encore qu'il est un couple login/password pour pouvoir s'y authentifier. Cela se fait généralement dans le fichier settings.xml du répertoire .m2 du dossier personnel du compte utilisateur (notamment pour que chaque développeur puisse y enregistrer le sien). On va faire correspondre un couple login/password pour chaque <id> de repository dans ce fichier.

- Editer le fichier ~/.m2/settings.xml et ajouter les lignes :

```
<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">
  <servers>
        <server>
                <id>internal</id>
                <username>admin</username>
                <password>adm1n</password>
        </server>
        <server>
                <id>snapshots</id>
                <username>admin</username>
                <password>adm1n</password>
        </server>
<servers>
</settings>
```

- Configurer le build Jenkins, remplacer la commande Maven du Job par clean deploy
- Exécuter le build.
- Accéder à Archiva : http://localhost:8080/archiva et parcourir le repository ; on doit y trouver désormais exo1-usine-logicielle-0.0.1-SNAPSHOT.jar dans le repository snapshots

Essayons maintenant de créer une release.

- Dans le projet Eclipse, editer le POM et saisir « 1.0.0 » dans le champ version
- Effectuer un commit
- On doit trouver maintenant sur Archiva la version 1.0.0 de notre programme dans le repository internal

Synthèse :

Nous avons donc vu, sur un exemple très simple :

- Comment créer un projet Maven dont le build gère la compilation, l'exécution des tests, le packaging
- Comment créer un Job Jenkins exécutant ce build sur le serveur d'intégration continue
- Comment créer un hook SVN permettant de déclencher automatiquement les builds des projets sur les commits des développeurs

- Comment enchaîner le build par une analyse de code

Partie 2 : Utilisation avancée

Dans cette partie nous allons voir :

- La notion de profile Maven
- Exécution de test d'intégration
- Déploiement automatisé sur Tomcat

Dans le projet exo2-usine-logicielle, nous allons créer une webapp.

Si le temps le permet, nous ferons cette partie en mode « interactif ».