# Maven & Git(Lab) : pour bien commencer

## Maven

Le compilateur Java peut s'avérer assez fastidieux à manipuler quand il s'agit de compiler des projets complexes qui intègrent des tests. Pour faciliter cette tâche, l'outil Maven permet d'automatiser les différentes étapes de compilation d'un projet de développement.

### Étape n°1 : Créer la structure du projet

La première tâche que Maven permet de réaliser est la création de la structure arborescente d'un projet de développement. Pour créer cette structure, il vous faut utiliser la commande `mvn archetype:generate`. Nous cherchons à créer un projet de développement Java (`maven-archetype-quickstart`) dont l'identifiant est ici `831` (choix par défaut). La plupart des paramètres qui suivent peuvent prendre la valeur par défaut, hormis `groupId` que nous fixons à `fil.coo` (et qui fera office de nom de *package*) et `artifactId` que nous fixons à `COO-TP1` (et qui correspond au nom du projet). Vous obtenez alors alos la séquence suivante (les choix par défaut sont validés en appuyant simplement sur la touche `[ENTRÉE]` du clavier) :

```
$~$ cd COO/
$COO$ mvn archetype:generate
[INFO] Scanning for projects...
Downloading: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-deploy-plugin/2.7/...
Downloaded: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-deploy-plugin/2.7/...
... [Nombreux chargements] ...
1665: remote -> us.fatehi:schemacrawler-archetype-plugin-lint (-)
Choose a number or apply filter (format: [groupId:]artifactId, case sensitive contains): 831: [ENTRÉE]
Choose org.apache.maven.archetypes:maven-archetype-quickstart version:
1: 1.0-alpha-1
2: 1.0-alpha-2
3: 1.0-alpha-3
4: 1.0-alpha-4
5: 1.0
6: 1.1
Choose a number: 6: [ENTRÉE]
Define value for property 'groupId': : fil.coo
Define value for property 'artifactId': : COO-TP1
Define value for property 'version':  1.0-SNAPSHOT: : [ENTRÉE]
Define value for property 'package':  fil.coo: : [ENTRÉE]
Confirm properties configuration:
groupId: fil.coo
artifactId: COO-TP1
version: 1.0-SNAPSHOT
package: fil.coo
 Y: : [ENTRÉE]
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Old (1.x) Archetype: maven-archetype-quick...
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: basedir, Value: /Users/[MON_LOGIN]/COO
[INFO] Parameter: package, Value: fil.coo
[INFO] Parameter: groupId, Value: fil.coo
[INFO] Parameter: artifactId, Value: COO-TP1
[INFO] Parameter: packageName, Value: fil.coo
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir: /Users/[MON_LOGIN]/COO/COO-TP1
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 21.386 s
[INFO] Finished at: 2016-09-08T13:07:05+02:00
[INFO] Final Memory: 14M/209M
[INFO] ------------------------------------------------------------------------
```

Une fois la configuration terminée, votre projet est désormais prêt pour la phase de développement.

**Dans le cadre de COO cette action devra impérativement être la première étape de vos projets (avant tout passage par Eclipse ou autre). Cela vous garantira de partir d'une structure « propre ».**

Par défaut, tout nouveau projet est configuré avec l'arborescence suivante :

```
$COO$ cd COO-TP1/
$COO-TP1$ tree
.
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── fil
    │           └── coo
    │               └── App.java
    └── test
        └── java
            └── fil
                └── coo
                    └── AppTest.java

9 directories, 3 files
```

Il faut retenir 3 informations importantes :

1. le fichier `pom.xml` contient les paramètres du projet et la configuration du processus de compilation,
2. le répertoire `src/main/java` contient tout le code applicatif du projet,
3. le répertoire `src/test/java` contient tous les tests unitaires du projet.

Evidemment les fichiers `App.java` et `AppTest.java` doivent à terme disparaitre de vos projets même si nous les utiliserons ici pour la suite de cet exemple.

### Étape n°2 : Compiler le projet

Dès à présent, votre projet peut être compilé et livré en utilisant la commande `mvn package` :

```
$COO-TP1$ mvn package
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building COO-TP1 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ COO-TP1 ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory /Users/[MON_LOGIN]/COO/test/COO-TP1/src/main/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ COO-TP1 ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to /Users/[MON_LOGIN]/COO/test/COO-TP1/target/classes
[INFO]
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ COO-TP1 ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory /Users/[MON_LOGIN]/COO/test/COO-TP1/src/test/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.1:testCompile (default-testCompile) @ COO-TP1 ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 1 source file to /Users/[MON_LOGIN]/COO/test/COO-TP1/target/test-classes
[INFO]
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ COO-TP1 ---
[INFO] Surefire report directory: /Users/[MON_LOGIN]/COO/test/COO-TP1/target/surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running fil.coo.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.006 sec
```

```
Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ COO-TP1 ---
[INFO] Building jar: /Users/[MON_LOGIN]/COO/test/COO-TP1/target/COO-TP1-1.0-SNAPSHOT.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 4.235 s
[INFO] Finished at: 2016-09-08T13:20:24+02:00
[INFO] Final Memory: 16M/170M
[INFO] ------------------------------------------------------------------------
```

Vous pouvez observer que Maven a d'abord compilé le code de l'application, puis les tests avant d'exécuter ces derniers en utilisant JUnit. Si aucun test n'échoue, alors Maven génère une archive `.jar` à partir du code compilé de votre projet (sans les tests) qu'il dépose dans le répertoire `target/`.

**Remarque** Vous pouvez spécifier la version de java à considérer pour le compilateur et les sources en complétant l'élément `properties` avec les informations suivantes (ici pour java 1.8)

```
<properties>
  ... other already defined properties
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
</properties>
```

Vous pouvez désormais modifier le code de l'application en ajoutant des fichiers dans les différents répertoires et recompiler le projet en utilisant cette même commande pour vous assurer que les tests sont toujours valides et que toute modification de l'application n'induit pas de régression vis-à-vis du comportement spécifié.
Pour supprimer le code compilé et ainsi "nettoyer" votre projet, vous pouvez utiliser la commande `mvn clean` qui supprimer proprement le répertoire `target/`.
Par défaut, l'archetype Maven qui crée le projet génère un fichier `pom.xml` qui se base sur JUnit 3, pour utiliser une version plus récente de JUnit, vous devez donc modifier le contenu de ce fichier `pom.xml` pour remplacer les lignes qui concernent JUnit par[1] :

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.6</version>
      <configuration>
        <includes>
          <include>**</include>
        </includes>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Vous pouvez remplacer le code JUnit 3 généré par défaut de la classe `AppTest` par le code JUnit 4 suivant pour vérifier le fonctionnement du plugin :

```
package fil.coo;
import org.junit.*;
import static org.junit.Assert.*;
public class AppTest {
```

[1] voir sur le portail le fichier `source-tp.maven.txt` fourni.

```
    @Test
    public void test() {
  assertTrue(true);
    }
}
```

Pour constater ce que mentionne Maven en cas d'erreur dans les tests, vous pouvez ajouter un appel à `fail()` dans la méthode `test()` avant de relancer la commance `mvn package`.
De même vous pouvez ajouter une erreur de syntaxe dans le fichier `App.java` avant d'exécuter cette commande pour constater les messages d'erreur de compilation.

### Étape n°3 : Générer la Javadoc

Maven peut également de générer automatiquement la Javadoc de votre projet. Pour ce faire, il vous faut modifier le fichier `pom.xml` de votre projet pour y insérer les lignes suivantes **au sein de la balise `<plugins>` présente dans la balise `<build>`** :

```
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-javadoc-plugin</artifactId>
      <version>2.10.4</version>
      <configuration>
        <reportOutputDirectory>${project.build.directory}/docs</reportOutputDirectory>
        <destDir>docs</destDir>
        <nohelp>true</nohelp>
      </configuration>
    </plugin>
```

En exécutant la commande `mvn javadoc:javadoc`, Maven génère la Javadoc au format HTML dans le répertoire `docs`.
Cette génération doit se dérouler sans message d'erreur ni warning de la part de l'outil javadoc.

### Étape n°4 : Produire une archive exécutable

En exécutant la commande `package`, Maven produit une archive de votre projet qui contient le *bytecode* (code interprétable par la machine virtuelle Java) de votre application, prêt à être exécuter. Pour éviter de saisir la ligne de commande complète pour lancer votre application, il est possible de faire en sorte que cette archive soit exécutable. Pour ce faire, il vous faut modifier le fichier `pom.xml` de votre projet pour y ajouter un autre *plugin* en insérant les lignes suivantes **à nouveau au sein de la balise `</plugins>`** :

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <version>3.0.2</version>
  <configuration>
    <archive>
      <manifest>
        <mainClass>fil.coo.App</mainClass>
      </manifest>
    </archive>
  </configuration>
</plugin>
```

Il vous faudra évidemment remplacer la valeur de la « `mainClass` » par le nom de la classz appropriée pour votre projet.

À partir de maintenant, votre application pourra être simplement exécutée en utilisant la commande `java -jar` comme suit :

```
$COO-TP1$ mvn package
...
$COO-TP1$ java -jar target/COO-TP1-1.0-SNAPSHOT.jar
Hello World!
```

Maven intègre beaucoup d'autres commandes et de plugins pour adapter le processus de compilation à vore convenance, vous pouvez consulter la documentation en ligne pour en savoir plus : `http://maven.apache.org`.
Maintenant que vous avez une structure de projet propre qui permet de stocker et organiser vos développements, nous allons voir comment gérer les différentes versions de ce code et partager votre projet avec vos collaborateurs en utilisant GitLab.

## GitLab

GitLab est un gestionnaire de projets de développements collaboratifs. Il intègre notamment une panoplie d'outils visant à faciliter les différents aspects lié au développement d'une application que sont la gestion des versions du code, la collaboration entre plusieurs contributeurs, la documentation et le partage du projet. Le département d'Informatique de l'Université de Lille met à votre disposition un dépôt GitLab que vous pouvez utiliser librement pour réaliser vos TPs et projets en collaborant efficacement avec vos camarades. Ce service est diponible à l'adresse suivante : `http://gitlab-etu.fil.univ-lille1.fr` et vous pouvez vous y connecter avec vos codes d'accès habituels. Vous pouvez également utiliser ce service pour rendre vos TPs et projets à votre chargé de TD/TP. Attention, les dépôts que vous avez créés sont automatiquement supprimés *en fin d'année*.
Cette fiche vous fournit donc un petit exemple d'utilisation de GitLab et Maven pour vous aider à débuter vos projets de développement sur de bonnes bases. Pour des explications sur le fonctionnement de GitLab ou des usages avancés, vous pouvez consulter la documentation en ligne de GitLab (`https://about.gitlab.com`)

### Étape n°5 : Créer son dépôt distant

Connectez-vous sur l'interface en ligne de GitLab puis cliquez sur New project. Indiquez un Project name pour votre projet, par exemple `COO-TP1`, cocher qu'il s'agit d'un projet Private, puis validez en cliquant sur Create project. Vous pouvez ensuite donner des droits à votre binôme sur ce dépôt pour qu'il puisse accéder et contribuer au code que vous allez développer collaborativement. Pour ce faire, cliquez sur le menu Members (en haut à droite, au niveau de la roue crantée), renseignez l'identifiant de votre binôme (son *login*), changer les droits d'accès pour Master, puis cliquez sur le bouton Add users to project. Renouvelez l'opération pour enregistrer votre chargé de TP/TP en tant que Developer afin qu'il puisse suivre et annoter vos développements.

### Étape n°6 : Synchroniser son dépôt local

Votre projet est désormais créé sur le dépôt distant et vous allez pouvoir le synchroniser avec votre machine. Avant cela, afin de faciliter l'authentification et vous éviter de saisir votre mot de passe lors de chaque opération, vous pouvez enregistrer votre clé publique SSH qui sera utilisée par GitLab pour vous authentifier. Pour ce faire, cliquez sur le menu Profile Settings puis sur l'onglet SSH Keys. Dans le champ Key, copiez votre clé publique SSH en faisant un copier/coller du contenu retourné par la commande:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEAklOUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU
GPl+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7xlELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7kOdJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3VOWw68/eIFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW4OZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== toto@laptop.local
```

Si jamais, le système d'exploitation vous indique que le fichier n'existe pas, alors vous devez créer une paire de clés publique/privée en utilisant la commande:

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/toto/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/toto/.ssh/id_rsa.
Your public key has been saved in /Users/toto/.ssh/id_rsa.pub.
The key fingerprint is:
43:c5:5b:5f:b1:f1:50:43:ad:20:a6:92:6a:1f:9a:3a toto@laptop.local
```

Une fois votre paire de clés SSH créée, vous pouvez copier le contenu de la clé publique (le fichier `.pub`) dans GitLab pour qu'il vous reconnaisse ensuite. Attention, il est possible que GitLab prenne quelques secondes ou minutes pour reconnaître votre clé publique SSH. Vous pouvez créer et enregistrer dans GitLab plusieurs clés publiques SSH selon que vous travaillez sur une machine de l'Université ou chez vous avec votre machine personnelle.
Vous pouvez maintenant synchroniser votre dépôt local avec le dépôt distant créé sur GitLab. Pour ce faire, positionnez vous dans le répertoire du projet puis exécutez les commandes suivantes :

```
$COO-TP1$ git init
Initialized empty Git repository in /Users/[MON_LOGIN]/COO/COO-TP1/.git/
$COO-TP1$ git remote add origin git@gitlab-etu.fil.univ-lille1.fr:[MON_LOGIN]/COO-TP1.git
```

La première commande permet d'activer Git sur votre projet créé avec Maven. Vous avez donc un dépôt Git local qui est considéré comme vide car aucun fichier n'a été enregistré pour l'instant. La deuxième commande configure l'adresse du dépôt distant (celui de GitLab) sur lequel vous pourrez partager votre code.

### Étape n°7 : Ajouter des fichiers

Un dépôt Git est destiné à héberger du code source et non pas des artefacts compilés. Pour éviter de stocker de tels fichiers par erreur, nous allons enregistrer un fichier `.gitignore` qui va indiquer à Git les fichiers qu'il doit ignorer. La façon la plus simple de générer ce fichier est de vous rendre sur le site `https://www.gitignore.io`, d'indiquer que vous réalisez un projet `maven`, cliquer sur le bouton `generate` et de copier le contenu qui vous est retourné dans un fichier `.gitignore` stocké à la racine de votre projet:

```
$COO-TP1$ touch .gitignore
$COO-TP1$ vi .gitignore
```

Git détecte que des fichiers se trouvent dans le répertoire courant mais il ne le prend pas en compte dans la gestion des versions :

```
$COO-TP1$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
# .gitignore
# pom.xml
# src/
nothing added to commit but untracked files present (use "git add" to track)
```

Pour enregistrer ces fichiers et répertoires dans Git, il faut d'abord les ajouter au dépôt en utilisant la commande :

```
$COO-TP1$ git add .gitignore pom.xml src/
```

Ensuite, nous allons lui donner un numéro de version en exécutant la commande :

```
$COO-TP1$ git commit -m "feat: Bootstrapping the project."
[master (root-commit) 7336815] feat: Bootstrapping the project
 4 files changed, 90 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 pom.xml
 create mode 100644 src/main/java/fil/coo/App.java
 create mode 100644 src/test/java/fil/coo/AppTest.java
```

Le message de *commit* passé en paramètre n'est pas à négliger car il doit synthétiser la nature de la modification qui a été apportée sur le code du projet. Il existe des conventions que nous vous encourageons fortement de suivre : `https://github.com/angular/angular.js/blob/master/CONTRIBUTING.md#commit`

À cet instant, si vous vous connectez sur l'interface de GitLab ou que votre binôme essayer de cloner votre dépôt (ou celui de votre binôme) en utilisant la commande :

```
git clone git@gitlab-etu.fil.univ-lille1.fr:[LOGIN_BINOME]/COO-TP1.git
```

Il récupérera un projet vide. En effet, les fichiers sont versionnés par Git mais pour autant les fichiers ne sont toujours pas visibles dans l'interface de GitLab. Pour ce faire, il faut donc synchroniser votre dépôt local avec le dépôt distant en exécutant la commande :

```
$COO-TP1$ git push -u origin master
Counting objects: 15, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (15/15), 1.58 KiB | 0 bytes/s, done.
Total 15 (delta 0), reused 0 (delta 0)
To git@gitlab-etu.fil.univ-lille1.fr:[MON_LOGIN]/COO-TP1.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

### Étape n°8 : Récupérer des fichiers

Les fichiers sont désormais disponibles en ligne et votre binôme peut les récupérer en exécutant la commande :

```
$COO-TP1$ git pull
remote: Counting objects: 15, done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 15 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (15/15), done.
From gitlab-etu.fil.univ-lille1.fr:[MON_LOGIN]/COO-TP1
 * [new branch]      master     -> origin/master
$COO-TP1$ ls
pom.xml src
```

**Pour résumer**

À chaque fois que vous souhaiterez partager un fichier avec votre binôme, il suffira donc de l'éditer localement puis d'exécuter la séquence de commandes :

```
$COO-TP1$ git add mon_fichier.ext mon_repertoire/
$COO-TP1$ git commit -m "Mon message de commit"
$COO-TP1$ git push
```

Et quand vous voudrez récupérer les modifications qui ont été enregistrées sur le dépôt distant, il vous suffira d'exécuter la commande :

```
$COO-TP1$ git pull
```

Bien entendu, Git ne se limite pas à ces quelques commandes. N'hésitez pas à consulter la documentation de Git pour découvrir son fonctionnement et les différentes commandes utilisables : `https://git-scm.com/documentation`.

## Projets

**Les rendus de vos projets se feront via votre dépôt Gitlab**.
Vous ajouterez systématiquemment à votre projet un fichier `README.md` (syntaxe markdown possible et recommandée) sur la branche master. Celui-ci aura pour rôle de présenter votre projet.
Vous devrez avoir utilisé les fonctions de base de Maven pour gérer le projet.
Assurez-vous qu'en exécutant sur votre dépôt la commande
`git clone ...`' (ou '`git pull`')
suivie de
`mvn package`
votre projet est bien récupéré et généré (doc, tests et jar exécutable) sans message d'erreur. Un `java -jar target/xxx.jar` doit ensuite suffire à l'exécuter.
Pour l'évaluation de votre projet, votre enseignant exécutera en premier lieu ces commandes. Vous serez fortement pénalisé si elles ne fonctionnement pas correctement.

## Foire Aux Questions

**1. Je n'ai plus de place sur mon compte après avoir lancé Maven**

Maven a la mauvaise habitude de télécharger ses dépendances dans le répertoire `~/.m2/repository/` qui se trouve à la racine de votre compte personnel. Pour éviter que Maven ne vous fasse franchir votre quota de disque, vous pouvez créer un lien symbolique vers un répertoire local temporaire de votre choix en exécutant la commande :

```
$$ rm -rf ~/.m2/repository/
$$ mkdir /tmp/mvn-repository
$$ ln -s /tmp/mvn-repository ~/.m2/repository
```

**2. La processus Maven s'arrête avec une erreur bizarre**

Regardez à votre quota disque et suivre la réponse à la question 1.

**3. Doit-on obligatoirement utiliser Maven dans nos projets de COO ?**

Oui.

**4. Doit-on obligatoirement utiliser GitLab pour rendre nos projets de COO ?**

Oui.