Formation Maven

1 Installation

Maven, comme Tomcat et ANT, s'installe simplement en dézippant une archive récupérée sur le site principal : http://maven.apache.org

Il existe à ce jour deux versions de Maven disponibles :

- 2.2.x : version stable
- 3.0.x : version stable récente, la totalité des plugins 'noyau' sont compatibles avec cette version. Certains plugins externes ne sont pas encore disponibles/compatibles. La compatibilité ascendante est assurée à 100% pour les fonctionnalités noyau.

Nous travaillerons au cours de cette séance avec une version 2.2.1, il sera spécifiquement fait mention si certaines fonctionnalités abordées sont différentes avec Maven 3.

Commencez par ouvrir une console (root) puis exécutez les instructions suivantes :

```
cd /usr/local
tar zxvf ~esup/Bureau/apache-maven-2.2.1-bin.tar.gz
ln -s apache-maven-2.2.1 maven
update-alternatives --install /usr/bin/mvn mvn /usr/local/maven/bin/mvn 0
```

Vérifiez que l'installation s'est bien déroulée en lançant la commande suivante :

```
mvn --version
Warning: JAVA_HOME environment variable is not set.
Apache Maven 2.2.1 (r801777; 2009-08-06 21:16:01+0200)
Java version: 1.6.0_24
Java home: /usr/lib/jvm/java-6-sun-1.6.0.24/jre
Default locale: fr_FR, platform encoding: UTF-8
OS name: "linux" version: "2.6.35-25-generic" arch: "i386" Family: "unix"
```

On remarque que Maven génère un avertissement à propos de la variable d'environnement JAVA_HOME qui n'est pas définie. Comme les commandes `java` et `javac` sont disponibles directement grâce au lien symbolique généré par la commande `update-alternatives`, ce n'est pas bloquant dans notre situation. Toutefois, certains plugins déduisent l'emplacement d'un binaire du JDK (par exemple `javadoc`) à partir de cette variable, il est donc recommandé de la spécifier :

- Soit dans l'environnement utilisateur (dans le fichier `.bashrc`, à faire pour tous les utilisateurs utilisant Maven)
- Soit directement dans le fichier `/usr/local/maven/bin/mvn`, il faudra veiller à reporter cette modification en cas de mise à jour de Maven

2 Mon premier projet

Nous allons dans un premier temps créer une arborescence pour notre projet, nous verrons par la suite en détail sa structure. Voici les commandes à exécuter dans un terminal (non root) :

```
mkdir -p esup-hello/src/main/java
gedit esup-hello/src/main/java/Main.java
```

Dans le fichier `Main.java`, nous allons écrire un code minimal et bien connu :

```
public class Main {

  public static void main(String [] args) {
    System.out.println("Hello World");
  }

}
```

Editons ensuite le fichier `pom.xml` qui servira de configuration à Maven :

```
gedit esup-hello/pom.xml
```

Remplissons-le avec le minimum permettant de définir un projet Maven :

```
<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>
  <groupId>org.esupportail</groupId>
  <artifactId>esup-hello</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <name>esup-hello</name>
</project>
```

Le projet est terminé, il ne reste plus qu'à lancer la commande Maven permettant sa construction. La première exécution peut être relativement longue puisque Maven va télécharger tous les modules dont il a besoin pour réaliser la construction du projet :

```
cd esup-hello
mvn install
```

Si tout s'est correctement passé vous devriez repérer les éléments suivants :

- Le fichier `Main.class` disponible dans le dossier `esup-hello/target/classes`
- Un JAR `esup-hello-1.0.0-SNAPSHOT.jar` disponible dans le dossier `esup-hello/target`
- Le même JAR déployé dans le dossier `~/.m2/repository/org/esupportail/esup-hello/1.0.0-SNAPSHOT`

Terminons ce premier projet en testant qu'il fonctionne correctement :

```
java -classpath ./target/esup-hello-1.0.0-SNAPSHOT.jar Main
```

3 Eclipse

Travailler en ligne de commande peut vite s'avérer fastidieux, d'autant que la plupart des développeurs utilisent déjà un environnement de développement intégré. Nous allons donc regarder comment configurer et utiliser le plugin m2eclipse dans Eclipse.

3.1 Configuration

Sur le bureau se trouve un raccourci vers une installation d'Eclipse où le plugin est déjà installé mais n'est pas encore configuré.

Une fois Eclipse démarré, fermez la page 'Welcome' pour accéder au plan de travail, puis choisissez dans le menu 'Window' l'option 'Preferences'.

Voici la liste des options à paramétrer :

- **General > Workspace** : vérifiez que l'encodage est bien forcé en UTF-8 pour tous les contenus (important sous Windows)
- **Java > Compiler** : vérifiez qu'Eclipse applique par défaut une vérification du code Java relative à la version 1.6 du JDK.
- **Java > Installed JREs > Execution Environnments** : pour les environnements J2SE-1.2, 1.3, 1.4, 1.5 et JavaSE-1.6 cochez l'installation courante du JDK (le JDK 1.6 est compatible avec toutes les versions précédentes)
- **Maven > Installations** : configurez une nouvelle installation pointant vers le dossier `/usr/local/maven` et activez la. Pour information, Eclipse embarque une version 3.x Snapshot de Maven.
- **Maven > POM Editor** : cochez l'option 'Open XML page in the POM editor by default'

Nous allons maintenant importer notre premier projet dans Eclipse. La suite de ces travaux pratiques présuppose une certaine habitude à l'utilisation d'Eclipse.

Créez un nouveau projet de type Java, faites le pointer vers notre dossier 'esup-hello' puis dans la configuration du projet (avant de finaliser sa création), réalisez les opérations suivantes :

- Supprimez le dossier `src/main/java` des sources
- Supprimez le JAR esup-hello-1.0.0.jar des librairies

Validez la création du projet.

3.2 Activation de Maven pour le projet

Pour activer la prise en charge de Maven dans la construction du projet, il suffit de faire un clic droit sur le projet et de choisir **Maven > Enable dependency management.**

Une fois cette option activée, Maven va automatiquement exécuter les opérations suivantes :

- Ajout du dossier `src/main/java` aux sources du projet (au sens Eclipse)
- Ajout du dossier `src/main/resources` aux sources du projet
- Ajout du dossier `src/test/java` aux sources du projet

- Ajout du dossier `src/test/resources` aux sources du projet
- Ajout d'une librairie J2SE-1.5
- Modification du Java Compiler pour utiliser une version J2SE-1.5

Toutes ces opérations sont liées à l'héritage des propriétés du super pom Maven . Malheureusement dans notre cas, certaines de ces présomptions (en particulier la version de Java à utiliser) sont erronées. Nous allons modifier ce comportement par défaut en surchargeant la configuration Maven dans notre fichier 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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.esupportail</groupId>
  <artifactId>esup-hello</artifactId>
  <version>1.0.0</version>
  <name>esup-hello</name>
  <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>
  </build>
</project>
```

Il reste à actualiser le projet pour refléter ces changements en faisant un clic droit sur le projet et en choisissant **Maven > Update project configuration**.

Vous pouvez valider la construction du projet en exécutant les tâches clean et install :

- **Clic-droit > Run as > Maven clean**
- **Clic-droit > Run as > Maven install**

3.3 Création d'un projet à l'aide de l'assistant

Notre premier projet a été créé partiellement à la main, puis importé dans Eclipse. Nous allons examiner la création complète d'un projet simple dans Eclipse :

- **New > Project**
- **Maven > Maven Project**
- Cocher la case **Create a simple project (skip archetype selection)**
- Choisir un **Group Id** (org.esupportail)
- Choisir un **Artifact Id** (esup-test, utilisé par Eclipse pour nommer le projet)
- Choisir une **Version** (1.0.0)
- Choisir un nom (utilisé par Netbeans pour nommer le projet)
- Valider

Le projet créé est sensiblement le même qu'à l'étape 3.2, il faut modifier son fichier pom.xml pour spécifier la version de Java utilisée et rafraichir les propriétés du projet.

On remarquera que la structure de répertoire créée est plus fournie que celle du tout premier projet avec l'apparition de dossiers `main/resources`, `test/java` et `test/resources`.

4 Dépendances

Nous allons reprendre le projet esup-hello et lui ajouter des fonctionnalités de logging à l'aide de la librairie log4j. Pour cela, nous allons commencer par éditer le fichier `pom.xml` et lui ajouter la section `dependencies` suivante (après la section `build`) :

```
<dependencies>
  <dependency>
    <groupId>commons-logging</groupId>
    <artifactId>commons-logging</artifactId>
    <version>1.1.1</version>
  </dependency>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.16</version>
  </dependency>
</dependencies>
```

Si tout s'est bien passé, l'enregistrement du fichier pom.xml a déclenché le rapatriement des librairies commons-logging et log4j vers le cache local et leur ajout au classpath du projet. Nous pouvons alors modifier notre fichier `Main.java` pour le faire ressembler à ceci :

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
public class Main {

	protected static Log log = LogFactory.getLog(Main.class);

	public static void main(String [] args) {
		log.info("Starting esup-hello...");
		System.out.println("Hello World");
	}
}
```

La dernière étape consiste en la création d'un fichier `log4j.xml`, celui-ci devant se trouver dans le classpath du projet, nous le créerons dans le dossier `src/main/resources` :

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

  <appender name="console-appender" class="org.apache.log4j.ConsoleAppender">
    <param name="encoding" value="UTF-8" />
    <layout class="org.apache.log4j.PatternLayout">
      <param name="conversionPattern"  value="%d{dd/MM/yyyy HH:mm:ss} - %m%n" />
    </layout>
  </appender>

  <root>
    <level value="info" />
    <appender-ref ref="console-appender" />
  </root>

</log4j:configuration>
```

L'exécution du projet (clic droit sur le fichier `Main.java` **> Run as > Java Application**) donne désormais un résultat différent avec une ligne supplémentaire dans la console.

5 Miroir, déploiement

Dans cette partie, nous allons examiner deux aspects fondamentaux de Maven :

- Miroir : capacité d'un 'client' Maven à s'adresser à un repository d'entreprise se chargeant du téléchargement et de la mise en cache des modules
- Déploiement : possibilité offerte aux développeurs de publier leurs modules dans un repository d'entreprise

Nous utiliserons pour cela le repository Nexus de Sonatype, préinstallé sur la machine virtuelle. Nexus gère les deux cas d'utilisation (miroir, déploiement) mais il est très important de distinguer ces deux fonctions pour des raisons que nous expliquerons plus loin.

5.1 Miroir

L'utilisation d'un miroir permet principalement deux choses :

- Travailler depuis un poste qui se trouverait sur un réseau privé et n'ayant pas d'accès direct à l'Internet
- Dans une entreprise limitée en débit vers l'Internet, le miroir permet d'éviter le téléchargement répétitif des mêmes modules par plusieurs développeurs, et d'accélérer les transferts

La configuration d'un miroir est propre au poste sur lequel le développeur travaille, il est donc logique de ne pas la trouver dans le fichier pom.xml d'un projet. Elle peut être réalisée à deux endroits distincts :

- Directement dans le fichier `/usr/local/maven/conf/settings.xml` : dans ce cas elle est globale à la machine et nécessite d'être recopiée à chaque nouvelle installation.
- Dans le fichier `~/.m2/settings.xml` : elle est alors limitée à l'utilisateur courant et indépendante de l'installation binaire Maven utilisée.

Nous allons créer et modifier le fichier ~/.m2/settings.xml (depuis une console avec `gedit`) :

```
<?xml version="1.0" encoding="UTF-8"?>
<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">

  <mirrors>
    <mirror>
      <id>local</id>
      <mirrorOf>central</mirrorOf>
      <name>Nexus</name>
      <url>http://localhost:9080/content/groups/public</url>
    </mirror>
  </mirrors>

</settings>
```

Une fois le fichier enregistré, il est nécessaire d'indiquer à Eclipse que la configuration utilisateur locale a changé :

Window > Preferences > Maven > User Settings > Update Settings

Enfin nous allons supprimer le cache local, et relancer la construction de notre projet :

```
rm -rf ~/.m2/repository
cd ~/esup-hello
mvn clean install
```

Comme lors de la toute première exécution, Maven va télécharger toutes ses dépendances ainsi que celles de notre projet, mais cette fois en passant par Nexus qui va automatiquement les mettre en cache. Vous pouvez parcourir le dépôt Nexus en vous y connectant (Firefox dispose d'un raccourci Nexus) avec le compte admin/admin et en choisissant le repository Public.

5.2 Déploiement

Nous souhaitons maintenant mettre à disposition d'autres développeurs notre module esup-hello en le publiant dans notre repository d'entreprise. Pour cela, nous devons dans un premier temps indiquer dans le fichier pom.xml de notre projet où nous allons le publier :

```
<distributionManagement>
  <repository>
    <id>local-nexus-release</id>
    <name>Nexus local (releases)</name>
    <url>http://localhost:9080/content/repositories/releases/</url>
  </repository>
  <snapshotRepository>
    <id>local-nexus-snapshot</id>
    <name>Nexus local (snapshots)</name>
    <url>http://localhost:9080/content/repositories/snapshots/</url>
  </snapshotRepository>
</distributionManagement>
```

Le fichier pom.xml ne contient jamais d'informations de sécurité permettant de publier dans un repository (le fichier pom.xml est distribué avec le projet). Pour cela il est nécessaire de réaliser un 'mapping' entre l'identifiant d'un repository dans le pom.xml avec des credentials placés dans le fichier settings.xml :

```
<servers>
  <server>
    <id>local-nexus-release</id>
    <username>admin</username>
    <password>admin</password>
  </server>
  <server>
    <id>local-nexus-snapshot</id>
    <username>admin</username>
    <password>admin</password>
  </server>
</servers>
```

Modifiez ensuite votre projet pour changer sa version en 1.0.0-SNAPSHOT. Lancez la tâche maven deploy. Lancez la une seconde fois et allez voir dans le repository correspondant le résultat. Puis remettez la version en 1.0.0 et tentez des déploiements successifs.

6 Parent / Module

Maven offre deux options intéressantes pour les développeurs travaillant en collaboration sur des projets complexes. La première est la possibilité de développer un méta-projet (de type POM) duquel hériteront une série d'autres projets. Il est ainsi possible de définir des propriétés, des dépendances et des comportements au niveau du projet parent qui seront partagés par tous les projets fils. Il reste bien entendu possible au niveau de chaque enfant de surcharger ce qui a été défini au niveau du parent.

La seconde possibilité offerte est celle des modules. La gestion de modules permet uniquement un raccourci au développeur, c'est-à-dire pouvoir exécuter une tâche Maven sur une série de projets qui sont indépendants.

Enfin il est très souvent courant de combiner ces deux fonctionnalités pour obtenir un méta-projet de type POM incluant lui-même des modules fils. Attention, il reste nécessaire de bien distinguer ce qui découle de l'utilisation d'un parent de ce qui vient d'un développement sous forme de module.

6.1 Création du projet esup-user

Pour la suite des démonstrations, nous allons utiliser un projet un peu plus complexe que notre esup-hello appelé esup-user. Ce projet est constitué de cinq modules distincts :

- esup-user : méta-projet de type POM parent de tous les autres modules
- esup-user-domain : module de type JAR regroupant les objets métier
- esup-user-dao : module de type JAR regroupant les fonctions d'accès aux données
- esup-user-service : module de type JAR regroupant les services métiers
- esup-user-web : module de type WAR proposant une application Web

Il est à noter que ces projets sont interdépendants au sens Maven (et Java) du terme :

- esup-user-dao dépend de esup-user-domain
- esup-user-service dépend de esup-user-dao (et de esup-user-domain)
- esup-user-web dépend de esup-user-service (et de esup-user-dao, et de esup-user-domain)

Dans une console, commencez par dézipper le projet :

```
cd
unzip Bureau/esup-user.zip
```

Il n'est pas possible de construire à la main un projet modulaire, en effet Eclipse ne gère pas de sous-projets directement inclus dans un autre projet (au sens filesystem). Le plugin Maven triche pour obtenir ce résultat : il crée un projet Eclipse par module (y compris le module parent) et gère ensuite des dépendances inter-projet (au sens Eclipse).

Cliquez sur **File > Import** et choisissez l'option **Maven > Existing Maven Projects** puis pointez sur le répertoire racine du projet esup-user. Si l'import se déroule correctement, vous devriez récupérer les cinq modules décrits plus haut.

Examinez le fichier `pom.xml` du module parent (esup-user) et notez les différences par rapport aux chapitres précédents :

- packaging de type 'pom'
- liste des modules fils

Examinez le fichier pom.xml d'un autre module (par exemple esup-user-domain) et repérez la section parent décrivant le père de ce module. Vous noterez également que certaines informations définies dans le module parent n'ont pas besoin d'être redéfinies dans le module fils (certaines dépendances, section `distributionManagement`, certaines propriétés et enfin certains plugins).

6.2 Déploiement dans Tomcat

Dans un premier temps, nous allons déployer l'ensemble de nos modules dans notre repository local ainsi que dans notre repository d'entreprise. C'est ici que la notion de module est importante, il suffit de réaliser ces opération sur le module père pour que Maven le décline sur chacun des modules fils (et dans l'ordre satisfaisant les différentes dépendances).

Clic-droit sur le projet esup-user **> Run as > Maven build…** puis choisissez la tâche `deploy`. Maven vous présente un compte-rendu module par module de l'échec ou du succès de l'opération.

Nous allons maintenant déployer notre module Web dans un Tomcat local. Pour cela, réalisez les opérations suivantes :

- Clic-droit sur le projet esup-user-web **> Properties > Tomcat**
- Cochez la case **Est un projet Tomcat**
- Tapez comme contexte `/esup-user`
- Tapez comme sous-répertoire racine `/target/esup-user-web-0.0.1`
- Validez et dans le menu en haut choisissez **Tomcat > Démarrer Tomcat**

Vous pouvez tester l'application en vous connectant à l'URL http://localhost:8080/esup-user

6.3 Modification du projet

Examinez la façon dont les dépendances sont définies, et modifiez le projet pour utiliser une version plus récente de Spring (3.0.5.RELEASE). Vérifiez que l'application fonctionne toujours après avoir lancé successivement les tâches `clean` et `package`.

7 Filtres et profils

7.1 Configuration du filtrage

Nous avons déjà vu qu'il était possible d'utiliser dans un fichier `pom.xml` des variables qui sont automatiquement remplacées lors de la construction du projet. Nous allons maintenant mettre en œuvre une fonctionnalité spéciale de Maven qui permet de filtrer certains fichiers en lien avec les propriétés définies dans le fichier `pom.xml`.

Les fichiers les plus à même d'être filtrés sont ceux présents dans le dossier `src/main/resources`, nous allons donc surcharger la définition de ce dossier dans le fichier `pom.xml` du projet parent :

```
<build>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <filtering>true</filtering>
    </resource>
  </resources>
  …
</build>
```

Nous pouvons dès lors utiliser des propriétés de la forme ${variable} dans les fichiers du répertoire src/main/resources. Commençons par le fichier log4j.xml du projet esup-user-web en rendant configurable le niveau et le type de log :

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

  <appender name="console-appender" class="org.apache.log4j.ConsoleAppender">
    <param name="encoding" value="UTF-8" />
    <layout class="org.apache.log4j.PatternLayout">
      <param name="conversionPattern" value="%d{dd/MM/yyyy HH:mm:ss} - %m%n" />
    </layout>
  </appender>

  <appender name="file-appender" class="org.apache.log4j.FileAppender">
    <param name="encoding" value="UTF-8" />
      <param name="file" value="${log4j.file}" />
      <layout class="org.apache.log4j.PatternLayout">
        <param name="conversionPattern" value="%d{dd/MM/yyyy HH:mm:ss} - %m%n" />
      </layout>
  </appender>

  <root>
    <level value="${log4j.level}" />
    <appender-ref ref="${log4j.appender}" />
  </root>

</log4j:configuration>
```

Nous avons défini trois variables :

- Le niveau de log `${log4j.level}`
- Le type d'appender `${log4j.appender}`
- Le fichier de log dans le cas d'un FileAppender `${log4j.file}`

Nous allons dans un premier temps définir ces nouvelles propriétés dans le fichier `pom.xml` du module `esup-user-web` :

```
<properties>
  <log4j.level>info</log4j.level>
  <log4j.appender>console-appender</log4j.appender>
  …
</properties>
```

Une fois ces modifications apportées, relancez la construction du projet et vérifiez que ces modifications sont bien prises en compte.

Modifions nos propriétés pour changer l'appender utilisé :

```
<properties>
  <log4j.level>info</log4j.level>
  <log4j.appender>file-appender</log4j.appender>
  <log4j.file>${catalina.base}/logs/${artifactId}.log</log4j.file>
  ...
</properties>
```

Notons au passage que Maven n'a pas connaissance de la variable `${catalina.base}` au moment où il construit le projet, et que cette variable ne sera donc pas remplacée. Par contre, log4j aura l'information au lancement de l'application via les variables d'environnement et pourra donc écrire son fichier dans le dossier `~esup/tomcat/logs`.

7.2 Définition d'un profil

L'application que nous avons définie peut être amenée à changer de configuration au cours de son cycle de vie. Par exemple en phase de développement, il peut être intéressant d'avoir un niveau de log en `debug`, et l'affichage directement dans la console. Par contre en production, il est plus judicieux d'avoir un niveau en `error` et un fichier de traces sur le serveur. Telle que notre application est paramétrée, le passage de l'un à l'autre de ces modes nécessite de modifier le fichier `pom.xml`. Nous allons donc définir un comportement par défaut de type production (niveau à `error` et fichier de trace) et un profil spécifique pour la phase de développement (niveau à `debug` et traces dans la console).

Commençons par reprendre notre fichier `pom.xml` pour lui ajouter notre nouveau profil :

```
<profiles>
  <profile>
    <id>devel</id>
    <properties>
      <log4j.level>debug</log4j.level>
      <log4j.appender>console-appender</log4j.appender>
    </properties>
  </profile>
</profiles>
<properties>
  <log4j.level>error</log4j.level>
  <log4j.appender>file-appender</log4j.appender>
  <log4j.file>${catalina.base}/logs/${artifactId}.log</log4j.file>
  …
</properties>
```

Nous n'avons surchargé dans notre profil que les variables spécifiques à celui-ci. Toutes les autres sont héritées du profil par défaut (les propriétés présentes de base dans notre fichier `pom.xml`).

Maven doit connaître le profil à activer lors de la construction du projet : clic-droit **> Run as > Maven build ...** :

- **Goals** : clean package
- **Profiles** : devel

Chaque tâche Maven exécutée doit désormais être accompagnée de l'activation explicite du profil `devel`, mais il est possible de conditionner son activation automatique (par exemple sur la présence d'un fichier à la racine du projet) :

```
<profiles>
  <profile>
    <id>devel</id>
    <activation>
      <file>
        <exists>devel</exists>
      </file>
    </activation>
    <properties>
      <log4j.level>debug</log4j.level>
      <log4j.appender>console-appender</log4j.appender>
    </properties>
  </profile>
</profiles>
```

Créez un fichier vide `devel` à la racine du projet esup-user et relancez un la tâche package, attention dans le cas d'un projet multi-module l'existence d'un fichier est testée par rapport au module père et non par rapport au module définissant le profil.

Nous allons pour terminer externaliser les propriétés liées à la configuration de notre application dans des fichiers distincts. Pour cela commençons par créer un fichier `default.properties` à la racine du module esup-user-web :

```
log4j.level=error
log4j.appender=file-appender
log4j.file=${catalina.base}/logs/${artifactId}.log
```

Créons également un fichier `devel.properties` au même endroit :

```
log4j.level=debug
log4j.appender=console-appender
```

Il reste à modifier notre fichier `pom.xml` pour supprimer la définition actuelle des propriétés log4j et pour déclarer nos filtres externes à la fois dans la balise `build` située à la racine et dans celle du profil spécifique :

```
<build>
  <filters>
    <filter>default.properties</filter>
  </filters>
  …
</build>
<profiles>
  <profile>
    <id>devel</id>
    <activation>
      <file>
        <exists>devel</exists>
      </file>
    </activation>
    <build>
      <filters>
        <filter>devel.properties</filter>
      </filters>
    </build>
  </profile>
</profiles>
```

8 Archétypes

Dans cette dernière partie, nous allons créer un archétype c'est-à-dire un méta-projet Maven qui servira de modèle pour la création de futurs projets du même type.

8.1 Création de l'archétype

Un archétype est un projet Maven de type JAR qui au départ ne se distingue en rien d'un autre projet. Commençons par créer un nouveau projet Maven dans Eclipse en utilisant l'assistant :

- Maven Project
- Create a simple project (skip archetype selection)
- Group Id : org.esupportail
- **Artifact Id** : esup-war-archetype
- **Version** : 1.0.0-SNAPSHOT
- **Packaging** : jar
- **Name** : esup-war-archetype
- Validez
- Supprimez le dossier src/test
- Supprimez le dossier src/main/java
- Créez un dossier src/main/resources/META-INF/maven

Le fichier de configuration principal d'un archétype s'appelle archetype.xml et doit se trouver dans ce dernier dossier créé, créons ce fichier :

```
<archetype xmlns="http://maven.apache.org/plugins/maven-archetype-
plugin/archetype/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/plugins/maven-archetype-
plugin/archetype/1.0.0 http://maven.apache.org/xsd/archetype-1.0.0.xsd">
  <id>esup-war-archetype</id>
</archetype>
```

Il est important que l'identifiant soit identique à l'`artifactId` du projet d'archétype.

La première chose à faire est de définir comme pour un projet classique où sera déployé notre archétype, nous devons donc modifier le fichier `pom.xml` de celui-ci :

```
<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>org.esupportail</groupId>
<artifactId>esup-war-archetype</artifactId>
<version>1.0.0-SNAPSHOT</version>
<name>esup-war-archetype</name>
<distributionManagement>
  <repository>
    <id>local-nexus-release</id>
    <name>Nexus local (releases)</name>
    <url>http://localhost:9080/content/repositories/releases/</url>
  </repository>
  <snapshotRepository>
    <id>local-nexus-snapshot</id>
    <name>Nexus local (snapshots)</name>
    <url>http://localhost:9080/content/repositories/snapshots/</url>
  </snapshotRepository>
</distributionManagement>
</project>
```

Nous devons maintenant définir quel sera le contenu du projet cible de notre archétype. Pour rappel, l'archétype permet de construire un projet Maven qui serait pré-rempli et pré-paramétré avec des choix par défauts. Le premier fichier important est donc le futur `pom.xml` du projet. Pour cela, nous allons créer un fichier `src/main/resources/archetype-resources/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>${groupId}</groupId>
<artifactId>${artifactId}</artifactId>
<version>${version}</version>
<packaging>war</packaging>
<name>${artifactId}</name>
<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>
</build>
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
</project>
```

Bien entendu, tout projet qui utilisera notre archétype aura sa propre identification Maven (`groupId`, `artifactId`, `version`) qui sera spécifié à la création du projet. C'est pourquoi on trouve ici des variables qui seront automatiquement remplacées lors de la construction.

Il ne reste plus qu'à déployer notre archétype dans notre repository local (dans un premier temps) afin qu'il soit disponible pour créer de nouveaux projets.

Clic droit **> Run as > Maven install**

8.2 Utilisation d'un archétype en ligne de commande

Nous allons temporairement utiliser la ligne de commande pour créer notre premier projet à partir de l'archétype construit juste au-dessus. Ouvrez une console (non root) et tapez les commandes suivantes :

```
cd
mvn archetype:generate -DarchetypeCatalog=local
```

Si tout s'est bien déroulé précédemment, vous devriez pouvoir choisir l'archétype `esup-war-archetype`. Vous devez alors saisir les informations concernant le projet que vous voulez construire à savoir son `groupId`, son `artifactId`, sa `version` et enfin le package Java racine.

Pour information, la création d'un projet à partir d'un archétype est faite en mode déconnecté, c'est-à-dire que si l'archétype vient à être mis à jour par la suite, les changements ne seront pas répercutés dans le projet qui en découle. Un projet créé par archétype ne garde aucune trace des particularités de sa création, et il est tout à fait possible de casser ce que l'archétype a construit par la suite.

8.3 Utilisation d'un archétype avec Eclipse

Pour créer un projet Maven dans Eclipse en utilisant un archétype voici les opérations à effectuer :

- **New > Project… > Maven > Maven Project**
- **Catalog** : Default Local
- Cochez la case **Include snapshot archetypes**
- Choisissez votre archétype
- Continuez la création comme un projet normal

A noter que par défaut, Eclipse utilise un archétype même si vous n'en spécifiez pas. Il y a un archétype simple pour chaque type de packaging (JAR, WAR, POM…).

8.4 Plus loin dans les archétypes

Nous avons jusque-là simplement créé un fichier `pom.xml` dans notre archétype. En fait, le dossier src/main/resources/archetype-resources d'un archétype peut contenir une structure similaire à un projet Maven standard et les fichiers qui correspondent (web.xml, log4j.xml, configuration Spring, images, filtres etc.).

Voici une liste d'amélioration à apporter à notre archétype :

- Filtrage automatique des ressources
- Définition du codage UTF-8 pour les sources
- Définition des paramètres de publication (distributionManagement)
- Ajout d'un fichier log4j.xml par défaut
- Ajout d'un fichier web.xml par défaut
- Définition des dépendances Spring (et de la version à utiliser)

Une partie de ces améliorations peuvent directement être apportées dans le fichier `pom.xml`. Pour toute inclusion de fichier supplémentaire, il est nécessaire de définir ces inclusions dans le fichier `archetype.xml` :

```
<sources>
  <source>src/main/java/Hello.java</source>
</sources>
<resources>
  <resource filtered="true">src/main/resources/log4j.xml</resource>
</resources>
```

A noter que les fichiers qui seront placés dans les ressources peuvent être filtrés par rapport à des propriétés définies dans le fichier `pom.xml` **de l'archétype**.

Attention : il semble qu'Eclipse n'aime pas la définition de ressources dans le fichier `archetype.xml` et plante lors de la création d'un projet s'appuyant sur un tel archétype. Cependant tout fonctionne correctement en ligne de commande.