

TP Industrialisation
Maven / GIT /
Jenkins / SONAR /
....:

Introduction Maven,
git et Intégration
Continue

Objectifs du TP

Comprendre le fonctionnement de maven.

Utiliser les artifacts

Configurer un projet eclipse avec maven.

Créer son propre MOJO

Générer des rapports maven

Utiliser Git pour sauvegarder le code source de votre projet

Utiliser un système d'Intégration Continue

Liens utiles

- Site de Maven : <http://maven.apache.org/>
- Plugin Checkstyle : <http://maven.apache.org/plugins/maven-checkstyle-plugin/>
- FAQ MAVEN developpez.com : <http://java.developpez.com/faq/maven/>

Partie 1 : Utilisation de maven

Création d'une application basique.

Pour initialiser un projet Java, vous pouvez utiliser l'archetype maven *maven-archetype-quickstart*. Vous avez juste à fournir un groupId et un artefactId.

Dans eclipse.

new -> other -> maven -> maven project. Vous devez sélectionner l'archetype, l'artifactId et le groupId

En ligne de commande (non nécessaire si vous l'avez fait depuis eclipse)

```
mvn archetype:create \  
-DgroupId=[your project's group id] \  
-DartifactId=[your project's artifact id] \  
-DarchetypeArtifactId=maven-archetype-quickstart
```

Ou simplement,

```
mvn archetype:create \  
-DgroupId=[your project's group id] \  
-DartifactId=[your project's artifact id]
```

Vous obtenez la structure de projet jointe

```
|-- src  
| |-- main  
| |   |-- java  
| |       |-- [your project's package]  
| |           |-- App.java  
| |-- test  
| |   |-- java  
| |       |-- [your project's package]  
| |           |-- AppTest.java  
|-- pom.xml
```

Par exemple si vous exécutez la commande

```
mvn archetype:generate \  
-DgroupId=fr.istic.master1.sir \  
-DartifactId=tpmaven
```

Vous obtiendrez l'architecture suivante.

```
|-- src  
| |-- main  
| |   |-- java  
| |   |   |-- fr  
| |   |       |-- istic  
| |   |           |-- master1  
| |   |               --sir  
| |   |                   |-- App.java  
| |-- test  
| |   |-- java  
| |   |   |-- fr  
| |   |       |-- istic  
| |   |           |-- master1  
| |   |               --sir  
| |   |                   |-- AppTest.java  
|-- pom.xml
```

Partie 2 : Configuration d'eclipse

Pour eclipse 4.X

Dépuis eclipse 4.X, le support de maven s'est amélioré. Pour importer votre projet. File -> import -> maven -> existing maven project.

Votre projet est configuré.

Partie 3: Gestion des dépendances

Intégrer à votre code source le fichier suivant

<https://raw.githubusercontent.com/barais/swingx/master/JXBusyLabelTest>

Vous verrez que le code ne compile pas car il manque une dépendance.

Intégrez maintenant la dépendance à swingx (dans le pom.xml).

```
<!-- https://mvnrepository.com/artifact/org.swinglabs/swingx -->
<dependency>

  <groupId>org.swinglabs</groupId>

  <artifactId>swingx</artifactId>

  <version>1.6.1</version>

</dependency>
```

Votre IDE va downloader la dépendance et la mettre automatiquement dans votre classpath. Dans ce sens, cela permet de ne mettre dans votre gestionnaire de source que le code source et le descripteur de projet (pom.xml).

Partie 4: Spécialisation du process de build.

Imaginons que vous souhaitiez ajouter une tâche dans le processus de build. Par exemple, compilez votre code source avec la version Java 1.7 (ou 1.8)

Ajoutez la section suivant à votre fichier pom.xml

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <!-- or whatever version you use -->
        <source>1.7</source>
        <target>1.7</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Si vous avez une erreur, vous pouvez la résoudre avec un « Clic droit >> Maven >> Update project »

Vous pouvez ajouter de nombreux plugin dans cette section.

<https://maven.apache.org/plugins/>

Partie 5 : Génération de rapports

Générer la javadoc

Ajoutez des commentaires à votre code projet tp1

Ajouter le code suivant dans le pom.xml de votre projet

```
<reporting>
  <plugins>

    <plugin>

      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-javadoc-plugin</artifactId>

    </plugin>
  </plugins>
</reporting>
```

Puis lancez : mvn site

Si ça ne marche pas, ajoutez dans <build> <plugins> ... </plugins></build>

```
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-site-plugin</artifactId>
    <version>3.7.1</version>

  </plugin>
```

Valider la qualité du code avec le plugin checkstyle

- Ajoutez à la section <reporting> du projet client le plugin checkstyle

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-checkstyle-plugin</artifactId>
    </plugin>
  </plugins>
</reporting>
```

Lancez : mvn site

Observez la section 'reporting' ajoutée dans le rapport récupéré dans \target\site\index.html

Quelle est la norme de codage à laquelle se réfère le rapport par défaut ?

1/ Comment imposer la norme de codage de Google?

Fichier de définition

https://raw.githubusercontent.com/checkstyle/checkstyle/master/src/main/resources/google_checks.xml

Pour Changer la version de checkstyle

voir => <https://maven.apache.org/plugins/maven-checkstyle-plugin/examples/upgrading-checkstyle.html>

pour Changer le fichier de configuration à utiliser

<https://maven.apache.org/plugins/maven-checkstyle-plugin/examples/custom-checker-config.html>

rajouter dans le « maven-checkstyle-plugin »

```
<configuration>
    <configLocation>google_checks.xml</configLocation>
</configuration>
```

Si ça ne marche pas, essayez avec la version 8.12 du google_checks

https://github.com/checkstyle/checkstyle/blob/checkstyle-8.12/src/main/resources/google_checks.xml

2/ Modifiez votre classe du projet tp1 de façon à diminuer le nb d'erreur ? Liens utiles :

Site de de l'outil CheckStyle : <http://checkstyle.sourceforge.net/>

Site du plugin Maven : <http://maven.apache.org/plugins/maven-checkstyle-plugin/>

Rapport croisé de source

```
<reporting>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-jxr-plugin</artifactId>
        </plugin>
    </plugins>
</reporting>
```

Lien utile : <http://maven.apache.org/plugins/maven-jxr-plugin/>

Quelle est la valeur ajoutée de ce plugin ? En particulier, montrez sa complémentarité avec CheckStyle

Désormais vous pouvez passer du rapport CheckStyle au code source en cliquant sur le numéro de ligne associé au commentaire CheckStyle.

Couverture de test

A quel point les développeurs ont réalisé des tests unitaires ? Quelles parties de l'application n'ont pas été testées ?

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>cobertura-maven-plugin</artifactId>
    </plugin>
  </plugins>
</reporting>
```

Lien utile : <http://mojo.codehaus.org/cobertura-maven-plugin/usage.html>

Identifier patterns d'erreur avec PMD (PMD is a source code analyzer)

<https://pmd.github.io/>

Ajoutez volontairement du code mort à votre code projet tp1 (une méthode pas utilisée par exemple)

Identifiez :

Code mort (Ex : variables ou paramètres non utilisés)

Duplication de code (Code copié/collé = possible bug copié/collé Code 'compliqué' (Ex : trop de if...else)

Ajoutez à la section <reporting> du projet persist le plugin PMD

```
<project>
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-pmd-plugin</artifactId>
        <version>2.5</version>
      </plugin>
    </plugins>
  </reporting>
</project>
```

Quels sont les deux nouveaux rapports générés ?

Qu'est ce que le rapport 'CPD Report' ?

Qu'est ce que le rapport 'PMD Report' ?

Connaître l'activité du projet

Quels et combien de fichiers modifiés par un développeur ?

Commitez votre projet sur github ou sur la forge/gitlab de l'ensai.

- Ajoutez à la section <reporting> du projet persist le plugin changelog

```
<project>
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-changelog-plugin</artifactId> </plugin>
      </plugins>
    </reporting>
  </project>
```

```
<scm>
<connection>scm:svn:svn://IPSERVEUR/repository1/monappli-web</connection>
<url>http://IPSERVEUR/svn/monappli-web</url>
</scm>
</project>
```

ipserveur doit être remplacé par l'IP de votre repo svn ou git.

Pour git

```
<scm>
  <connection>scm:git:https://github.com/PATH</connection>
  <developerConnection>scm:git:https://github.com/PATH </developerConnection>
  <url>scm:git:https://github.com/PATH </url>
</scm>
```

- Lancez : mvn site

- Que s'est-il passé ?

Le répertoire /target/site situé dans votre projet contient maintenant trois rapports d'activité :

- changelog : rapport indiquant toutes les activités sur le SCM.
- dev-activity : rapport indiquant par développeur le nombre de commits, de fichiers modifiés.
- file-activity : rapport indiquant les fichiers qui ont été révisés.

Partie 7 : Créez votre propre MOJO

Il y a des outils qui sont bien pratiques, voire parfois nécessaires à tout développeur qui se respecte. Un logiciel de gestion de build comme Ant ou Maven en fait partie. Outre l'avantage indéniable de permettre de télécharger la moitié de l'internet lors du premier build sur une machine *clean*, Maven permet aussi via son système de plugins d'effectuer tout un tas d'actions allant plus loin que la compilation des classes et leur packaging dans un JAR. Je vous propose de regarder comment créer son premier plugin Maven.

D'abord il faut un besoin

Eh oui, parce que se contenter de créer un plugin qui affiche "Hello World" à chaque build, ce n'est pas très utile... Alors tant qu'à faire, mettons nous en situation réelle. Prenons un plugin qui compte le nombre de classes et d'attributs/méthodes par classe dans notre code.

Création du squelette de plugin Maven

En ligne de commande.

```
mvn archetype:generate -DgroupId=fr.sir.tpmaven -DartifactId=classcounter-maven-plugin -DarchetypeArtifactId=maven-archetype-mojo
```

Puis import dans eclipse

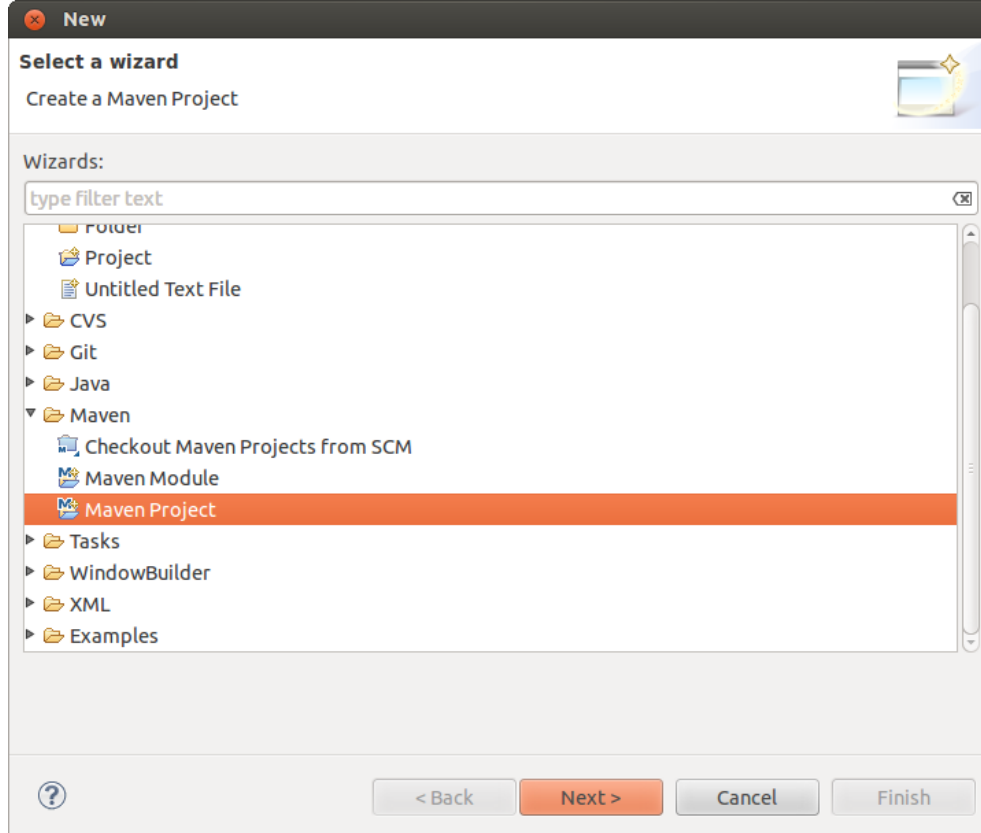
File -> import -> maven -> import existing maven project -> select the generated project.

Sous Eclipse

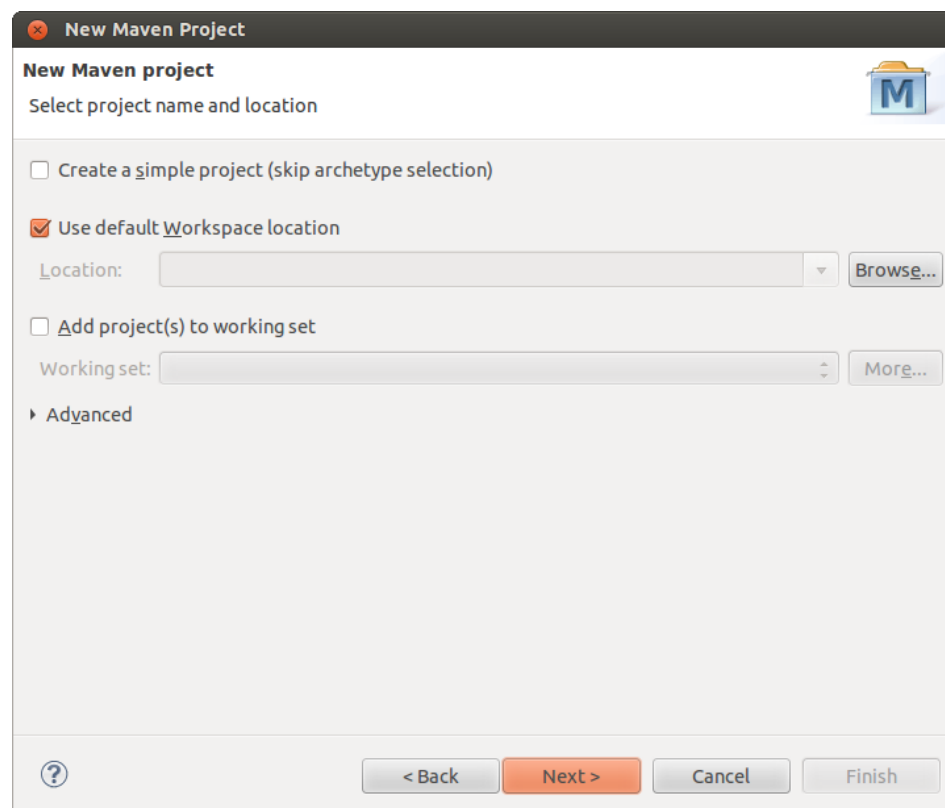
Un plugin Maven n'est rien d'autre qu'un projet Maven avec un *packaging* de type **maven-plugin** :

Sous eclipse 4.X

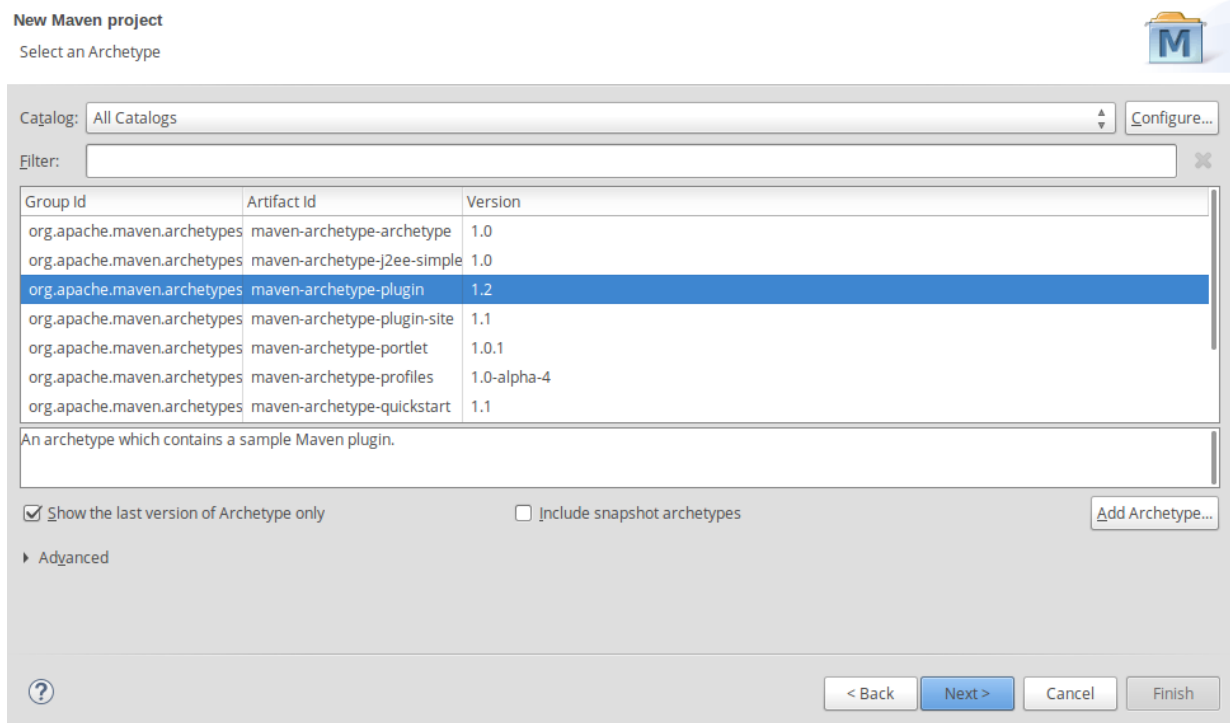
Créez un projet Maven.



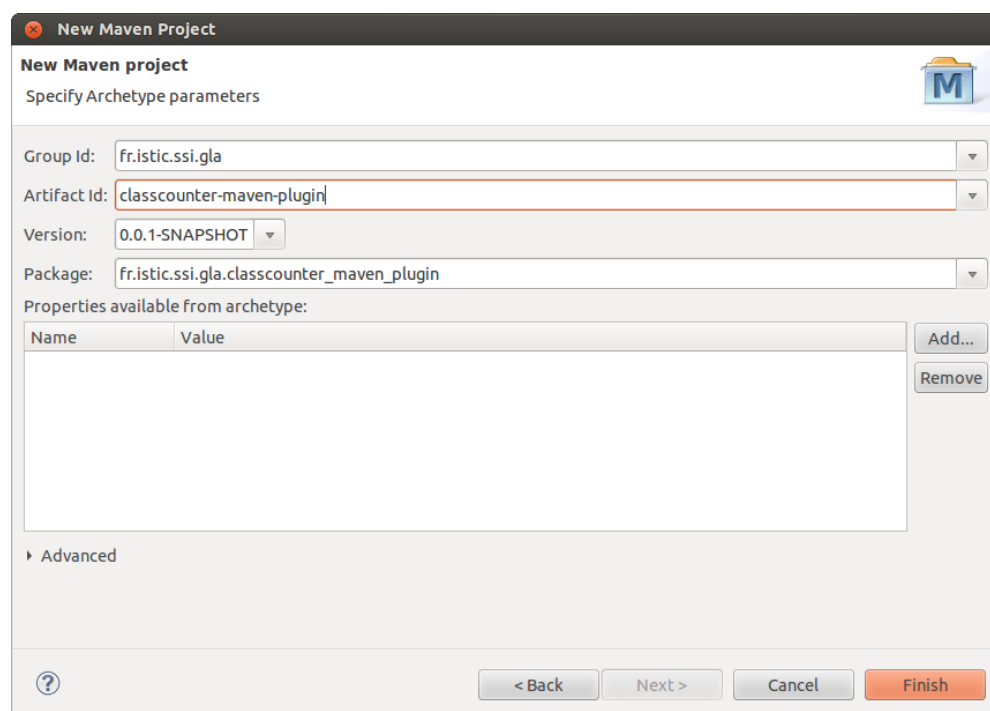
à l'écran suivant, vous pouvez laisser les options par défaut.



Ensuite choisissez le mojo du projet (template de création du projet). Pour créer un projet de plugin pour maven il faut sélectionner l'archetype `org.apache.maven.archetypes:maven-archetype-plugin`



Ensuite renseignez le nom du projet (group id et artifact id).



Pour eclipse 4.X

La convention veut que les plugins soient nommés *xyz-maven-plugin*, ce qui permet d'appeler les goals de la manière suivante lors d'un build, il faut aussi que le **groupId** soit `org.apache.maven.plugins` :

```
$ mvn classcounter:some-goal
Or
$ mvn classcounter-maven-plugin:some-goal
```

Notez qu'il n'est pas nécessaire de spécifier le nom complet du plugin, Maven va compléter avec `"-maven-plugin"`, ça fait quelques caractères en moins à écrire 😊. Si la convention de nommage n'est pas suivie, il faut impérativement spécifier le nom complet du plugin dans la ligne de commande.

Le projet du MOJO contient le POM suivant.

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

  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>classcounter-maven-plugin</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>maven-plugin</packaging>
</project>
```

Ensuite, un plugin contient un ou plusieurs MOJO, qui vont exécuter le “vrai” code du plugin. Un MOJO correspond en fait à un *goal*. Par exemple, dans le plugin `dependency-maven-plugin`, on retrouve les goals “[tree](#)” (`mvn dependency:tree`) et “[resolve](#)” (`mvn dependency:resolve`). Créons donc un squelette de MOJO pour notre générateur de code :

Notre MOJO étend `AbstractMojo`, qui fournit toute l’infrastructure utile à la création d’un goal. La méthode abstraite `execute()` contient le code à exécuter, c’est donc à nous de l’implémenter. Pour l’instant elle ajoute une ligne de log saluant le monde (je sais, j’ai dit tout à l’heure qu’on ne se contenterait pas de faire un hello world...). La méthode `getLog()` est un exemple de facilités fournies par l’`AbstractMojo`.

```
package fr.istic.sir.classcounter_maven_plugin;

import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin MojoExecutionException;
import org.apache.maven.plugins.annotations.Execute;
import org.apache.maven.plugins.annotations.LifecyclePhase;
import org.apache.maven.plugins.annotations.Mojo;
import org.apache.maven.plugins.annotations.Parameter;

@Mojo( name = "count" )
@Execute( goal = "count",
  phase = LifecyclePhase.COMPILE )

public class ClassCounterMojo
extends AbstractMojo
{

  public void execute() throws MojoExecutionException
  {
    getLog().info("Hello World! ");
  }
}
```

Notez également les annotations, qui contiennent deux tags particuliers. @goal spécifie le nom du goal correspondant à notre MOJO, cette annotation tag est obligatoire. @phase permet de dire à Maven que notre plugin intervient durant la phase de génération des sources du [lifecycle Maven](#)(avant la compilation des sources, donc).

<https://maven.apache.org/guides/plugin/guide-java-plugin-development.html>

Pour utiliser notre plugin depuis un autre projet, il faut d'abord l'installer dans notre dépôt local :

```
$ mvn install
```

Notre plugin est prêt à être exécuté. Configurons le pom.xml de notre projet *tpmaven* pour permettre l'exécution du plugin lors du build :

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>classcounter-maven-plugin</artifactId>
      <version>1.0-SNAPSHOT</version>
    </plugin>
  </plugins>
</build>
```

Puis lançons un build Maven :

```
$ mvn classcounter:count
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building classcounter-maven-plugin 1.0-SNAPSHOT
[INFO] -----
[INFO]
[INFO] --- classcounter-maven-plugin:1.0-SNAPSHOT:count (default-cli) @ classcounter-
maven-plugin ---
[INFO] Hello World!
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 0.322s
[INFO] Finished at: Mon May 14 14:47:54 CEST 2014
[INFO] Final Memory: 2M/81M
[INFO] -----
```

Ajout de paramètres au plugin

Notre cahier des charges requiert la possibilité de configurer certaines parties du générateur. Pour cela, ajoutons des paramètres à notre MOJO:

```
package fr.istic.sir.classcounter_maven_plugin;

import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoExecutionException;
import org.apache.maven.plugins.annotations.Execute;
import org.apache.maven.plugins.annotations.LifecyclePhase;
import org.apache.maven.plugins.annotations.Mojo;
import org.apache.maven.plugins.annotations.Parameter;

@Mojo( name = "count")
@Execute( goal = "count",
phase = LifecyclePhase.COMPILE)

public class ClassCounterMojo
extends AbstractMojo
{
    @Parameter(defaultValue="french")
    String language;

    public void execute() throws MojoExecutionException
    {
        getLog().info("Indication will be given in the following language " + language);
        getLog().info("Hello World! ");
    }
}
```

Notre attribut utilise également des tags Javadoc pour indiquer que c’est un paramètre du plugin. Ce paramètre est obligatoire et prend une valeur par défaut.

L’attribut “alias” de @parameter permet de définir le nom du paramètre s’il est différent du nom de l’attribut Java. Ainsi, par défaut il y aura un paramètre nommé “language”, puisqu’on ne lui a pas donné d’alias. Une liste plus exhaustive des tags Javadoc est disponible sur le [site de Sonatype](#).

Ouvrons à nouveau le pom.xml de notre projet pour ajouter la configuration du plugin :

```
<build>
  <plugins>
    <plugin>
      <groupId>fr.istic.master1.sir</groupId>
      <artifactId>classcounter-maven-plugin</artifactId>
      <version>1.0-SNAPSHOT</version>
      <configuration>
        <language>english</language>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Pour vérifier que la configuration est bien injectée au runtime, ajoutons un log :

```
getLog().info("Indication will be given in the following language " + language);
mbp:GeneratorMavenPlugin bastien$ mvn classcounter:count
...
[INFO] --- classcounter-maven-plugin:1.0-SNAPSHOT:count (default-cli) @ classcounter-
maven-plugin ---
[INFO] Hello World!
[INFO] Indication will be given in the following language English
[INFO] -----
[INFO] BUILD SUCCESS
...
```

Maintenant que nous savons créer, configurer et exécuter un plugin Maven, il ne reste plus qu'à implémenter la méthode `execute()` pour appeler les vrais services.

Nous allons utiliser une librairie externe pour compter le nombre de classe.

Dans le pom.xml de votre plugin, ajoutez la dépendance à clapper

```
<dependency>
  <groupId>org.clapper</groupId>
  <artifactId>classutil_2.10</artifactId>
  <version>1.2.0</version>
</dependency>
```

Vous pouvez ensuite vous inspirer de

```

package fr.istic.sir.classcounter_maven_plugin;

import java.io.File;
import java.util.ArrayList;
import java.util.List;

import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoExecutionException;
import org.apache.maven.plugins.annotations.Execute;
import org.apache.maven.plugins.annotations.LifecyclePhase;
import org.apache.maven.plugins.annotations.Mojo;
import org.apache.maven.plugins.annotations.Parameter;
import org.clapper.classutil.ClassFinder;
import org.clapper.classutil.ClassInfo;

@Mojo( name = "count")
@Execute( goal = "count",
phase = LifecyclePhase.COMPILE)

public class ClassCounterMojo
extends AbstractMojo
{
    @Parameter(defaultValue="french")
    String language;

    @Parameter( defaultValue = "${project.build.directory}", readonly = true )
    private File outputDirectory;

    public void execute()
    throws MojoExecutionException
    {
        File f = outputDirectory;
        File f1 = new File(outputDirectory.getAbsolutePath(),"classes");
        List<File> files = new ArrayList<File>();
        files.add(f1);
        ClassFinder finder = new
ClassFinder(scala.collection.JavaConversions.asScalaBuffer(files));
        if ("french".equals(language))
            this.getLog().info("nombre de classe " + finder.getClasses().size());
        else
            this.getLog().info("number of classe " + finder.getClasses().size());
        scala.collection.Iterator<ClassInfo> it = finder.getClasses().toIterator();
        while (it.hasNext()){
            ClassInfo c = it.next();
            if ("french".equals(language))
            {
                this.getLog().info("\t Pour la classe " + c.name());
                this.getLog().info("\t \t Nbre attributs " + c.fields().size());
                this.getLog().info("\t \t Nbre methodes " + c.methods().size());
            }else{
                this.getLog().info("\t For the class named " + c.name());
                this.getLog().info("\t \t Number of filed " + c.fields().size());
                this.getLog().info("\t \t Number of methods " +
c.methods().size());
            }
        }
    }
}

```


Comme vous avez pu le voir, créer un plugin Maven n'est pas très compliqué. Les plugins sont suffisamment souples et configurables pour pouvoir exécuter du code déjà existant (par exemple remplacer un `main()` par un plugin Maven) sans trop de difficultés. La palette de plugins déjà disponibles est assez vaste, mais si un jour vous avez un besoin spécifique vous saurez que le coût de création d'un plugin n'est pas très élevé.

Enfin, il existe plein de paramètre au niveau des annotations. Voici un exemple de Mojo plus riche.

```
package fr.istic.sir.classcounter_maven_plugin;
import java.io.File;

import org.apache.maven.execution.MavenSession;
import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoExecution;
import org.apache.maven.plugin.descriptor.PluginDescriptor;
import org.apache.maven.plugins.annotations.Execute;
import org.apache.maven.plugins.annotations.LifecyclePhase;
import org.apache.maven.plugins.annotations.Mojo;
import org.apache.maven.plugins.annotations.Parameter;
import org.apache.maven.plugins.annotations.ResolutionScope;
import org.apache.maven.project.MavenProject;
import org.apache.maven.settings.Settings;

@Mojo( name = "MyMojo",
      aggregator = true,
      executionStrategy = "always",
      inheritByDefault = true,
      instantiationStrategy = InstantiationStrategy.SINGLETON,
      defaultPhase = LifecyclePhase.COMPILE,
      requiresDependencyResolution = ResolutionScope.NONE,
      requiresDependencyCollection = ResolutionScope.NONE, // (since Maven 3.0)
      requiresDirectInvocation = false,
      requiresOnline = false,
      requiresProject = true,
      threadSafe = true ) // (since Maven 3.0)
@Execute( goal = "test",
        phase = LifecyclePhase.COMPILE)
public class MyMojo
extends AbstractMojo
{
    @Parameter( name = "parameter",
              alias = "myAlias",
              property = "a.property",
              defaultValue = "an expression, possibly with ${variables}",
              readonly = false,
              required = false )
    private String parameter;

    // sample objects taken from Maven API through PluginParameterExpressionEvaluator

    @Parameter( defaultValue = "${session}", readonly = true )
    private MavenSession session;

    @Parameter( defaultValue = "${project}", readonly = true )
    private MavenProject project;

    @Parameter( defaultValue = "${mojoExecution}", readonly = true )
    private MojoExecution mojo;
```

```
@Parameter( defaultValue = "${plugin}", readonly = true ) // Maven 3 only
private PluginDescriptor plugin;
```

```
@Parameter( defaultValue = "${settings}", readonly = true )
private Settings settings;
```

```
@Parameter( defaultValue = "${project.basedir}", readonly = true )
private File basedir;
```

```
@Parameter( defaultValue = "${project.build.directory}", readonly = true )
private File target;
```

```
public void execute()
{
}
}
```

[1] <http://labs.excilys.com/2012/05/14/mon-premier-plugin-maven/>

[2] http://www.objis.com/formation-java/IMG/pdf/TP8_reporting.pdf

.....

Partie 8 : Intégration, avec l'outil Sonar

Téléchargez Sonar : <https://www.sonarqube.org/downloads/>

Installez Sonar : <https://docs.sonarqube.org/display/SONAR/Documentation>

Lancez : mvn sonar:sonar



Utilisation dans un contexte d'intégration continue

Nous allons ici observer les bénéfices pour l'utilisation d'un outil d'intégration continue Jenkins.

Commitez votre code sur github (public) ou bitbucket(privé)

Partie 9 : Intégration Continue

Partie 9.1 : Gestion de version (Git)

Placez votre code sur github ou gitlab.

Partie 9.2 : Jenkins

Téléchargez Jenkins.

<http://jenkins-ci.org/>

Prenez la version **Java Web Archive (.war)**

Démarrez jenkins

```
> java -jar jenkins.war --httpPort=9900
```

Allez dans votre navigateur : <http://localhost:9900>.

Configurez jenkins pour compiler et exécuter votre projet.

<https://medium.com/@anusha.sharma3010/build-a-simple-maven-project-in-jenkins-da7a2a4ae202>

<https://medium.com/@amitvermaa93/jenkins-github-with-java-maven-project-c17cdba7062>

Par défaut, jenkins ne contient pas le plugin pour gérer des repository Git, Il vous faut installer le plugin “Git Plugin”. De plus, vous devez configurer Maven (voir Configure System)

Ensuite créer un job, définissez les sources en indiquant l’url du repository git que vous avez préalablement créé sur github ou bitbucket et enfin définissez les goals maven pour le build.

La configuration d’un job peut prendre de nombreux paramètres. Par exemple, vous pouvez définir la condition de déclenchement d’un build (“Build Triggers”), vous pouvez limiter la sauvegarde des anciens builds (“Discard old builds”), vous pouvez aussi envoyer des notifications lorsque le build échoue (“E-mail notification” qui nécessite une configuration dans “Configure System”).

À ce point, vous devez avoir un jenkins sur lequel il y a un job défini. Lancer un Build pour vérifier que vous avez bien configuré votre projet. Maintenant nous allons ajouter sur le jenkins, la possibilité d’utiliser sonar et cobertura ainsi que pmd.

[https://wiki.jenkins-ci.org/display/JENKINS/Sonar+plugin \(OUT DATED\)](https://wiki.jenkins-ci.org/display/JENKINS/Sonar+plugin+(OUT+DATED))

<https://wiki.jenkins.io/display/JENKINS/SonarQube+plugin>

<https://wiki.jenkins-ci.org/display/JENKINS/Cobertura+Plugin>

<https://wiki.jenkins-ci.org/display/JENKINS/PMD+Plugin>

Attention pour Cobertura, vous avez besoin de définir le format de sortie en xml.

Pour cela, il existe deux solutions:

- la première consiste à ajouter une option dans la définition du build maven: “-Dcobertura.report.format=xml”
- la deuxième consiste à modifier la configuration dans votre pom et d’ajouter l’option de configuration appropriée (voir sur la page de Cobertura plugin)

!!!! À la fin du TP, il est préférable que vous supprimiez le dossier ~/.hudson sauf si vous souhaitez conserver la configuration de votre jenkins