Ateliers Maven

TP1 : Installation de Maven et Projet simple

Objectif

Ce premier projet permet de créer le plus petit projet Java (Hello world) avec Maven. Il permet de se familiariser avec le fichier pom, les commandes Maven principales et la notion de repository local. Un site de documentation est également généré.

Installation maven

Télécharger une distribution de Maven et la décompresser dans un répertoire de travail (*c:\Formation\Maven, /home/<user>/Maven*)

Positionner les variables d'environnement suivantes :

`M2_HOME`

`M2=$M2_HOME/bin`

`MAVEN_OPTS (par exemple " -Xms256m -Xmx512m ")`

`PATH=$M2:$PATH`

Vérifier que JAVA_HOME pointe vers la bonne version de Java

Lancer `mvn -version` pour vérifier le bon fonctionnement

Personnalisation settings.xml

Nous voulons personnaliser l'emplacement du repository local où Maven télecharge tous les artefacts et déclarer un nouveau groupe de plugin.

Créer un fichier *${user.home}/.m2/settings.xml* à partir de *$M2_HOME/conf/settings.xml*

Renseigner la balise *<localRepository/>* à la valeur voulue (*c:\Formations\maven\repository /home/<user>/Maven/repository*)

Ajouter dans <pluginGroups>, la ligne suivante :

<pluginGroup>fr.jcgay.maven.plugins</pluginGroup>

Création d'un projet simple

1. Créer un projet dénommé «*hello*» en utilisant le plugin ***archetype*** et en indiquant des valeurs appropriées pour :
   - *groupId*
   - *packageName*

Par exemple :

```
mvn archetype:generate -DgroupId=org.formation -DartifactId=hello -
    DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4 -
    DinteractiveMode=false
```

2. Observer la structure du répertoire créé, ainsi que les fichiers de configuration et ceux du repository local. Quelle est la version du plugin maven-install utilisée ? Quel code source a été généré ?
3. Editer le fichier *pom.xml* pour ajouter les informations suivantes :
   - Une description du projet qui inclut la propriété ***file.encoding*** de Java
   - Le type de licence
   - L'organisation et une URL associée
   - Un développeur

   Utiliser le plugin d'aide pour afficher le pom effectif utilisé par maven et vérifier la résolution de la propriété.
4. Exécuter ***mvn install*** une première fois et observer les fichiers générés dans le répertoire projet.
5. Exécuter le programme généré avec la commande :
   java -cp target/simple-1.0-SNAPSHOT.jar <package-name>.App
6. Générer la documentation du projet avec ***mvn site*** puis parcourir le site

TP2 : Dépendances

Objectif

Ce TP permet de voir les techniques classiques de gestion des dépendances :

- Déclaration basique des dépendances dans le pom projet
- Utilisation d'un POM parent mutualisant les librairies communes et les n° de version
- Utilisation de groupe de dépendances

Etapes

1. Créer un projet dénommé « *simple-weather* » en utilisant le plugin ***archetype*** en indiquant *org.formation.weather* pour *package-name*, 1.0 pour la version et une valeur appropriée pour *groupId.*
2. Supprimer les fichiers sources générés et copier les sources et ressources fournies sans copier les tests pour l'instant
3. Editer le fichier *pom.xml* pour ajouter les informations complémentaires habituelles
4. Le code source fourni a des dépendances sur les librairies publiques suivantes :
   - Dom4J 1.6.1 et Jaxen 1.1.1 pour le parsing XML
   - Velocity 1.5 pour formater l'affichage
   - Log4j 1.2.14 pour les traces générées par l'exécution

   Modifier le fichier *pom.xml* pour ajouter ces dépendances en allant chercher les bons *groupId* et *artifactId* sur le site http://www.mvnrepository.com
5. Exécuter ***mvn install*** et si le build réussi, exécuter le programme grâce au plugin exec :
   mvn exec:java -Dexec.mainClass=org.formation.weather.Main
   Ce plugin permet d'exécuter la classe principale avec le classpath des dépendances Maven.
6. Copier les tests fournis et faire en sorte que tous les tests s'exécutent correctement.

Utilisation d'un POM parent

Afin d'éviter de répliquer dans nos différents projets les mêmes informations, nous allons mutualiser les informations communes dans un POM parent.
Ces informations mutualisées sont :

- Les informations liées à l'organisation
- La dépendance sur Junit
- Les numéros de versions des autres dépendances

Créer un dossier '*parent-project*' au même niveau que le dossier projet. Créer le fichier *pom.xml* dans *parent-project* qui reprend les informations liées, définit la dépendance sur JUnit et positionne les versions des autres dépendances.
Modifier ensuite le *pom.xml* de *simple-weather* afin qu'il fasse référence au POM parent.

Vérifier votre pom effectif, puis lancer ***mvn clean install***

Groupe de dépendances

Les librairies dom4j et jaxen sont utilisées pour tout projet désirant faire du parsing XML. Nous décidons donc de regrouper ces 2 librairies dans un groupe de dépendance afin de facilement les référencer.

Créer un dossier *xml* au même niveau que le projet parent et créer un nouveau fichier pom.xml avec comme type de package : « pom »

Garder la référence au POM parent et Regrouper les dépendances sur les libraires XML.
Installer l'artefact dans le repository local.
Faire référence à cette dépendance dans le projet *simple-weather*

TP3 : Configuration cycle de vie, profiles

Objectif

Ce TP permet d'aborder les points suivants :

- Application de filtres sur le répertoire ressources
- Les profiles
- Assemblage pour la création d'une distribution

Gestion des ressources

Créer un répertoire *src/main/velocity* et déplacer le fichier *output.vm*
Ajouter ce nouveau répertoire ressources dans la configuration

Externaliser les informations du niveau de trace présent dans *log4j.properties* dans un autre fichier *src/main/dev.properties*
Définir ensuite le filtre nécessaire dans la configuration.

Configuration du compilateur

Configurer le plugin compiler afin que la compilation soit compatible avec java 5 et que les classes produites contiennent des informations pour le debugger.

Création d'un profile

Configurer un profile « production » qui surcharge la configuration par défaut.

En particulier, il appliquera un nouveau filtre (*src/main/prod.properties*) au fichier de ressources et modifiera la configuration du compilateur :

- Ne plus inclure les informations de debug dans les classes
- Demandeur au compilateur d'optimiser le code
- Afficher les avertissements lors de la compilation
- Être en mode verbeux

Création d'une distribution

Lors de l'activation du profile « *production* », nous voulons associer la création d'une distribution à la phase package.

Pour cela, configurer le plugin *Assembly*.

Exécuter *mvn clean package* et vérifier la génération de la distribution.

La distribution n'est cependant pas exécutable, nous devons fournir les informations nécessaires au plugin pour générer un fichier *MANIFEST.MF* contenant la référence à la classe principale.

Ajouter les balises :

```
<archive>
  <manifest>
     <mainClass>org.maven.myapp.Main</mainClass>
  </manifest>
</archive>
```

Tester l'exécutable généré avec la commande

java -jar target/simple-weather-child-1.0-jar-with-dependencies.jar

TP5 : Projets multi-modules

Dans ce TP, nous allons mettre en place un projet Maven multi-modules qui produit deux applications :

- Un outil utilisable via une commande en ligne permettant de questionner la météo Yahoo
- Une application web qui propose une interface web pour faire la même chose.

Les deux applications stockent les résultats dans une base de données et utilisent la même logique applicative et de persistance.

Le projet multi-modules est donc composé de cinq modules :

- ***module-model*** : Un module modèle décrit les objets métiers des applications. Il utilise les annotations Hibernate pour automatiser la persistance. Le module est référencé par les deux applications
- ***module-service*** : Un module service contient toute la logique pour récupérer les données de Yahoo et de parser le XML fourni. Il a une dépendance sur le projet *model* et il est référencé par les deux applications.
- ***module-persist*** : Ce module contient l'interface d'accès aux données de persistances (*Data Access Objects – DAO*). Les deux applications utilisent ce module et ce module a des dépendances sur le modèle, *Hibernate* et le framework *Spring.*
- ***module-web*** : L'application Web contient de contrôleur MVC Spring qui utilise le module service et le module DAO. Il a donc des dépendances directes sur les modules *module-service* et *module-persist* et une dépendance transitive sur *module-model*.
- ***module-command*** : La commande en ligne comme le module web a des dépendances directes sur les modules *module-service* et *module-persist* et une dépendance transitive sur *module-model*.

Objectifs

- Mettre en place une architecture multi-modules
- Découvrir le plugin Hibernate3

Etapes

Projet multi-module

Création d'un projet multi-modules nommé « *multi-module* ». Les informations complémentaires habituelles peuvent être reprise du projet parent des Tps précédents.

Module modèle

Création du module modèle : « *module-model* » .

1. Référence au parent
2. Ajout des dépendances vers Hibernate : *hibernate-annotations* et *hibernate-common-annotations*
3. Exclusion de *javax.transaction/jta* car non utilisé

Copier les sources fournis et Exécuter ***mvn install***

Module service

Création du module service : « *module-service* » . Reprendre le *pom.xml* du TP2 et ajouter une référence au projet parent ainsi qu'une dépendance vers le *module-model*.

Copier les sources fournis et Exécuter ***mvn install***

Module persist

Création du module de persistance « *module-persist*» .

1. Référence au parent
2. Ajout des dépendances vers Hibernate : *hibernate.3.2.5.ga*
3. Exclusion de *javax.transaction/jta* car non utilisé et pas disponible dans le repository central
4. Ajout de dépendance vers Spring : *org.springframework/spring_2.0.7*

Module web

Création du module web : « *module-web*» .

1. Référence au parent
2. Ajout des dépendances vers les modules ***service*** et ***persist***
3. Ajout de dépendances vers *Spring* et *velocity_1.5*
4. Ensuite nous définissions deux plugins (Jetty et Hibernate3) comme suit :

```
  <build>
    <finalName>simple-webapp</finalName>
    <plugins>
      <plugin>
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>maven-jetty-plugin</artifactId>
        <dependencies>
           <dependency>
             <groupId>hsqldb</groupId>
             <artifactId>hsqldb</artifactId>
             <version>1.8.0.7</version>
           </dependency>
        </dependencies>
      </plugin>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>hibernate3-maven-plugin</artifactId>
        <version>2.0</version>
        <configuration>
           <components>
             <component>
               <name>hbm2ddl</name>
               <implementation>annotationconfiguration</implementation>
             </component>
           </components>
        </configuration>
        <dependencies>
           <dependency>
             <groupId>hsqldb</groupId>
             <artifactId>hsqldb</artifactId>
             <version>1.8.0.7</version>
           </dependency>
        </dependencies>
      </plugin>
    </plugins>
</build>
```

Copier les sources fournis et exécuter ***mvn install***

2. Génération de la base de données. Pour cela, nous utilisons le plugin Hibernate3. Dans le répertoire du projet webapp, exécuter : ***mvn hibernate3:hbm2ddl***
3. Vérifier la création de la base de données dans le répertoire *${basedir}/data*
4. Exécution de Jetty et test de l'installation à l'URL

http://localhost:8080/module-web/weather.x?zip=60202

Module commande

Création du module de commande en ligne : « *module-command*» .

1. Référence au parent
2. Ajout de dépendances *Spring2.0.7*, *Velocity1.5*, *hsqldb_1.8.0.7* et les modules *service* et *persist*
3. Création de la distribution et exécution via la commande :
   java -cp target/module-command-jar-with-dependencies.jar org.formation.weather.Main
4. Exécuter dorénavant *mvn install* au niveau du projet *multi-module*
5. Refaire éventuellement un refactoring des POM pour gérer les numéros de versions dans les POM parents
6. Exécuter *mvn dependency:analyze* et éliminer si besoin les références directes non déclarés

TP6 Nexus : Dépôts d’artefacts

Démarrer un serveur Nexus via Docker

```
docker volume create --name nexus-data
docker run -d -p 8081:8081 --name nexus -v nexus-data:/nexus-data
sonatype/nexus3
```

Présentation des dépôts gérés par Nexus, en particulier ***maven-central, maven-public, maven-releases, maven-snapshot***

Effectuer un déploiement vers le dépôt des snapshots

Utiliser le plugin *Maven Release* pour effectuer une release complète

TP7 : Tests unitaires et d'intégration

Reprendre le projet ***product-service*** fourni.

Visualiser le *pom.xml*

7.1 Comprendre le projet et le plan de build

Comprendre le cycle de build, pour cela vous pouvez vous doter de l'excellent plugin buildPlan.

https://jeanchristophegay.com/en/posts/maven-build-plan/

Quelle plugin et quelle version par défaut sont utilisée à la phase maven ***test*** ?

Quelle version de *JUnit* utilise ce projet ?

7.2 Distinction unitaires et intégration

Sur le projet précédent, mettre en place une configuration Maven afin que les tests de la couche *controller* soit effectué dans la phase d'intégration.

7.3 Couverture des tests

Mettre en place JaCoCo et visualiser les rapports de couverture de test

TP8 : Analyse statique

8.1 CheckStyle

Configurer le précédent projet pour visualiser le rapport checkstyle des règles par défaut lors de ***mvn site***

8.2 Sonarqube

Télécharger et installer un Sonarqube.

Configurer le projet afin d'effectuer une analyse après la phase *verify*

TP9 : Intégration Jenkins

Installation

Déclaration d’outils

Utilisation dans Jenkinsfile