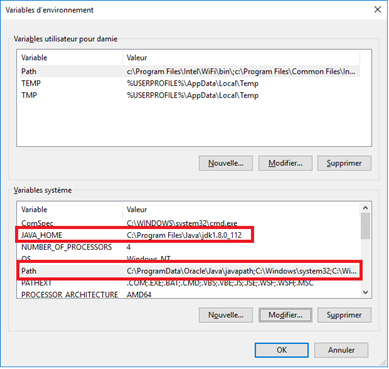
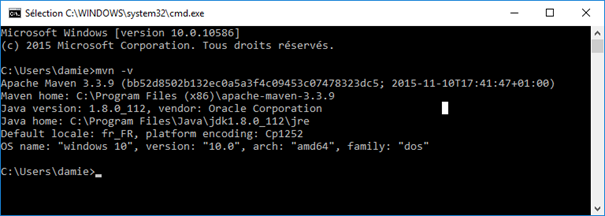
Notes du cours sur Maven le 10/01/2017  
  
Maven est un plugin (?) qui va permettre de centraliser les dépendances d’un projet, que ce soit les librairies utilisées ou les tests rattaché au développement. Si tous les développeurs sont sur le même projet Maven, ils ont l’assurance de tous utiliser les mêmes dépendances, avec la même version pour chacun. Cela permet une centralisation de la “configuration” d’un projet.  
L’utilisation vu en cours correspond à un projet en Java, mais il existe une version pour chaque langage. En Java, on importera Junit pour les tests, mais il s’agirait de Nunit pour du Dev en .NET par exemple. Pour fonctionner ici, Maven a besoin d’avoir accès au JDK installé sur la machine via les variables d’environnement. Il faut donc tout d’abord ajouter le dossier d’installation de Maven à la variable d’environnement PATH :

  
  
Pour s’assurer de la bonne configuration du projet, ouvrir un cmd et taper mvn -v. Si vous avez correctement configurer le PATH, voici ce qui doit s’afficher :   
  
Pour créer un projet géré par Maven sous Intellij, il suffit de créer un nouveau projet Maven avec l’assistant de création de projet intégré (fichier ⇒ nouveau ⇒ projet ⇒ Maven). choisir de créer via archetype, choisir l'archétype “quickstart”, groupId : com.esgi, artifactId : peu importe. Laisser la class App inchanger, et modifier la classe AppTest comme suit :

package com.esgi;

import org.junit.Test;

import static org.junit.Assert.assertTrue;

*/\*\**

*\* Unit test for simple App.*

*\*/*

public class AppTest

{

  @Test

  public void should\_assert\_true(){

*assertTrue*(true);

  }

}

package com.esgi;   représente le nom du package en cours de developpement (?)

import org.junit.Test;  permet d’utiliser la balise @Test (?)

import static org.junit.Assert.assertTrue; contient la methode assertTrue() qui permet de valider ou non le test (?)

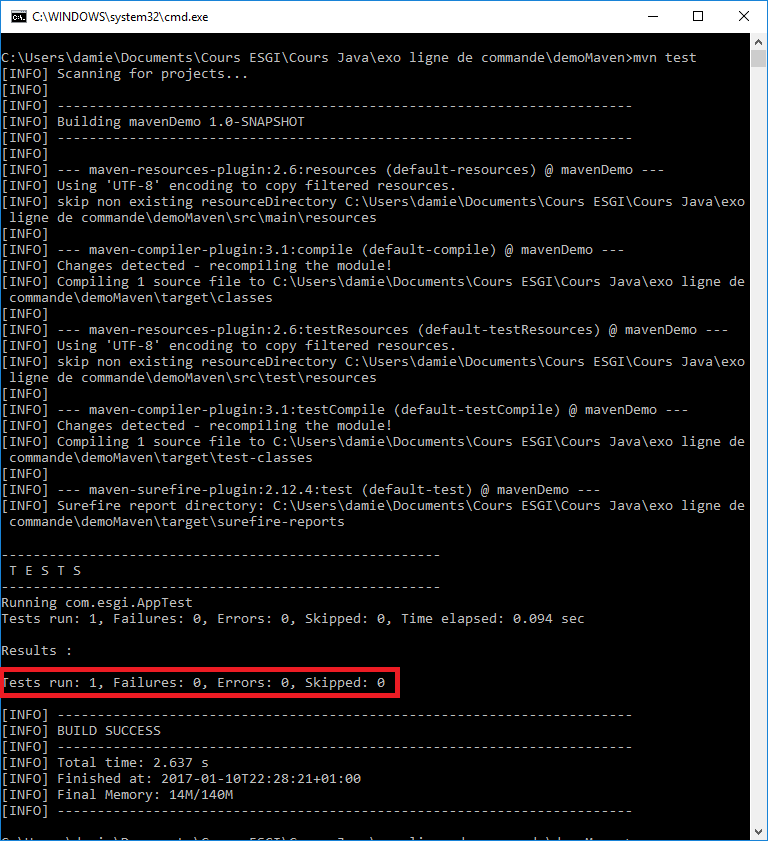
@Test est une balise. toute fonction déclaré après cette balise est incluse dans l’ensemble des test qui doivent être validés pour approuver  l’ensemble des test de Maven

*assertTrue*(true); doit contenir true(?). on mettra typiquement à l’interieur le retour d’une fonction dont on veut tester le comportement. si assertTrue reçoit false,  alors la phase de test de Maven notera une failure spécifiquement sur cette fonction.

Une fois Maven et le JDK installé, et que le projet est correctement implémenté, on peut commencer à utiliser Maven pour ce pourquoi on l’as conçu :

* valider : (?)
* compiler : appel javac et créer les .java de nos .class
* tester : exécute toutes les fonctions de test et récupère leur résultat, puis affiche un compte rendu. si une fonction de test échoue (assertTrue contient un false) ou si une exception est levé, la prochaine étape ne s'exécute pas.
* compresser : créer une archive .jar de notre projet. Sous cette forme, il peut être utilisé par un autre projet de la même manière que n’importe quelle autre package java grâce à un import .
* vérifier : vérifie l’intégrité du package créé (checksum et autre)
* installer : Copie l’archive .jar créé dans le dossier caché .m2 de Maven (?)
* déployer : (?)

Maven suis un cycle de vie, ce qui signifie que chaque étape lancé exécutera à chaque fois toutes les autres étapes précédente. Exemple : si je veut lancer une compression, Maven va d’abord valider, puis compiler, puis tester, et seulement après va compresser.

Pour lancer une étape : 

On peut voir plus haut que dans le résumé des test, 1 test a été lancé, dont 0 failure (assertTrue contenant false), 0 errors (exceptions levés), donc tout les test sont validés et mvn peut passer au packaging. Comme tous les test ont été effectué, lors du packaging, mvn ne relancera pas les test pour gagner du temp. il est donc recommandé d’effacer l’historique avec “mvn clean” avant de lancer un packaging. une fois le packaging terminé, un dossier target est créé, où on peut trouver le fichier .jar.

Ce .jar a été créé en regroupant toutes les dépendances du projet, défini dans le fichier pom.xml. il est donc possible de rajouter autant de paramètre voulue dans ce fichier, ce qui affecte le résultat ( le .jar). on retrouve dans ce fichier toutes les information relatives à Maven:

ce bloc regroupe les information de base du projet : c’est ce qui le permet de l’identifier

<groupId>com.esgi</groupId>

<artifactId>mavenDemo</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>jar</packaging>

ce bloc va renseigner le plugin utilisé par Maven pour mener les test et sa version (Java = Junit)

<dependencies>

<dependency>

  <groupId>junit</groupId>

  <artifactId>junit</artifactId>

  <version>4.12</version>

  <scope>test</scope>

</dependency>

</dependencies>

en l'état, le .jar généré n’est pas exécutable : il est seulement utilisable comme package, car son point d’entré (sa classe qui contient un public static void main) n’as pas été renseigné. il est possible de rajouter des information dans le fichier pom.xml pour que lors de la création de l’archive, le point d’entré de l’application soit pris en compte, et qu’on puisse donc exécuter le .jar avec une commande du type : “java App” pour obtenir un “hello world”. une rapide consultation du manuel (RTFM) nous propose de rajouter un bloc à notre fichier de configuration :

<build>

<plugins>

  <plugin>

    <groupId>org.apache.maven.plugins</groupId> le plugin se trouve ici

    <artifactId>maven-shade-plugin</artifactId> il est identifié ainsi

    <version>2.4.3</version> il est de tel version

    <executions>

      <execution>

        <phase>package</phase> il concerne la phase de compression

        <goals>

          <goal>shade</goal> nom du plugin(?)

        </goals>

        <configuration>

          <transformers>

            <transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer"> (?)

              <mainClass>com.esgi.App</mainClass> point d’entré

            </transformer>

          </transformers>

        </configuration>

      </execution>

    </executions>

  </plugin>

</plugins>

</build>

TODO :

* début du cours sur la centralisation des librairies j’ai rien pigé, à quelle moment ya une centralisation des dépandance, et comment plusieurs devs vont utiliser Maven pour s’assurer de tous avoir le meme environement de dev
* confirmer/préciser tous les marqueurs “(?)”
* fin du cours sur l’accession exclusive au fonction d’accès au BDD par plusieurs threads
* tout ce que j’ai oublié d’abroder sur Maven
* GIT EST EXCLUE DE CETTE NOTE, IL S’AGIT D’UN AUTRE COURS