

BLIN Sébastien  
COLLIN Pierre-Henri



# École supérieure d'ingénieurs de Rennes

2ème Année  
Parcours Informatique

---

## Architecture Logicielle

Compte-Rendu TP n°1

---

Sous l'encadrement de :  
TEMPLE Paul

# 1 Descriptif du projet

Ce premier TP d'Architecture Logicielle nous a permis de mettre en place tout un processus de compilation de la gestion de dépendance et de l'installation du projet en local en passant par l'automatisation des tests avec *Maven*, à la relecture du code (QA) par un tiers, analyse statique avec *Sonar* jusqu'à l'automatisation des builds sur différentes plateformes avec *Jenkins*

## 2 Maven

### 2.1 Description

Maven est un outil de gestion et de compréhension de projets. Maven fournit différentes fonctionnalités de : Construction, documentation, rapport, gestion des dépendances, gestion des sources, mise à jour de projet et déploiement.

Chaque projet Maven est décrit par un fichier *pom.xml* présent à la racine du répertoire du projet. Ce fichier contient tous les éléments permettant de gérer le cycle de vie du projet.

### 2.2 Utilisation

#### 2.2.1 Génération de projets

Pour créer un simple projet Java, on utilise l'archetype (un template de projet qui permet de gagner du temps et de respecter une certaine convention) Maven suivant : *maven-archetype-quickstart*.

La commande utilisée est la suivante : *mvn archetype:generate*

On doit fournir un *groupId* (identifiant du groupe de projets) et un *artifactId* (identifiant du projet).

#### 2.2.2 Génération de rapports

Pour générer la javadoc, on utilise le plugin *maven-javadoc-plugin* qu'on ajoute dans la section `<reporting>` du *pom.xml* :

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-javadoc-plugin</artifactId>
  <version>2.9.1</version>
</plugin>
```

Ensuite, nous avons lancé la commande *mvn site* qui génère un site web pour le projet et qui contient la javadoc du projet.

Nous avons également ajouté le plugin *maven-checkstyle-plugin* qui est un outil de contrôle du code et qui permet de vérifier le style du code source. La norme de codage utilisée par défaut est celle de Sun Microsystems (*sun\_checks.xml*). Parmi les règles prédéfinies, on trouve aussi celles de Google (*google\_checks.xml*). Par ailleurs, il est tout à fait possible d'utiliser un ensemble de règles personnalisées. Une nouvelle section apparaît alors sur le site.

Enfin, nous avons ajouté le plugin *maven-jxr-plugin* qui permet de référencer et de trouver plus

facilement des lignes de code spécifiques. Ici, l'association avec le plugin checkstyle permet de lier automatiquement le numéro de la ligne pour chaque règle non respectée et le code source.

### 2.2.3 Génération de packages

Afin de personnaliser l'étape de compilation et l'étape de création des artefacts, nous avons ajouté le plugin *maven-compiler-plugin* dans la section `<build>` du `pom.xml` qui permet de spécifier à Maven les options de configuration pour l'étape de compilation. En effet, la section `<reporting>` est dédiée à la génération des rapports.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.1</version>
  <configuration>
    <source>1.7</source>
    <target>1.7</target>
  </configuration>
</plugin>
```

Nous avons également ajouté le plugin *maven-jar-plugin* qui permet à Maven de packager le projet.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <version>2.6</version>
</plugin>
```

Le dernier plugin ajouté est *maven-assembly-plugin* qui permet de créer des archives.

```
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>2.5.5</version>
  <configuration>
    <descriptors>
      <descriptor>src/assembly/assemblySrc.xml</descriptor>
      <descriptor>src/assembly/assemblyBin.xml</descriptor>
    </descriptors>
    <finalName>TP1-Maven-${project.version}</finalName>
  </configuration>
  <executions>
    <execution>
      <id>make-assembly</id>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

```
        </execution>
    </executions>
</plugin>
```

#### 2.2.4 Tests

Pour exécuter les tests du projet à chaque goal test ou à chaque compilation , on utilise le plugin *maven-surefire-plugin*.

### 2.3 Pourquoi l'utiliser ?

Maven sert à compiler le code et s'occuper de la phase de production. Il s'occupe pour nous de la gestion des dépendances. On a juste à décrire les dépendances du projet. C'est très pratique car on ne stocke pas toujours les bibliothèques tierces sur les serveurs de collaboration, simplement une référence. Si un collègue récupère le projet et qu'il n'a pas les bibliothèques, il ne peut pas le compiler, d'où Maven.

Pour les gros projets séparés en plusieurs modules, Maven permet d'avoir une hiérarchie de projet, de la déclarer et d'avoir des choses en commun. Il permet de fixer les versions des bibliothèques utilisées.

## 3 SonarQube

### 3.1 Description

SonarQube est une plateforme open-source utilisée en QA. Cet outil sert à vérifier (analyse statique) les règles de mise en forme, la complexité d'un programme, les commentaires, les bugs potentielles. De nombreux modules existent pour ajouter la gestion d'autres langages. Les métriques utilisées sont :

- Le nombre de lignes de code (ainsi que le nombre de fichiers, répertoires, de fonctions, de classes, etc)
- Le pourcentage de code dupliqué
- La complexité (mesurée en nombres de fonctions/classes/fichiers)
- Les problèmes de code (des issues sont ouvertes, classées par ordre d'importance (Bloquant, critique, majeur, mineur, informatif))

Les problèmes détectés par l'outil sont paramétrables à l'aide d'un jeu de règles. Par défaut pour notre projet, nous avons utilisé le profil Sonar Way (Java) qui contient 192 règles par défaut. Ces règles sont activables/désactivables. De plus des règles peuvent être créées manuellement (nous n'avons pas testé cette fonctionnalité). Généralement les règles contiennent une description précise.

### 3.2 Utilisation

L'utilisation de cet outil fut assez basique. Elle consistait de lancer *Sonar* et d'exécuter :

```
mvn sonar:sonar
```

Ce qui permettait d'obtenir le projet sur l'interface de *Sonar*. Ensuite, nous avons pu parcourir l'interface et paramétrer les profils pour détecter plus d'erreurs ou enlever les erreurs reconnues.

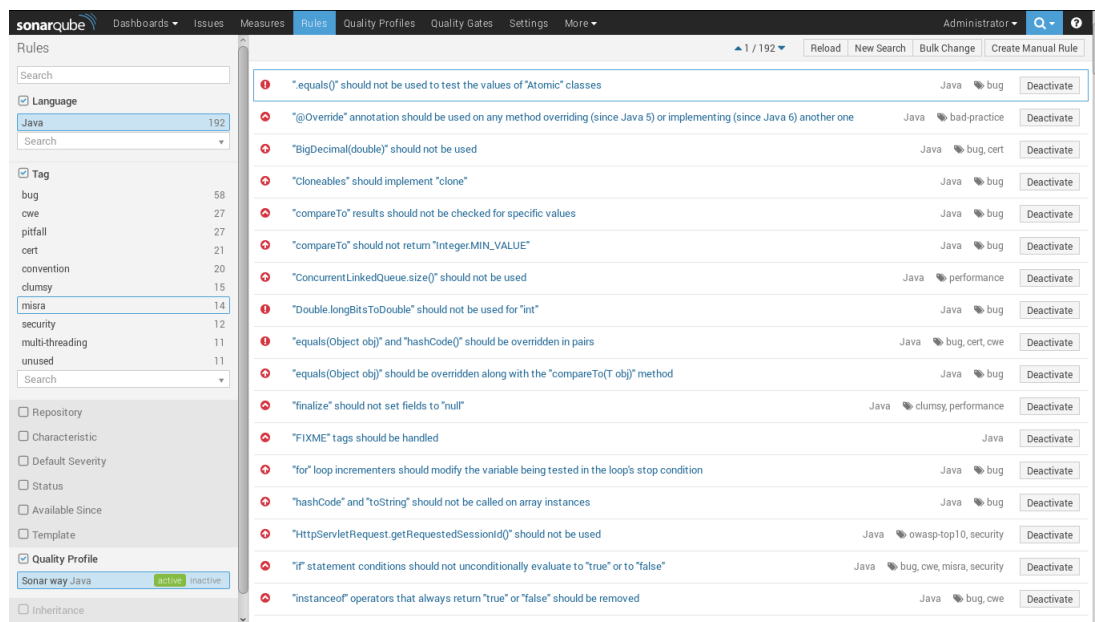


FIGURE 1 – Le paramétrage des règles du profil *Sonar Way (Java)*

### 3.3 Pourquoi l'utiliser ?

Les outils d'analyse statiques sont très utilisés dans les projets, car un développeur ne travaille généralement pas seul. Du coup ce type d'outil est utile pour les raisons suivantes :

- Un code propre est généralement un code facile à lire, à comprendre. ce qui facilite le travail aux personnes travaillant avec le développeur.
- Une erreur dans le code est rapidement arrivée. Une analyse statique permet d'éviter des bugs faciles à trouver comme la double libération d'une zone mémoire, l'utilisation d'un pointeur non initialisé, etc.

Il existe d'autres outils d'analyse statiques qui n'ont pas été abordés dans ce TP, notamment l'analyseur statique de *clang*.

## 4 Jenkins

### 4.1 Description

*Jenkins* est un serveur d'automatisation de compilation. Cet outil permet d'automatiser des compilations, des tests et le déploiement d'une application. De plus il est personnalisable avec des centaines de plugins.

### 4.2 Utilisation

Ce TP nous a permis d'automatiser les builds d'un git distant. Dans notre cas, il s'agit du dépôt disponible à l'adresse <https://github.com/AmarOk1412/MinisTPS8>. Ce dépôt contenant plusieurs matières, nous avons créé un projet de type Multi-configurations et dans la partie *Build*, nous avons ajouté un appel à un script shell contenant :

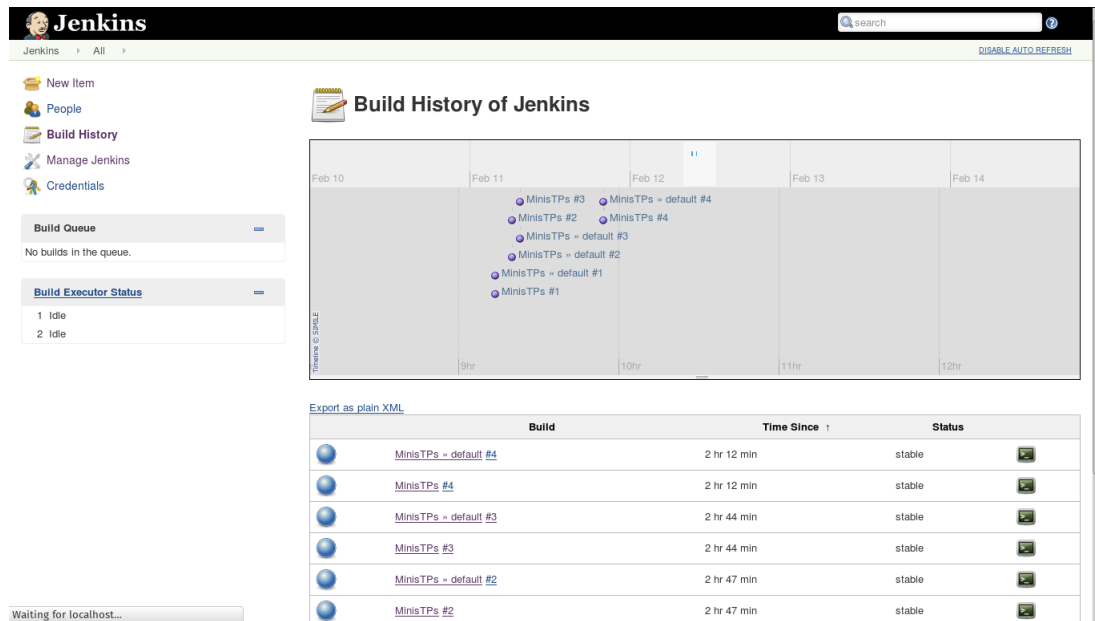


FIGURE 2 – L'historique des builds de *Jenkins*

```
sh -c 'cd AL/TP1 && mvn clean package install'
```

Bien sûr, ce script peut-être complété pour compiler tous les TP's et effectuer tous les tests. Nous avons ensuite paramétré le *pom.xml* de *Maven* afin d'avoir un package propre en sortie de compilation. L'archive générée par la compilation automatisée est disponible dans le workspace (tout comme la sortie de la console, ce qui peut-être utile en cas de bug).

### 4.3 Pourquoi l'utiliser ?

Les outils d'intégration continue sont utiles car ils permettent de :

- Savoir quel commit a apporté un bug (automatisation des tests) et permet de savoir si quelque chose c'est mal déroulé sur une plateforme ce qui permet de prévenir les développeurs (mail) et de régler le problème rapidement.
- Pouvoir automatiser le déploiement de l'application sans suivi (et paramétrer des cycles de déploiement (par exemple tous les jours pour une version alpha, toutes les semaines pour une bêta et tous les mois pour une version stable)).

D'autres outils existent comme *Travis* qui est aussi utilisé sur le dépôt (<https://travis-ci.org/AmarOk1412/MinisTPS8>)

The screenshot displays the Travis CI web interface. At the top, the navigation bar includes the Travis CI logo, links for Blog, Status, and Help, and the user profile 'AmarOk'. A search bar for repositories is located on the left. Below it, a 'My Repositories' section lists several repositories with their build status (all green checkmarks), commit hashes, and completion times. The main area shows the details for the repository 'AmarOk1412 / MinisTPS8', indicating a 'build passing' status. It lists the current branch as 'master' and shows the commit 'eb0cb9c'. The build log is visible, showing the cloning of the repository and the execution of build scripts. The log includes system information and the output of Java and Maven commands.

Travis CI

Blog Status Help

AmarOk

Search all repositories

My Repositories +

- ✓ AmarOk1412/MinisTPS8 # 9  
Duration: 1 min 30 sec  
Finished: about an hour ago
- ✓ zestedesavoir/zds-site # 5079  
Duration: 16 min 28 sec  
Finished: about 15 hours ago
- ✓ RORI-Chatterbot/RORI # 16  
Duration: 1 min 30 sec  
Finished: about a month ago
- ✓ RORI-Chatterbot/EntryServer # 18  
Duration: 51 sec  
Finished: about a month ago
- ✓ RORI-Chatterbot/DesktopClient # 13  
Duration: 1 min 10 sec  
Finished: about a month ago

AmarOk1412 / MinisTPS8 build passing

Current Branches Build History Pull Requests More options

✓ master begin report  
Commit eb0cb9c  
Compare 3024f5d..eb0cb9c  
AmarOk authored and committed  
#9 passed  
Elapsed time 1 min 30 sec  
about an hour ago

Remove log Raw log

```
1 Using worker: worker-linux-docker-c9510230.prod.travis-ci.org:travis-linux-6
2
3 Build system information system_info
67
68 $ git clone --depth=50 --branch=master https://github.com/AmarOk1412/MinisTPS8.git AmarOk1412/MinisTPS8 git_checkout 1/310
69
70 This job is running on container-based infrastructure, which does not allow use of 'sudo', setuid and setgid executables.
71 If you require sudo, add 'sudo: required' to your .travis.yml
72 See https://docs.travis-ci.com/user/workers/container-based-infrastructure/ for details.
73 $ java -Xmx32m -version
74 java version "1.7.0_76"
75 Java(TM) SE Runtime Environment (build 1.7.0_76-b13)
76 Java HotSpot(TM) 64-Bit Server VM (build 24.76-b04, mixed mode)
77 $ javac -J-Xmx32m -version
78 javac 1.7.0_76
79 $ sh -c 'cd AL/TP1 66 mvn compile 66 mvn test' 23/560
80 [INFO] Scanning for projects...
```

FIGURE 3 – L’outil *Travis-CI* sur le dépôt du projet