Tutoriel pour bien débuter avec Maven et Java

Par Gokan EKINCI

Date de publication : 23 janvier 2015

Dernière mise à jour : 8 mai 2017

TOUT PUBLIC

Durée : 2h

Maven est un outil créé par Apache, il permet de **faciliter la gestion de son projet Java**. Maven est parfois qualifié de « **Dependency Management Tool** » (Outil de Gestion de Dépendance en français), c'est un outil qui est très utilisé dans l'univers Java, il est aussi très apprécié des professionnels.

Pour réagir au contenu de ce tutoriel, un espace de dialogue vous est proposé sur le forum.

Commentez

I - Présentation des qualités de Maven

Maven permet de **mieux structurer son projet Java**. En effet, lorsque nous créons un nouveau projet, Maven sépare la partie liée au code du projet, le code des tests unitaires et autres fichiers statiques (images, PDF etc.). L'ensemble du projet est géré à partir d'un seul fichier : le **pom.xml**. Le **pom.xml** représente le **cœur du projet Maven** car il contient toutes les informations liées au projet (nom du projet, auteur du projet, version du projet etc.) mais aussi les « dépendances » du projet.

Un développeur qui connaît Maven aura beaucoup plus de facilité à s'intégrer au contenu de votre projet.

Maven permet de gérer les JAR du projet à votre place, c'est ce qu'on appelle **la gestion des dépendances**. Un gros projet Java contient généralement plusieurs dizaines de JAR externes (ex : **postgresql-9.3-1100.jdbc4.jar**, **junit-4.11.jar**, etc.), ces JAR sont appelés des « **dépendances** » car nous avons besoin de les inclure dans nos projets. En précisant dans le fichier **pom.xml** les JAR que l'on souhaite inclure dans le projet, Maven va télécharger ces JAR à partir du repository Maven pour nous, il n'est donc plus nécessaire d'aller sur divers sites pour télécharger les JAR puis de les inclure nous-même dans le projet via l'IDE (P.-S. : C'est le même principe que le repository des distributions Linux et le téléchargement d'outils avec la commande « apt-get »). Imaginez juste le gain de temps que vous aurez lorsque vous n'aurez plus à vous prendre la tête pour mettre à jour une dépendance, lorsque vous n'aurez plus à dupliquer le code source des autres projets dans le projet courant…

Pour télécharger les JAR une première fois, Maven a besoin d'une connexion Internet. Une fois téléchargés, ces JAR sont placés dans le repository local (sur votre ordinateur) et récupérés à partir de ce dernier.

Maven va **compiler le projet Java** en se servant du compilateur officiel Java (javac) et lancer les tests pour vérifier la non-régression du projet, le tout sans qu'on ait à se casser la tête avec des commandes complexes ou des problèmes liés aux packages (sacrés packages !). Voyez l'exemple :

Outil	**Commande pour compiler des fichiers .java en .class**
javac	javac package1/*.java package2/*.java package3/*.java packageN/*.java
Maven	mvn compile

II - Installation de Maven

Pour installer et utiliser Maven, vous devez avoir :

- un JDK sur votre machine (au moins JDK7 pour Maven 3.3, JDK6 pour Maven 3.2, et JDK5 pour Maven 3.0 et 3.1) ;
- 10 Mo d'espace libre pour le téléchargement du zip et au moins 500 Mo pour le téléchargement des JAR utilitaires (prévoyez 600 Mo de libres).

*Ce tutoriel a été testé sous Linux (Ubuntu 15.04) et Windows (7 et 10) avec les JDK de Sun/Oracle 7 et 8. Notez que si vous ne disposez pas d'un dossier **/jdk** dans votre répertoire d'installation Java, cela signifie très certainement que vous avez juste une JRE d'installée sur votre machine. Voici **le lien** pour installer le JDK8 d'Oracle.*

Quel que soit votre système d'exploitation, téléchargez une version récente du **.zip** ou **.tar.gz** de Maven, seule l'extension du fichier compressé diffère, leur contenu est **identique** (**lien téléchargement**). Notez que pour ce tutoriel, nous installerons la dernière version stable de Maven, soit la version **3.3.9** à l'heure où j'écris ce tutoriel. Pour ceux qui utilisent Ubuntu, il vous est possible d'installer Maven via le dépôt APT avec la commande `apt-get install maven`, je n'utiliserai **pas** cette méthode car je souhaite garder le contrôle sur la **version** et le **répertoire d'installation** de Maven.

	Link	Checksum	Signature
Binary tar.gz archive	apache-maven-3.3.9-bin.tar.gz	apache-maven-3.3.9-bin.tar.gz.md5	apache-maven-3.3.9-bin.tar.gz.asc
Binary zip archive	apache-maven-3.3.9-bin.zip	apache-maven-3.3.9-bin.zip.md5	apache-maven-3.3.9-bin.zip.asc
Source tar.gz archive	apache-maven-3.3.9-src.tar.gz	apache-maven-3.3.9-src.tar.gz.md5	apache-maven-3.3.9-src.tar.gz.asc
Source zip archive	apache-maven-3.3.9-src.zip	apache-maven-3.3.9-src.zip.md5	apache-maven-3.3.9-src.zip.asc

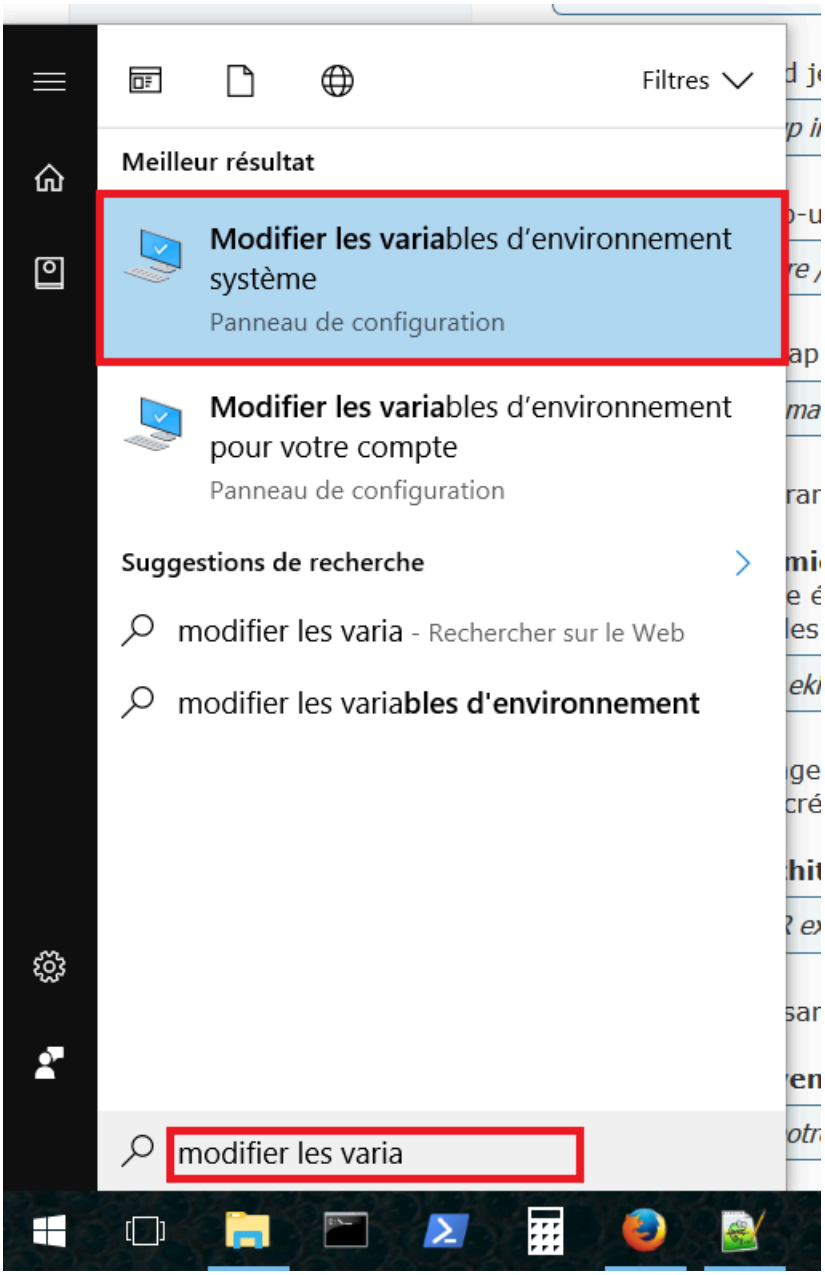
Décompressez le fichier téléchargé sur votre machine, voici quelques répertoires conseillés :

- pour **Linux : /opt/apache-maven-3.3.9**
- pour **Windows : C:\Program Files\Apache Software Foundation\apache-maven-3.3.9**

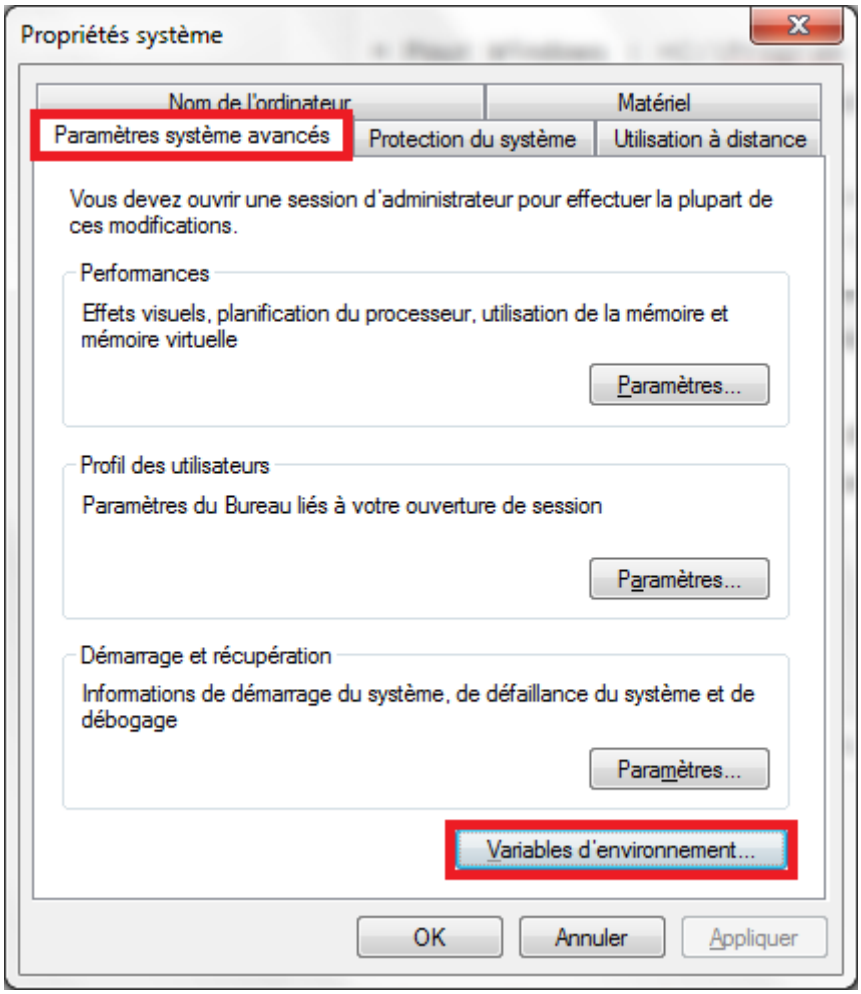
L'idée est maintenant de pouvoir utiliser Maven en ligne de commande, pour cela je vais vous montrer comment faire pour ajouter une variable d'environnement.

Pour les Linuxiens, ajoutez la variable d'environnement `JAVA_HOME` pointant vers le répertoire de votre JDK puis mettez `export PATH=/opt/apache-maven-3.3.9/bin:$PATH` dans *~/.profile*

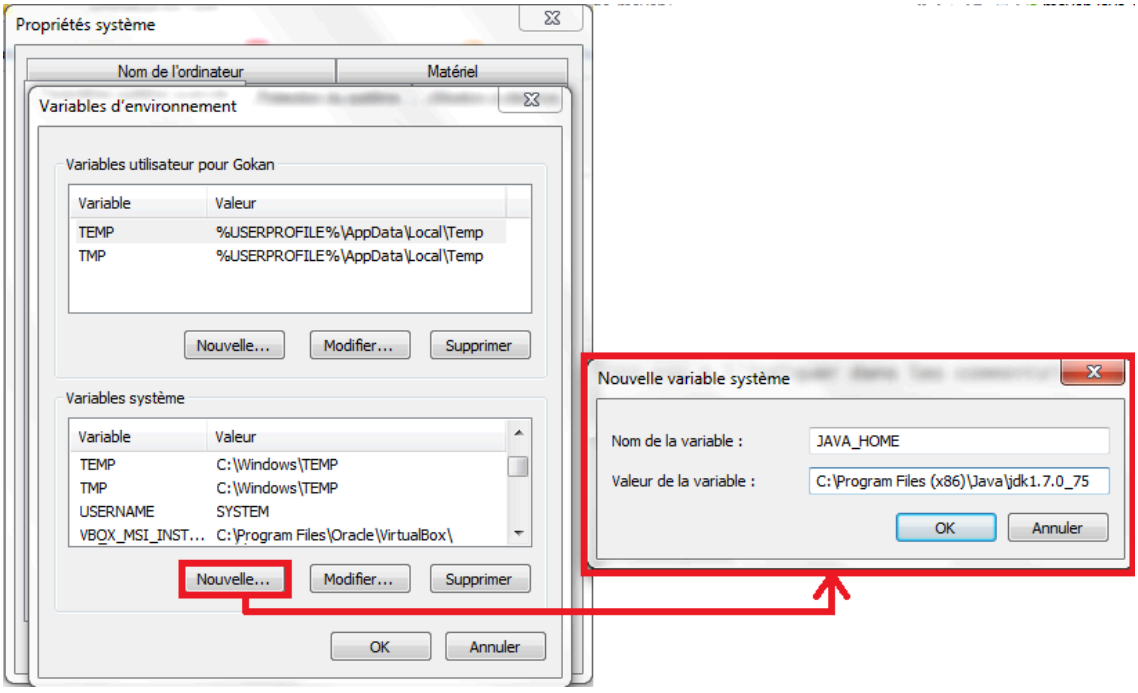
Pour les Windowsiens, allez dans le menu démarrer et saisissez « Modifier les variables d'environnement système » pour lancer le programme (en mode administrateur).



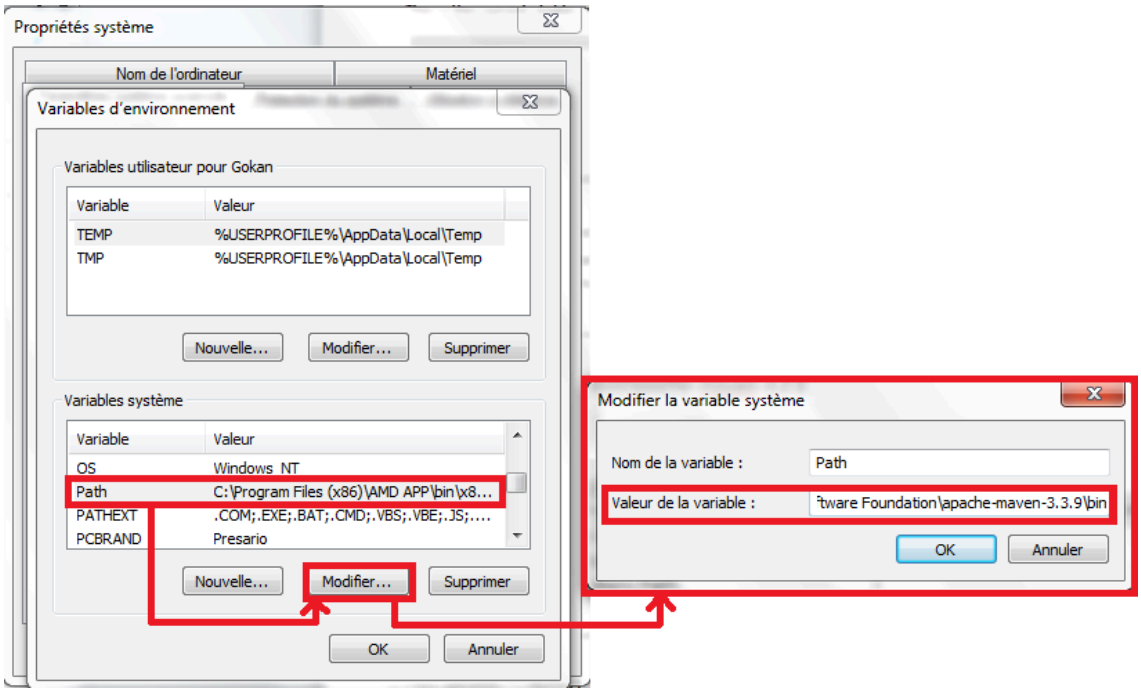
Puis dans l'onglet « Paramètres système avancés » cliquez sur le bouton « Variable d'environnement… » :



Un nouveau pop-up intitulé « Variables d'environnement » se lance. Nous allons d'abord ajouter la variable d'environnement JAVA_HOME sur votre machine, ceci permettra à Maven de connaître l'emplacement de votre JDK.

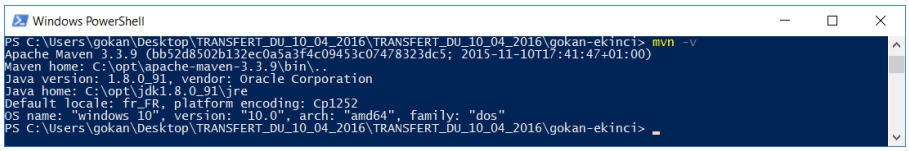


Toujours au niveau du pop-up intitulé « Propriété système », nous allons indiquer le chemin où se situe Maven pour pouvoir utiliser la commandemvn. Sélectionnez la variable d'environnement « Path » et appuyez sur le bouton « Modifier… ». Un pop-up intitulé « Modifier la variable système » se lance. Ajoutez le répertoire ***/bin*** de Maven comme ceci : « ;C:\Program Files\Apache Software Foundation\apache-maven-3.3.9\bin » (vous remarquerez que j'ai ajouté un point-virgule avant le **C:**, il s'agit d'un séparateur pour les variables d'environnement) :



Enfin, appuyez sur le bouton **OK** pour valider votre modification.

Nous allons très vite savoir si Maven est bien installé sur votre machine. Lancez votre console (**cmd.exe** pour ceux qui sont sous Windows) et entrez la commande suivante : `mvn -v`



Il s'agit d'une commande pour connaître la version de Maven. Si ce dernier indique votre version, c'est que Maven est bien installé. Si vous rencontrez un quelconque problème (ce qui ne devrait pas arriver en théorie ;-)), n'hésitez pas à l'indiquer dans les commentaires.

III - Création d'un premier projet Maven

Dans cette partie nous allons créer notre premier projet Maven et étudier « son cœur », le fichier **pom.xml**.

Lancez la console, placez-vous dans le répertoire où vous souhaitez créer le projet Java, et saisissez la commande suivante :

```
mvn archetype:generate -DgroupId=gokan.ekinci -DartifactId=identifiant-de-mon-projet
```

Le paramètre `archetype:generate` permet de créer un nouveau projet Maven, `-DgroupId=gokan.ekinci` d'indiquer votre compagnie ou nom ou pseudo, et `-DartifactId=identifiant-de-mon-projet` l'identifiant de votre application. À vrai dire vous pouvez juste utiliser la commande `mvn archetype:generate` pour créer votre projet mais utiliser ces paramètres accélère le processus de création de votre projet quand vous n'êtes plus à votre première création de projet. Notez que

lors de la création du projet la console va vous poser des questions, comme vous ne connaissez pas suffisamment Maven appuyez juste sur la touche **Entrée** de votre clavier pour accepter les valeurs par défaut.

La première fois que vous créez votre projet Maven, ce dernier va télécharger plusieurs JAR utilitaires (d'où les 500 Mo indiqués dans les prérequis).

Si vous avez bien suivi mes instructions, vous devriez vous retrouver avec un dossier **src** et un fichier **pom.xml** dans le répertoire où vous avez choisit de créer votre projet. Étudions le contenu du **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>gokan.ekinci</groupId>
    <artifactId>identifiant-de-mon-projet</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>identifiant-de-mon-projet</name>
    <url>http://maven.apache.org</url>

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

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

Analysons les balises :

Balise	Signification
<project>	La balise racine du fichier pom.xml, elle contient des paramètres pour valider le contenu XML
<modelVersion>	La version du POM (Project Object Model)
<groupId>	L'identifiant de la personne, équipe, ou entreprise qui développe le projet.
<artifactId>	Identifiant du projet (c'est aussi nom du projet au niveau du dossier racine).
<version>	La version du projet.
<packaging>	Représente le format de sortie (généralement : « jar » pour les applications JSE, « war » pour les applications JEE, « ear » pour les projets

	plus complexes). Si la balise n'est pas indiquée la valeur par défaut est « jar »
<name>	Nom du projet
<url>	Site web du projet
<description>	Description du projet
<dependencies>	Contient toutes les dépendances (les JAR) utilisées dans le projet

IV - Analyse d'une architecture Maven

Maintenant que nous avons créé notre projet, étudions son architecture. Le répertoire de notre projet est accompagné d'un fichier **pom.xml** et d'un dossier **/src** qui va contenir les dossiers **/main/java** pour le code source du projet et **/test/java** pour les tests unitaires. Ici gokan.ekinci représente le package des classes App et AppTest qui ont été créées par défaut :

```
identifiant-de-mon-projet
|-- pom.xml
`-- /src
    |-- /main/java/gokan/ekinci/App.java
    `-- /test/java/gokan/ekinci/AppTest.java
```

Le contenu ci-dessous présente des chemins pour que vous puissiez ajouter des fichiers statiques (.png, .pdf) dans un dossier **resources**, ou bien des fichiers web (.jsp, .html) dans un dossier **webapp** si le projet est de type Java EE (cf: packaging **war**) :

```
    identifiant-de-mon-projet
    |-- pom.xml
    `-- /src
        |-- /main
        |   |-- /java/gokan/ekinci/App.java
        |   |
        |   |-- /resources
        |   |   |-- /images
        |   |   |   `-- mon_image.jpg
        |   |   |
        |   |   `-- /pdf
        |   |       `-- mon_pdf.pdf
        |   |
        |   `-- /webapp
        |       |-- /WEB-INF
        |       |   `-- web.xml
        |       |-- index.jsp
        |       `-- /jsp
        |           `-- fichier.jsp
        |
        `-- /test/java/gokan/ekinci/AppTest.java
```

Avant de passer à la partie suivante, modifions légèrement le code du fichier **App.java** (je vais juste ajouter une balise <h1>) puis transformons le projet en JAR exécutable :

```
package gokan.ekinci;

public class App {
    public static void main( String[] args ) {
        String htmlText = "<h1>Hello World!</h1>";
        System.out.println( htmlText );
    }
}
```

Pour créer des JAR exécutable avec Maven, il faut ajouter le bloc XML suivant au **pom.xml** en précisant le nom de la classe contenant la méthode main() :

```
<build>
    <!-- Ajout du plugin pour créer un JAR exécutable -->
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jar-plugin</artifactId>
            <version>2.6</version>
            <configuration>
                <archive>
                    <manifest>
                        <mainClass>gokan.ekinci.App</mainClass>
                    </manifest>
                </archive>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Aperçu du **pom.xml** complet :

```
<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>gokan.ekinci</groupId>
    <artifactId>identifiant-de-mon-projet</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>identifiant-de-mon-projet</name>
    <url>http://maven.apache.org</url>

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

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>


    <build>
        <!-- Ajout du plugin pour créer un JAR exécutable -->
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <version>2.6</version>
                <configuration>
                    <archive>
                        <manifest>
                            <mainClass>gokan.ekinci.App</mainClass>
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Lancez la console, placez-vous dans le répertoire **/identifiant-de-mon-projet**, saisissez la commande suivante : mvn package.

Un nouveau dossier **/target** vient d'être créé à côté de **/src**. Ce dossier **/target** contient les classes compilées, vous y trouverez également notre JAR exécutable avec un nom sous la forme <artifactId>-<version>.<extension>, en ce qui nous concerne le fichier se nomme **identifiant-de-mon-projet-1.0-SNAPSHOT.jar**. Comme ce fichier se trouve dans le dossier **/target**, placez-vous sous ce dernier et lancez le programme :

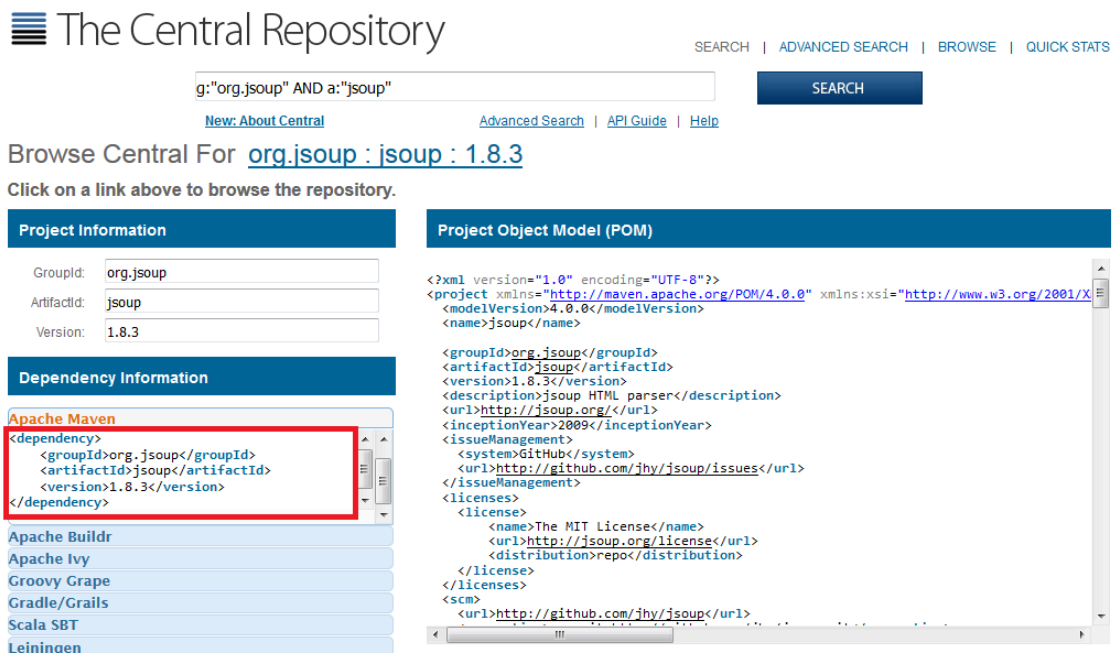
```
cd target
java -jar identifiant-de-mon-projet-1.0-SNAPSHOT.jar
```

Le programme affiche : **<h1>Hello World!</h1>**

V - Le repository Maven

Nous avons créé notre premier projet Maven, qu'attendons-nous pour inclure une dépendance dans notre projet ? Nous allons donc inclure dans notre projet une petite dépendance intitulée **jsoup** pour supprimer les balises HTML d'un texte.

Vous pouvez jeter un coup d'œil au Repository central de Maven sur http://search.maven.org/ afin de récupérer le bloc XML de la dépendance que vous recherchez :

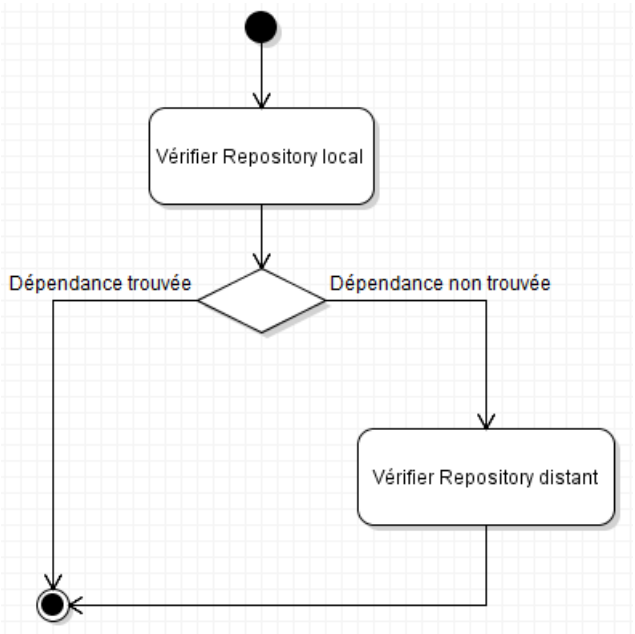


Il existe d'autres moteurs de recherche, parfois plus à jour comme **MVN Repository** (http://mvnrepository.com) :

Voici le bloc de code XML que nous allons inclure dans la balise <dependencies> de notre **pom.xml :**

```
<dependency>
    <groupId>org.jsoup</groupId>
    <artifactId>jsoup</artifactId>
    <version>1.8.3</version>
</dependency>
```

Lorsque vous allez lancer Maven, ce dernier va lire votre fichier **pom.xml** et chercher les dépendances dans votre repository local (votre ordinateur). Si une dépendance n'est pas présente dans le repository local, Maven va chercher dans un repository distant (dans notre cas le repository central de Maven) puis placer la dépendance récupérée dans le repository local :



Au fait, savez-vous où se trouve votre repository local par défaut ? :)

- Sous Windows : **C:\Users\{compte-windows}\.m2**
- Sous Linux : **~/.m2**

Si vous souhaitez modifier l'emplacement par défaut de votre repository local, il vous suffit de décommenter la balise <localRepository> du fichier */conf/settings.xml* présent dans votre répertoire d'installation Apache Maven et d'indiquer le nouveau chemin.

Par défaut un JAR ne contient pas les autres JAR indiqués dans le **pom.xml** (cf : les dépendances), vous imaginez pourquoi, nous pourrions nous retrouver avec des dépendances de plusieurs méga-octets et des exécutables dans l'ordre du giga-octet. Le souci c'est que dans ce cas précis notre JAR est un exécutable et doit inclure toutes ses dépendances pour être opérationnel. Nous allons donc mettre à jour le <build> de notre **pom.xml** pour réaliser cette opération, ne vous en faites pas, voici le **pom.xml** complet :

```
<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>gokan.ekinci</groupId>
    <artifactId>identifiant-de-mon-projet</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>identifiant-de-mon-projet</name>
    <url>http://maven.apache.org</url>

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

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>

        <!-- Ajout de la dépendance jsoup -->
        <dependency>
            <groupId>org.jsoup</groupId>
            <artifactId>jsoup</artifactId>
            <version>1.8.3</version>
        </dependency>
    </dependencies>


    <build>
        <!-- Ajout du plugin pour créer un JAR exécutable -->
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-assembly-plugin</artifactId>
                <version>2.6</version>
                <configuration>
                    <archive>
                        <manifest>
                            <mainClass>gokan.ekinci.App</mainClass>
                        </manifest>
                    </archive>
                    <descriptorRefs>
                        <descriptorRef>jar-with-dependencies</descriptorRef>
                    </descriptorRefs>
                </configuration>
```

```
                <executions>
                    <execution>
                        <id>make-assembly</id>
                        <phase>package</phase>
                        <goals>
                            <goal>single</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

Ajoutez Jsoup au code :

```
package gokan.ekinci;
import org.jsoup.Jsoup;

public class App {
    public static void main( String[] args ) {
        String htmlText = "<h1>Hello World!</h1>";
        System.out.println( Jsoup.parse(html).text() );
    }
}
```

Allez dans le dossier où se situe votre **pom.xml** et exécutez la commande suivante : mvn clean package

Cette commande aura pour effet de nettoyer votre projet (en supprimant le dossier **/target**) puis de recréer votre JAR exécutable. Retournez dans le dossier **/target** et vous remarquerez qu'il existe deux JAR, nous choisirons d'exécuter le nouveau qui possède un nom plus long et qui pèse 310Ko de plus (votre dépendance est incluse dans le jar) :

```
java -jar identifiant-de-mon-projet-1.0-SNAPSHOT-jar-with-dependencies.jar
```

Le programme affiche : **Hello World!**, les balises <h1> ont été nettoyées par **jsoup**.

VI - Build Lifecycle

Vous l'avez déjà vu dans les parties précédentes avec compile et package, Maven propose des commandes pour nous permettre de gérer le **cycle de vie** de notre projet Maven, ces commandes sont ce qu'on appelle les « phases » du **Build Lifecycle**.

Intéressons-nous aux autres phases que nous propose Maven.

La commande mvn validate permet de valider votre **pom.xml**. Par exemple si j'oublie de fermer une balise XML du **pom.xml**, Maven m'indiquera qu'il y a une erreur en donnant le numéro de la ligne concernée. Cependant cette commande n'ira pas vérifier si votre code Java est correct.

La commande mvn test va lancer le code des tests unitaires présent dans le répertoire **/src/test/java**. Il n'est pas nécessaire de compiler le projet avant de lancer cette commande, Maven s'en chargera lui-même.

La commande mvn verify est différente de validate, elle permet de valider le package. Cette commande encapsule plusieurs phases, si nécessaire, verify va non seulement chercher à valider le contenu du **pom.xml**, mais aussi compiler le code source, lancer les tests unitaires, créer le package.

La commande mvn install est similaire à package, mais présente une légère différence, votre package sera déployé sur votre repository local pour que vous puissiez l'utiliser à partir d'un autre projet. Vous pourrez par exemple créer un nouveau projet Maven en incluant ceci dans le **pom.xml :**

```
<dependency>
    <groupId>gokan.ekinci</groupId>
    <artifactId>identifiant-de-mon-projet</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

La commande mvn deploy va déployer votre package vers un autre repository. Hé oui, Maven ne se résume pas seulement à votre repository local ou le repository central. En entreprise vous n'aurez parfois pas accès au repository central, vous aurez peut-être à utiliser le repository de votre entreprise. Voici un exemple de configuration vous permettant d'indiquer un repository vers lequel vous pourrez déployer votre package :

```
<project ...>
    <distributionManagement>
        <!-- Pour la publication des releases -->
        <repository>
            <id>nexus-releases</id>
            <name>releases</name>
            <url>http://192.168.20.7:9500/nexus/content/repositories/releases/</url>
        </repository>

        <!-- Pour la publication des snapshots -->
        <snapshotRepository>
            <id>nexus-snapshots</id>
            <name>snapshots</name>
            <url>http://192.168.20.7:9500/nexus/content/repositories/snapshots/</url>
        </snapshotRepository>
    </distributionManagement>
</project>
```

Dans l'exemple précédent, nous indiquons deux chemins vers lesquels seront déployés nos projets. Si notre projet est en cours de développement (ex : <version>1.0-SNAPSHOT</version>) alors notre package sera déployé vers ce qui est indiqué dans <snapshotRepository>. Si notre projet est stable (ex : <version>1.0</version>, vous remarquerez qu'il suffit d'enlever SNAPSHOT pour être considéré comme une version release) alors notre package sera déployé vers ce qui est indiqué dans <repository>.

Deux personnes téléchargeant une même version SNAPSHOT à différentes périodes peuvent avoir un package différent, mais deux personnes téléchargeant une même version RELEASE ***doivent toujours avoir le même package****.*

Récapitulatif des phases :

Nom	Utilité
validate	Valider le pom.xml
compile	Compiler les fichiers .java en .class dans le dossier /target
test	Lancer les tests unitaires dans /src/test/java
package	Créer un package (un fichier « .jar », « .war », « .ear »)
verify	Lancer des tests pour vérifier la validité du package
install	Déployer le package vers le repository local afin qu'il puisse être accessible dans d'autres projets
deploy	Déployer le package vers un repository distant

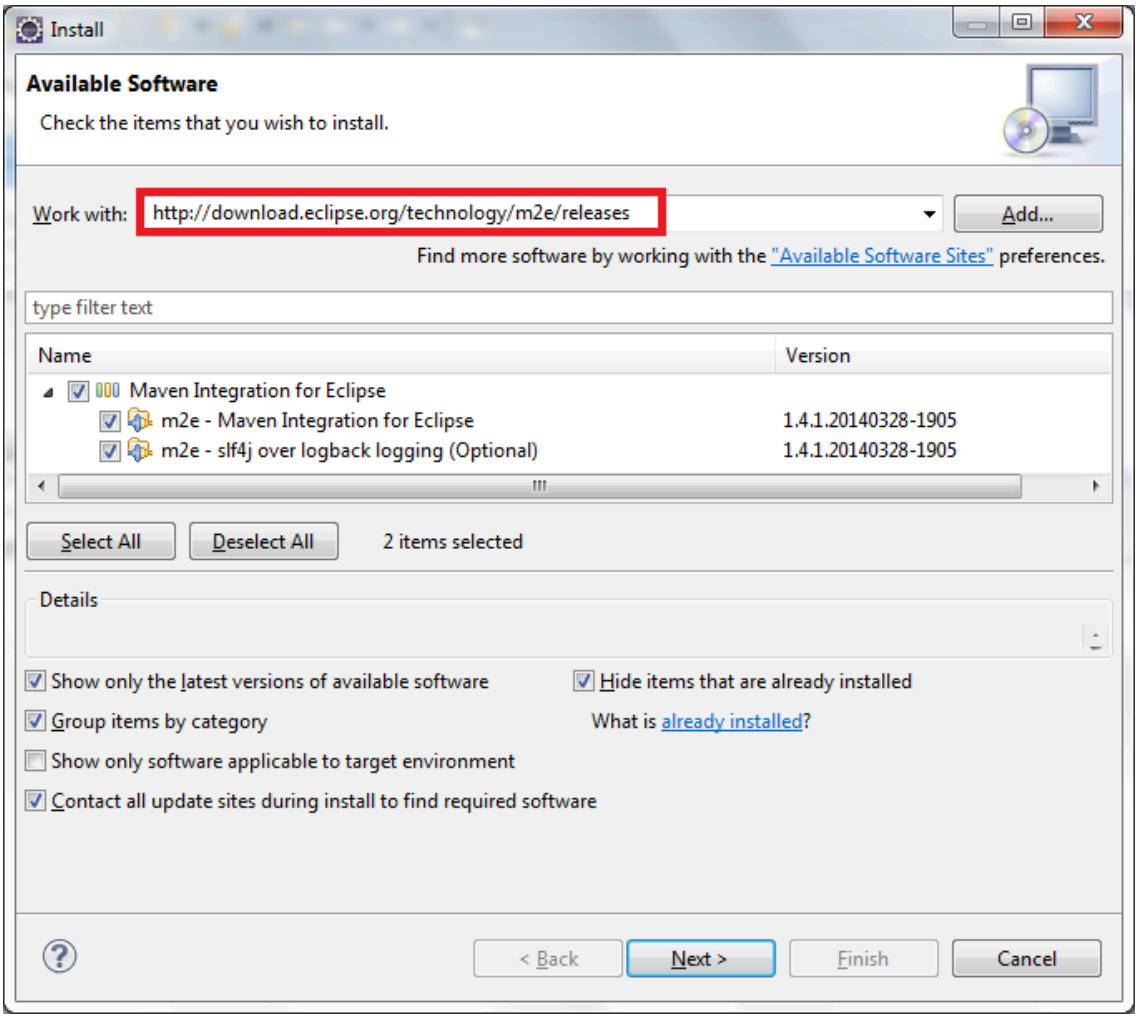
VII - Création d'un projet Maven avec un IDE

Développer ses applications à partir d'un simple éditeur de texte et savoir utiliser Maven en ligne de commande sont des tâches très instructives, mais savoir utiliser Maven à partir d'un IDE est plus pratique pour développer ses applications. Dans cette partie, je vais vous montrer comment on intègre un projet Maven dans un IDE Eclipse et IntelliJ.

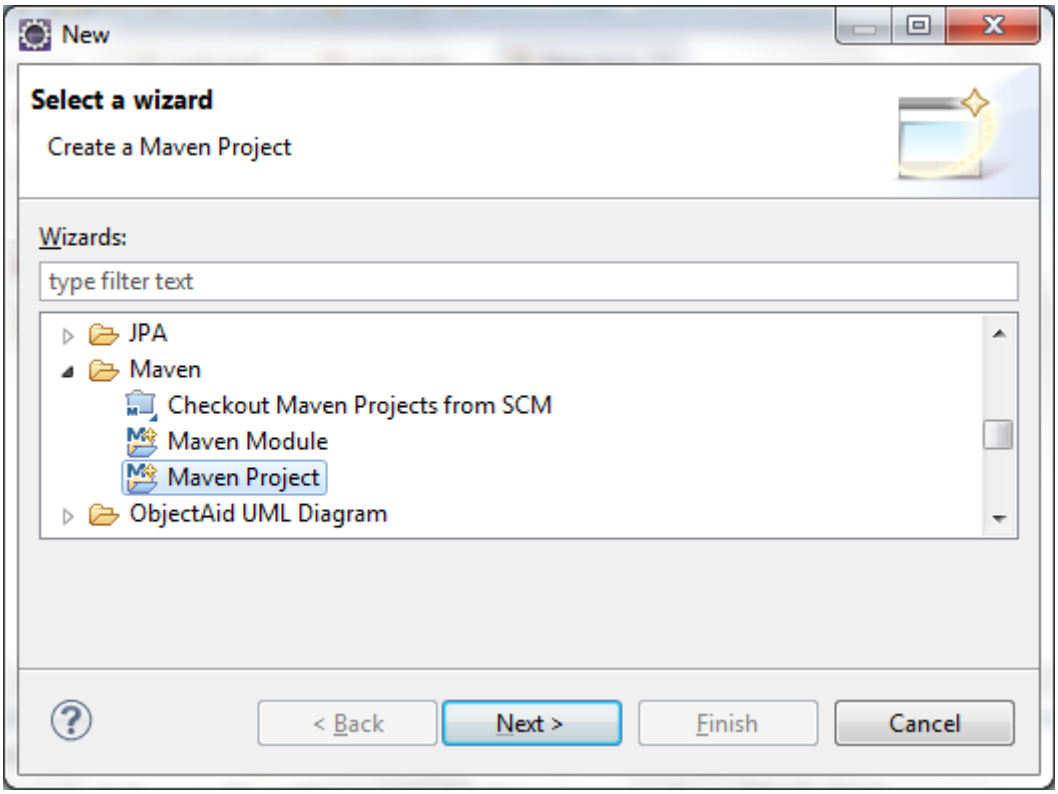
VII-A - Création d'un projet Maven avec Eclipse

Dans cette partie je vais vous montrer comment installer le plugin **m2eclipse** et utiliser Maven avec l'IDE Eclipse (la version utilisée est **Eclipse Luna SR2**).

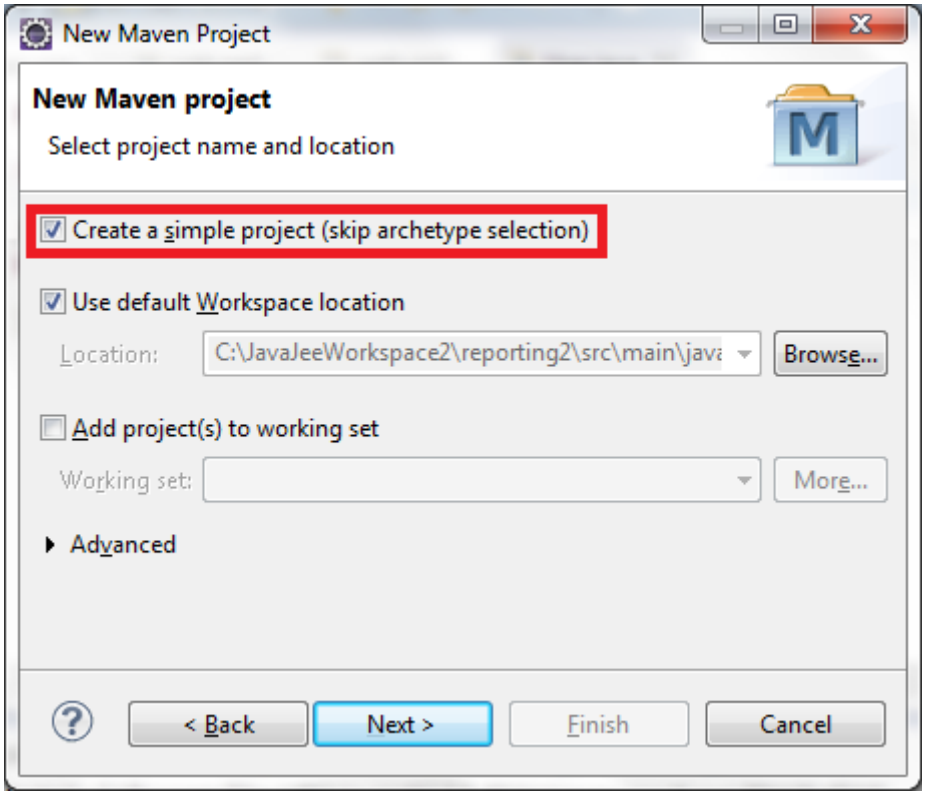
Dans la barre Eclipse, allez dans « Help > Install New Software… », saisissez « http://download.eclipse.org/technology/m2e/releases » dans le champ « Work with: ». Eclipse se chargera de trouver le plugin, il ne vous restera plus qu'à cocher les champs ci-dessous, accepter la licence, faire **Next** jusqu'au bout pour que l'installation puisse se faire :



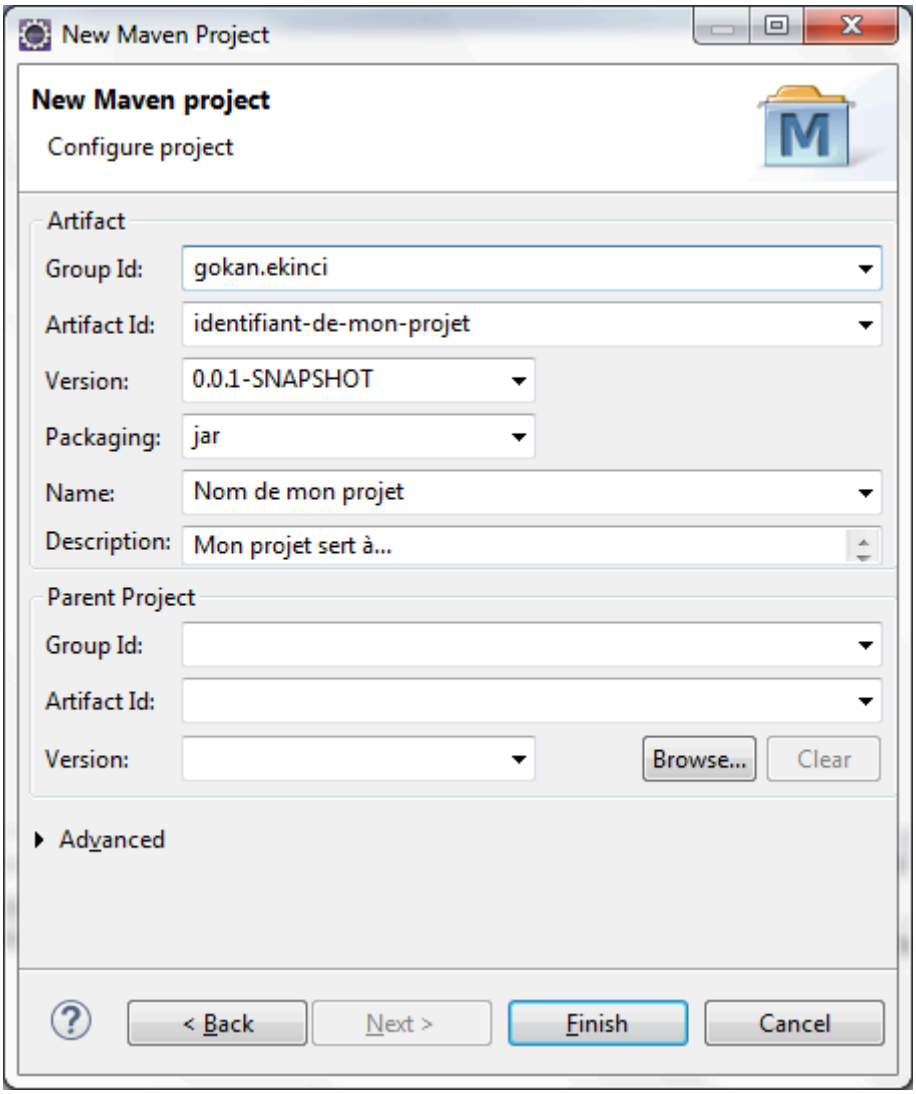
Pour créer un nouveau projet avec Maven, allez dans la barre Eclipse « File > New > Others… > Maven > Maven Project » :



À l'étape suivante, pensez à cocher la case « Create a simple project (skip archetype selection) » pour créer un simple projet Maven :



Il ne vous reste plus qu'à renseigner les informations de base requises pour le **pom.xml** et faire **Finish :**



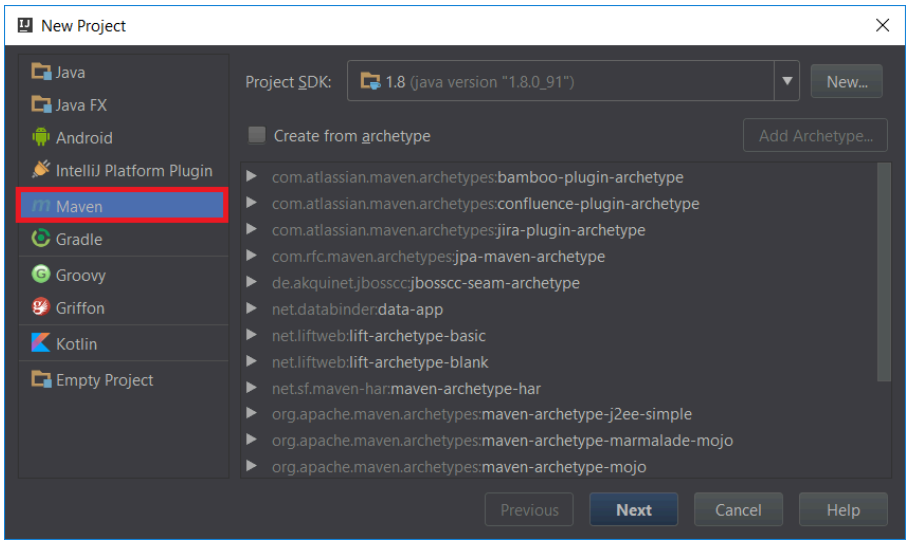
Quelques informations supplémentaires :

- Grâce à Maven, vous aurez beaucoup plus de facilité à intégrer un projet Maven existant dans Eclipse qu'un simple projet Eclipse de base dans Eclipse (cela peut vous paraître étrange mais importer le projet Eclipse d'une autre personne transformera très souvent votre IDE en sapin de Noël). Pour importer un projet Maven existant dans votre IDE rien de plus simple, allez dans la barre Eclipse : « **File > Import… > Maven > Existing Maven Projects** »
- Les IDE vont généralement générer des fichiers internes au fonctionnement du projet, tels qu'un dossier *.settings* ou les fichiers *.classpath*, *.project* ou *.idea* à côté du **pom.xml**. Lorsqu'on utilise un outil de versionning tel que Git ou SVN, il est strictement inutile d'uploader ces fichiers, les coéquipiers qui vont récupérer le projet n'en ont pas besoin, ces fichiers ne feront que polluer votre repository Git ou SVN. N'hésitez pas à indiquer ces fichiers locaux dans votre fichier *.gitignore* (pour Git) ou utiliser le paramètre `svn:ignore` (pour SVN).

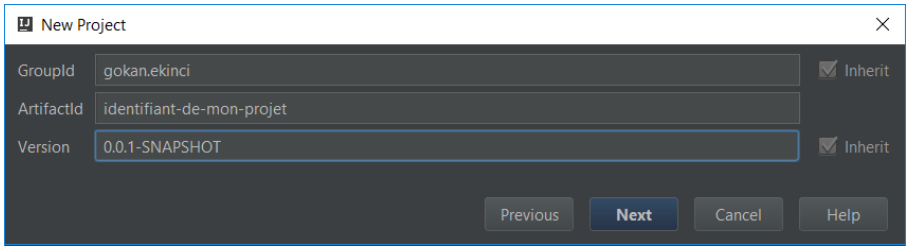
VII-B - Création d'un projet Maven avec IntelliJ

Dans cette partie je vais vous montrer comment utiliser Maven avec l'IDE IntelliJ (la version utilisée est **IntelliJ Community 2016.1.2**).

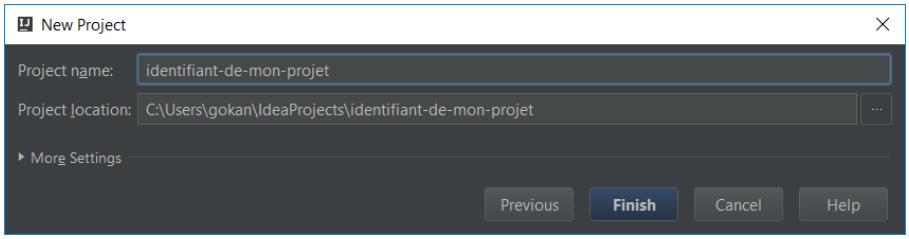
Pour créer un projet Maven avec IntelliJ il suffit de faire « File > New > Project… ». Une pop-up se lance, choisissez « Maven » comme indiqué ci-dessous :



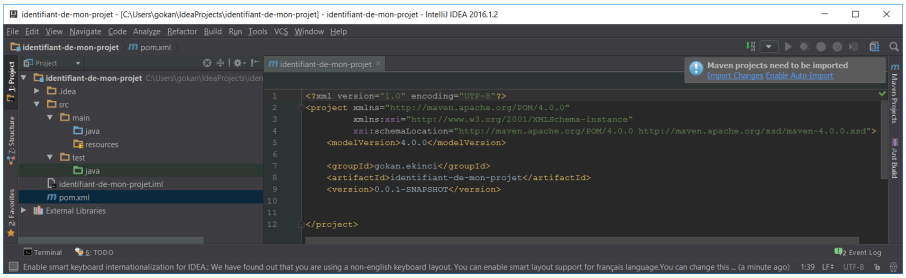
Il ne vous reste plus qu'à renseigner les informations de base requises pour le **pom.xml :**



Et enfin d'indiquer le nom de votre projet IntelliJ (vous pouvez reprendre l'identifiant de votre projet, soit l'**artifactId**) ainsi que l'emplacement sur votre disque :



IntelliJ vous a généré un **pom.xml**, votre projet est prêt :



Information supplémentaire:

- Les IDE vont généralement générer des fichiers internes au fonctionnement du projet, tels qu'un dossier *.settings* ou les fichiers *.classpath*, *.project* ou *.idea* à côté du **pom.xml**. Lorsqu'on utilise un outil de versionning tel que Git ou SVN, il est strictement inutile d'uploader ces fichiers, les coéquipiers qui vont récupérer le projet n'en ont pas besoin, ces fichiers ne feront que polluer votre repository Git ou SVN.

N'hésitez pas à indiquer ces fichiers locaux dans votre fichier *.gitignore* (pour Git) ou utiliser le paramètre `svn:ignore` (pour SVN).

VIII - Les projets multi-modules Maven

Maven permet de **mieux organiser son projet Java**. En effet, un gros projet implique parfois un découpage en plusieurs parties. Maven permet de hiérarchiser vos projets grâce à son concept de multi-modules, vous pourrez ainsi créer des projets Maven (cf: des modules) dans vos projets Maven.

Dans l'exemple qui va suivre, nous avons un projet « trains » avec une architecture orientée service RESTful, l'objectif sera de nous familiariser avec le concept de multi-modules Maven. Vous n'aurez rien à développer, le code source du projet est disponible dans **ce repository Github** :

1 Clonez le projet sur votre machine ;
2 Renommez le dossier **maven-tutorial-trains** en **trains** (cette étape est facultative) ;
3 Exécutez la commande `mvn clean install` dans le projet **trains**.

VIII-A - L'architecture du projet

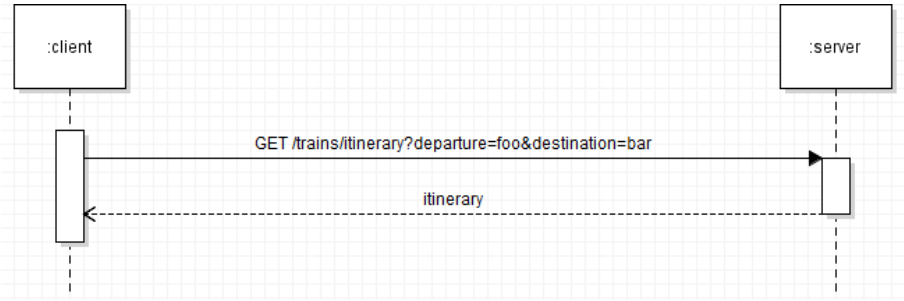
Le projet parent **trains** contient juste un **pom.xml** et regroupe tous nos modules, ce projet nous sert à **centraliser la version de toutes nos dépendances** pour nos modules :

```
/trains
|--- pom.xml
|
|--- /trains-client
|    |--- /src/main/java/fr/ekinci/trains/client
|    |                                  |--- MainClient.java
|    |                                  `--- TrainsRestClient.java
|    `--- pom.xml
|
|--- /trains-server
|    |--- /src/main/java/fr/ekinci/trains
|    |                            |--- /server/TrainsController.java
|    |                            `--- TrainsApplication.java
|    `--- pom.xml
|
`--- /trains-common
     |--- /src/main/java/fr/ekinci/trains/common/ItineraryItem.java
     `--- pom.xml
```

Voici les trois sous-modules **trains :**

- **trains-client :** ce projet contient notre client REST pour interroger le serveur sur l'itinéraire des trains. Nous utiliserons la bibliothèque REST **Retrofit** ;
- **trains-server :** ce projet expose un service REST pour informer sur l'itinéraire des trains. Nous utiliserons le **micro-framework Spring Boot** ;
- **trains-common :** ce projet contient des classes communes entre nos modules **trains-client** et **trains-server**, cela nous évitera de dupliquer du code. Nous utiliserons la bibliothèque **Lombok** pour générer nos getters/setters et builder (**Si** vous utilisez un IDE, pensez à installer le plugin Lombok afin que les annotations `@Data` et `@Builder` soient reconnues). N'hésitez pas à lire **le cours de thierryler sur les annotations Lombok**.

VIII-B - Diagramme de séquence



Pour résumer ce diagramme de séquence, notre client envoie une requête de type GET et le serveur retourne un itinéraire qui sera **mocké**.

VIII-C - Le projet parent trains

Le projet **trains** va piloter l'ensemble des modules, pour cela le **pom.xml** doit faire quatre choses :

1 Déclarer tous les modules ;
2 Déclarer toutes les versions des dépendances grâce à la balise <properties>. Les versions pourront être récupérées grâce au symbole ${} ;
3 Déclarer toutes les dépendances grâce à la balise <dependencyManagement> afin de déléguer la gestion des versions au projet parent ;
4 Déclarer tous les plugins grâce à la balise <pluginManagement> afin de déléguer la gestion des versions au projet parent

*Les modules hériteront de la configuration par défaut que vous aurez définie dans les balises **dependencyManagement** et **pluginManagement** du projet parent. Vous n'aurez par exemple plus à préciser la **version** de votre dépendance, mais rien ne vous empêchera d'overrider (redéfinir) la configuration de votre module si vous le désirez.*

```
<project ...>
    <!-- 1. Les modules du projet trains -->
    <modules>
        <module>trains-client</module>
        <module>trains-server</module>
        <module>trains-common</module>
    </modules>

    <!-- 2. Les versions -->
    <properties>
        <lib.version>1.0.1</lib.version>
        <plugin.version>3.1.8</plugin.version>
        <!-- et cetera -->
    </properties>

    <!-- 3. Les dépendances -->
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>dependency-groupdId</groupId>
                <artifactId>dependency-artifactId</artifactId>
                <version>${lib.version}</version>
            </dependency>
            <dependency>...</dependency>
            <dependency>...</dependency>
        </dependencies>
    </dependencyManagement>
```

```
    <!-- 4. Les plugins -->
    <build>
        <pluginManagement>
            <plugins>
                <plugin>
                    <groupId>plugin-groupdId</groupId>
                    <artifactId>plugin-artifactId</artifactId>
                    <version>${plugin.version}</version>
                </plugin>
                <plugin>...</plugin>
                <plugin>...</plugin>
            </plugins>
        </pluginManagement>
    </build>
</project>
```

VIII-D - Les modules du projet trains

Les trois modules du projet **trains** doivent indiquer dans leur **pom.xml :**

1 Leur parent ;
2 Les dépendances et plugins (sans avoir à préciser les versions, c'est le projet parent qui s'en charge).

```
<project ...>
    <!-- 1. Le parent -->
    <parent>
        <groupId>gokan.ekinci</groupId>
        <artifactId>trains</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <relativePath>../pom.xml</relativePath>
    </parent>

    <!-- 2. Les dépendances et plugins-->
    <dependencies>
        <dependency>
            <groupId>dependency-groupdId</groupId>
            <artifactId>dependency-artifactId</artifactId>
            <!-- Inutile de préciser la version, sauf si vous souhaitez apporter une valeur
 différente de celle définit dans le <dependencyManagement> du projet parent -->
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>plugin-groupdId</groupId>
                <artifactId>plugin-artifactId</artifactId>
                <!-- Inutile de préciser la version, sauf si vous souhaitez apporter une valeur
 différente de celle définit dans le <pluginManagement> du projet parent -->
            </plugin>
        </plugins>
    </build>
</project>
```

VIII-D-1 - Le module trains-common

Ce module va regrouper toutes nos classes modèles qui seront communes entre **trains-client** et **trains-server**. Nous avons une classe `ItineraryItem` :

```
@Data
@Builder
public class ItineraryItem {
    // Mode de transport (métro, RER etc)
    private String mode;
    // Station de départ
```

```
    private String departureStation;
    // Direction du train
    private String direction;
    // Nombre de station total
    private int nbStations;
    // Durée du train
    private int duration;
    // Station d'arrivée
    private String arrivalStation;
}
```

Afin que cette classe soit disponible dans **trains-client** et **trains-server**, ces deux projets devront déclarer une dépendance vers **trains-common :**

```
<dependency>
    <groupId>gokan.ekinci</groupId>
    <artifactId>trains-common</artifactId>
    <version>${project.version}</version>
</dependency>
```

VIII-D-2 - Le module trains-client

Ce module contient notre client REST TrainsRestClient ainsi qu'une classe MainClient dédiée à l'exécution du code pour lister les itinéraires :

```
public interface TrainsRestClient {
    @GET("/trains/itinerary")
    Call<List<ItineraryItem>> getItinerary(@Query("departure") String
 departure, @Query("destination") String destination);
}
```

Le main() de MainClient :

```
public static void main(String[] args) throws IOException {
    String departure = "87, bd Jean Jaurès, 92110, Clichy";
    String destination = "1, Rue du Chat qui Pêche, 75005, Paris";

    TrainsRestClient restClient = getRetrofit("http://
localhost:8080").create(TrainsRestClient.class);
    List<ItineraryItem> itineraryItems = restClient.getItinerary(departure,
 destination).execute().body();

    for (ItineraryItem item : itineraryItems) {
        System.out.println("Mode de transport    : " + item.getMode());
        System.out.println("Départ              : " + item.getDepartureStation());
        System.out.println("  -> en direction de : " + item.getDirection());
        System.out.println("Nom de stations     : " + item.getNbStations() + " | Durée : " +
 item.getDuration() + " min");
        System.out.println("Arrivée             : " + item.getArrivalStation());
        System.out.println("-+-+-+-+-+-+-+-+-+-+-+-+-");
    }
}
```

VIII-D-3 - Le module trains-server

Ce module contient une classe TrainsApplication pour initialiser Spring Boot et un contrôleur. Voici le code du contrôleur :

```
@RestController
@RequestMapping("/trains")
public class TrainsController {

    @RequestMapping(value = "/itinerary", method = RequestMethod.GET)
```

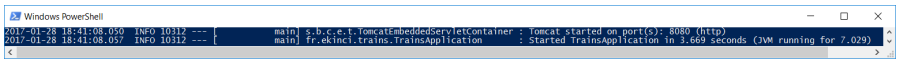
```
    public List<ItineraryItem> getItinerary(
        @RequestParam("departure") String departure,
        @RequestParam("destination") String destination
    ) {
        return getMock();
    }

    public List<ItineraryItem> getMock() {
        List<ItineraryItem> itineraryItems = new ArrayList<>();
        itineraryItems.add(
            ItineraryItem.builder()
                .mode("Métro Ligne 13")
                .departureStation("Mairie de Clichy")
                .direction("Châtillon Montrouge")
                .nbStations(9)
                .duration(14)
                .arrivalStation("Invalides")
                .build()
        );
        itineraryItems.add(
            ItineraryItem.builder()
                .mode("RER C")
                .departureStation("Invalides")
                .direction("Gare d'Austerlitz")
                .nbStations(2)
                .duration(6)
                .arrivalStation("Saint-Michel Notre-Dame")
                .build()
        );
        return itineraryItems;
    }
}
```

VIII-E - Démonstration

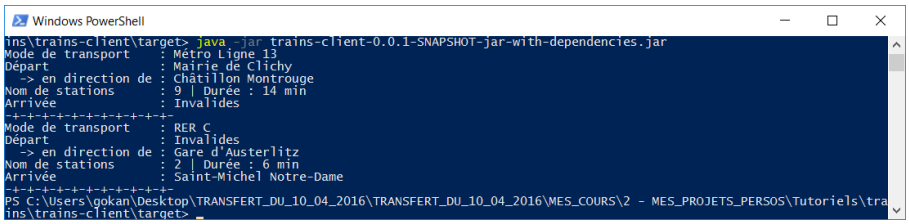
Lorsque vous avez cloné le projet **trains**, vous lancez la commande mvn clean install, cela a téléchargé toutes les dépendances et généré le dossier target des projets.

Serveur : ouvrez une nouvelle console, allez dans le projet **trains-server**, et exécutez la commande mvn spring-boot:run pour lancer le serveur. Votre serveur va par défaut se lancer sur le port 8080 (vous pouvez le vérifier en lisant les messages affichés sur la console, ou même modifier le port via la propriété server.port si 8080 est réservé par un autre service sur votre machine) :



Vous pouvez vérifier que votre service est opérationnel en ouvrant votre navigateur et en allant à l'adresse suivante : **http://localhost:8080/trains/itinerary?departure=foo&destination=bar**

Client : ouvrez une nouvelle console, allez dans le projet **trains-client**, allez dans le dossier **/target** et exécutez la commande java -jar trains-client-0.0.1-SNAPSHOT-jar-with-dependencies.jar pour lancer votre client :



IX - Conclusion

Maven simplifie grandement le développement des applications Java, il serait dommage de s'en priver, et d'ailleurs pourquoi ne pas pousser vos connaissances ?

- **Sonatype Nexus** : un outil pour mettre en place votre propre repository distant.
- Maven permet de lancer vos tests :
    - **JUnit** : le framework par excellence pour créer vos **tests unitaires** ;
    - **Mockito** : un framework très utilisé pour **mocker** des objets dans vos tests ;
    - **spring-test** : un module Spring pour vos **tests d'intégration**.
- Maven a lui aussi des concurrents plus ou moins équivalents dans l'univers Java :
    - **Ant** : un outil plus léger que Maven, il possède moins de fonctionnalités que ce dernier (ex.: pas de cycle de vie) ;
    - **Gradle** : un outil plus jeune que Maven, il possède les mêmes fonctionnalités que ce dernier, mais utilise le langage Groovy au lieu du XML pour son équivalent du **pom.xml** (cf: **build.gradle**).
- Maven est un outil initialement créé pour le langage Java, mais il existe des alternatives pour les autres technologies : **Composer** pour PHP, **pip** pour Python, **npm** pour JavaScript, **NuGet** pour C#, etc.

X - Remerciements

Je tiens à remercier **Mickael Baron** pour ses conseils, **Robin56** pour la relecture technique et **ClaudeLELOUP** pour la relecture orthographique.