



# Maven

---

David THIBAU – 2023

[david.thibau@gmail.com](mailto:david.thibau@gmail.com)



# Agenda

---

## **Introduction**

- Outils de build
- Principes Maven

## **Configuration**

- Gestion des dépendances
- Adaptation du cycle de build
- Profils de build
- Packaging war
- Projets multi-modules

## **Dépôts et releases**

- Les dépôts d'artefacts
- Le processus de release

## **Tests**

- Tests unitaires/Tests d'intégration
- Couverture des tests

## **Analyse statique**

- Checkstyle
- Intégration Sonarqube

## **Intégration Jenkins**

- Configuration Jenkins / vs Maven wrapper / docker



# Introduction

---

**Outils de build**  
Principes Maven



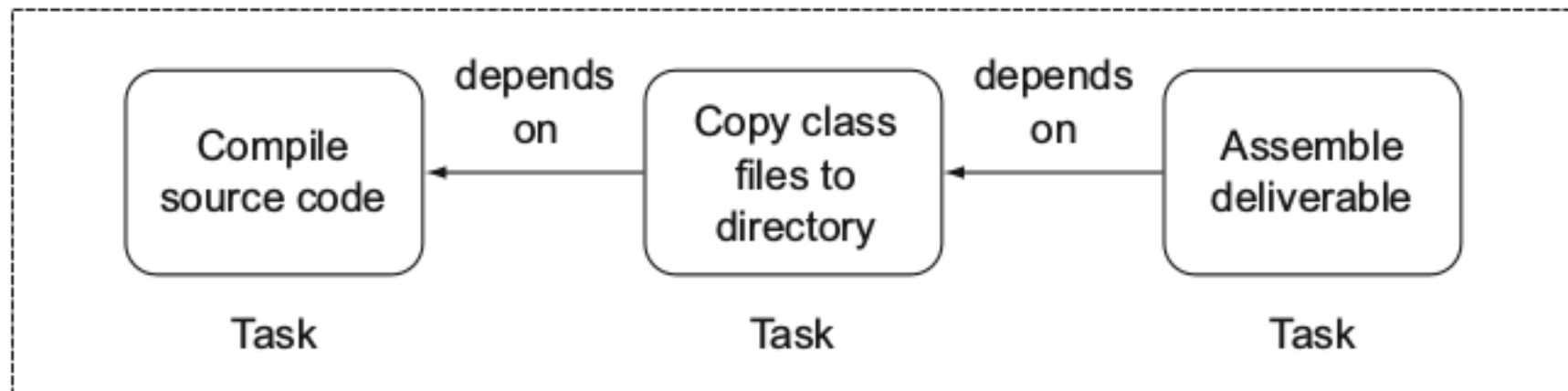
# Pré-requis d'un build

- ♦ **Proscrire les interventions manuelles** sujettes à erreur et chronophage
- ♦ Créer des builds **reproductibles** : Pour tout le monde qui exécute le build
- ♦ **Portable** : Ne doit pas nécessiter un OS ou un IDE particulier, il doit être exécutable en ligne de commande
- ♦ **Sûr** : Confiance dans son exécution

# Graphe de build

Un build est une séquence ordonnée de tâches unitaires.

Les tâches ont des dépendances entre elles qui peuvent être modélisées via un **graphe acyclique dirigé** :





# Composants d'un outil de build

---

**Le fichier de build** : Contient la configuration requises pour le build, les dépendances externes, les instructions pour exécuter un objectif sous forme de tâches inter-dépendantes

**Une tâche** prend une entrée effectue des traitements et produit une sortie. Une tâche dépendante prend comme entrée la sortie d'une autre tâche

**Moteur de build** : Le moteur traite le fichier de build et construit sa représentation interne. Des outils permettent d'accéder à ce modèle via une API

**Gestionnaire de dépendances** : Traite les déclarations de dépendances et les résout par rapport à un dépôt d'artefact contenant des méta-données permettant de trouver les dépendances transitive



# Monde Java

**Apache Ant** : L'ancêtre. Pas de gestionnaire de dépendances. Plein de tâches prédéfinies. Facilité d'extension. Pas de convention

**Maven** : L'actuel. Gestionnaire de dépendances. Convention plutôt que configuration. Extension par mécanisme de plugin. Verbeux car XML

**Gradle** : Gestionnaire de dépendances. Allie les qualités de Maven et des capacités de codage du build. Concis



# Introduction

---

Outils de build  
**Principes Maven**



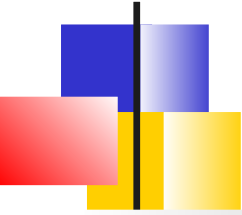


# Qu'est ce que Maven

---

- Définition officielle (Apache) : Outil de gestion de projet qui fournit :
  - un **modèle** de projet
  - un ensemble de **standards**
  - un **cycle de vie (ou de construction)** pour un projet Java
  - un système de **gestion de dépendances**
  - de la logique pour exécuter des **objectifs** via des **plugins** à des **phases** bien précises du cycle de vie/construction du projet
- Avec Maven, on décrit son projet en utilisant un modèle bien défini. (*pom.xml*)
- Ensuite, Maven peut appliquer de la logique transverse grâce à un ensemble de plugins partagés par la communauté Java (pouvant être adaptés)

# Convention plutôt que Configuration

- 
- Concept permettant d'alléger la configuration en respectant des conventions
  - Par exemple, Maven présuppose une organisation du projet dont le but est de produire un jar:
    - **`${basedir}/src/main/java`** contient le code source
    - **`${basedir}/src/main/resources`** contient les ressources
    - **`${basedir}/src/test/java`** contient les classes de tests
    - **`${basedir}/src/test/resources`** contient les ressources de tests
    - **`${basedir}/target/classes`** les classes compilées
    - **`${basedir}/target`** contient le jar à distribuer
  - Les plugins cœur de Maven se basent sur cette structure pour compiler, packager, générer des sites webs de documentation, etc..



# Avant/Après

---

- Avant Maven, une personne était dédiée à l'élaboration des scripts de build
- Avec Maven, il suffit de :
  - Check-out du source
  - Exécuter ***mvn install***



# Modèle conceptuel d'un projet

---

- Maven maintient un modèle de projet. Les développeurs doivent le renseigner :
  - Coordonnées permettant d'identifier le projet
  - Informations projet : Mode de licence, Développeurs et contributeurs
  - Dépendances vis-à-vis d'autres projets
  - Personnalisation du build par défaut
  - Profils de build
  - ...
- Tout ceci est décrit dans un fichier XML unique : ***pom.xml***



# Exemple *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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.ch03</groupId>
  <artifactId>simple</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>simple</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```



# Coordonnées Maven

---

- **groupid** : Le groupe, la société. La convention de nommage stipule qu'il commence par le nom de domaine de l'organisation qui a créé le projet (*com.plb* ou *org.apache*, ..)
- **artifactId** ; Un identifiant unique à l'intérieur de *groupid* qui identifie un projet unique
- **version** : Une release spécifique d'un projet. Les projets en cours de développement utilise un identifiant spécial : SNAPSHOT.
- Le format de **packaging** conditionne le cycle de construction (Exemples : jar, war, pom) il ne fait pas partie de l'identifiant projet.

Les coordonnées *groupid:artifactId:version* rendent le projet unique.

# Propriétés

- Un POM peut inclure des propriétés :  
`${project.groupId}-${project.artifactId}`
- Ces propriétés sont évaluées à l'exécution
- Maven fournit des propriétés implicites :
  - **env** : Variables d'environnement système
  - **project** : Le projet
  - **settings** : Accès à la configuration de *setting.xml*
  - Propriétés **Java**
- On peut en définir via :  
`<properties> <foo>bar</foo> /properties>`



# Commandes Maven

---

Les commandes Maven sont exécutées via :

***mvn <plugins:goals> <phase>***

La commande peut donc spécifier :

- L'exécution d'un **objectif** d'un plugin particulier
- L'exécution d'une **phase** ... provoquant l'exécution de plusieurs objectifs de différents plugins





# Options Maven

---

**-h** : Permet d'afficher toutes les options disponibles

Les plus importantes étant :

**-D** : Permet de fixer une propriété de la JVM

`mvn -Dsonar.token=$TOKEN`

**-f** : Indique un autre fichier que *pom.xml*

**-l** : Permet d'indiquer un fichier de log

**-o** : Offline, permet de tester si on peut builder sans réseau

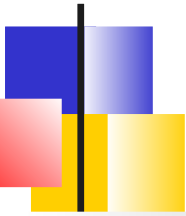
**-P** : Profils de build

**-q** : Quiet, n'affiche que les erreurs

**-U** : Force la vérification des releases manquantes et réactualise les SNAPSHOTs

**-s** : Permet d'indiquer un autre fichier *settings.xml*

**-X** : Debug



# Variables d'environnement

---

Maven est également sensible à certaines variables d'environnement.

**JAVA\_HOME** est utilisé pour localiser le compilateur

**MAVEN\_OPTS** permet de positionner des options de démarrage de la JVM

Ex : -Xms256m -Xmx512m

Les options de la JVM peuvent également être positionnées dans le fichier *.mvn/maven.config*

**MAVEN\_ARGS** les options de Maven

Les options peuvent également être positionnées dans *.mvn/jvm.config*



# Plugins et objectifs

---

- Un **plugin** Maven est composé d'un ensemble de tâches atomiques : les **objectifs** (goals)
- Exemples :
  - le plugin **Jar** contient des objectifs pour créer des fichiers jars
  - le plugin **Compiler** contient des objectifs pour compiler le code source
  - le plugin **Surefire** contient des objectifs pour exécuter les tests unitaires et générer des rapports de tests.
  - D'autres plugins plus spécialisés comme le plugin *Sonar*, le plugin *Jruby*, ...

Maven fournit également la possibilité de définir ses propres plugins. Un plugin spécifique peut-être écrit en Java, Groovy, ...

Les plugins sont bien documentés :

Ex: <https://maven.apache.org/surefire/maven-surefire-plugin/>



# Maven Plugins et objectifs

---

- Un objectif est une tâche spécifique qui peut être exécutée en *standalone* ou comme faisant partie d'un build plus large.
- La syntaxe d'exécution est la suivante :  
**mvn <plugin>:<objectif>**
- Un **objectif est l'unité de travail** dans Maven  
Exemples : l'objectif de compilation dans le compilateur plugin
- Les objectifs peuvent être configurés via un certain nombre de propriétés.  
Par exemple : La version du JDK est un paramètre de l'objectif de compilation.



# Cycle de vie Maven

---

- Le cycle de construction est une **séquence ordonnée de phases** impliquées dans la construction d'un projet
- Pour chaque packaging, Maven supporte différents cycles de vie. Le cycle de construction par défaut est le plus souvent utilisé, pour un packaging jar il commence avec une phase validant l'intégrité du projet et termine avec une phase déployant une release du projet dans un dépôt
- Les phases du cycle de vie sont fixes mais intentionnellement « vagues » : *validation*, *test* ou *déploiement*  
Elles peuvent signifier différentes choses en fonction des projets.

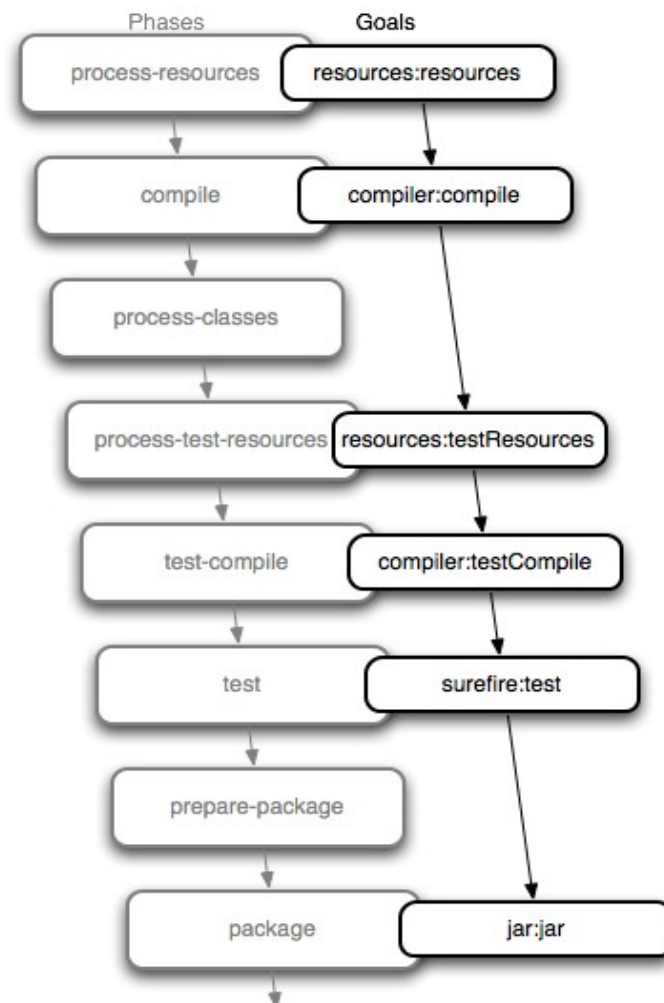


# Phase et objectifs

---

- Les objectifs d'un plugin peuvent être **attachés** à une phase du cycle de construction. Lorsque Maven atteint une phase, il exécute tous les objectifs qui lui sont attachés.
- Par exemple, lors de l'exécution de **mvn install** *install* représente la phase et son exécution provoquera l'exécution de plusieurs objectifs
- L'exécution d'une phase exécute également toutes les phases précédentes dans l'ordre défini par le cycle de construction.

# Cycle de vie par défaut



Note: There are more phases than shown above, this is a partial list

# Objectifs par défaut pour un packaging jar

- ***process-resources / resources:resources*** : Copie tous les fichiers ressources de *src/main/resources* dans le répertoire *target/classes*
- ***compile / compiler:compile*** : compile tout le code source présent dans *src/main/java* vers *target/classes*.
- ***process-test-resources / resources:testResources*** : copie toutes les ressources du répertoire *src/test/resources* vers le répertoire de sortie pour les tests
- ***test-compile / compiler:testCompile*** : compile les cas de test présent dans *src/test/java* vers le répertoire de sortie des tests
- ***test / surefire:test*** : exécute tous les tests et crée les fichiers résultats, termine le build en cas d'échec
- ***package / jar:jar*** : crée un fichier jar à partir du répertoire de sortie
- ***install / install:install*** : Installe l'artefact dans le dépôt local. Par défaut *~/.m2*
- ***deploy / deploy:deploy*** : Déploie l'artefact vers le dépôt distant. Par défaut MavenCentral





# Exemple

---

Exemple : Lancement des objectifs liés à la phase test suivi de l'appel de l'objectif *sonar* du plugin *sonar*

```
$ mvn test sonar:sonar
```

# POM parents et POM effectif

- Il peut exister des relations d'héritages entre les POMs :
  - POM parent utilisé pour mutualiser des configurations communes
  - POM d'un projet multi-modules
- Tous les POM héritent du POM racine dénommé Super POM
- Le fichier *pom.xml* effectif utilisé pour l'exécution est donc en fait une combinaison du fichier *pom* du projet, de tous les fichiers POMs des projets parents, du fichier POM racine de Maven et des configurations spécifiques de l'utilisateur
- Afin de consulter le fichier réellement utilisé par Maven, il suffit d'utiliser le plugin d'aide :

**\$ mvn help:effective-pom**



# POM Parent

---

La référence vers le POM parent est indiquée via la balise **<parent>**

```
<parent>
  <groupId>org.formation.maven</groupId>
  <artifactId>parent</artifactId>
  <version>1.0</version>
</parent>
```

Maven recherche alors le *pom.xml* correspondant dans le dépôt local.

On peut également indiquer l'élément **<relativePath/>** si le pom parent n'a pas été installé dans le dépôt

```
<parent>
  <groupId>org.formation.maven</groupId>
  <artifactId>parent</artifactId>
  <version>1.0</version>
  <relativePath>../parent-project/pom.xml</relativePath>
</parent>
```



# Fichier settings.xml

---

Un autre fichier configure le comportement Maven :  
***settings.xml***

Il se place généralement dans `${HOME}/.m2` et on y trouve principalement des personnalisations de l'utilisateur :

- Identifiants de l'utilisateur pour se connecter à des dépôts
- Définitions des serveurs miroirs
- Personnalisation de l'emplacement du dépôt local
- Définition de groupes de plugins, non standard

Il peut également se trouver dans le répertoire projet ou être indiqué via l'option `-s` de la commande *mvn*



# Éléments de *settings.xml*

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"  
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0  
      https://maven.apache.org/xsd/settings-1.0.0.xsd">
```

```
  <localRepository/>  
  <interactiveMode/>  
  <offline/>  
  <pluginGroups/>  
  <servers/>  
  <mirrors/>  
  <proxies/>  
  <profiles/>  
  <activeProfiles/>
```

```
</settings>
```



# Exemple configuration Proxy

---

```
<settings>
.
.
<proxies>
  <proxy>
    <id>example-proxy</id>
    <active>true</active>
    <protocol>http</protocol>
    <host>proxy.example.com</host>
    <port>8080</port>
    <username>proxyuser</username>
    <password>somepassword</password>
    <nonProxyHosts>www.google.com|*.example.com</nonProxyHosts>
  </proxy>
</proxies>
.
.
</settings>
```



# Configuration

---

## **Gestion des dépendances**

Adaptation du cycle de build

Profils de build

Packaging war

Projets multi-modules



# Gestion des dépendances

- La possibilité de localiser un artefact à partir de ces coordonnées permet d'indiquer les dépendances dans le projet POM.
- Le fait que le POM associé à tout artefact soit stocké dans le dépôt permet à Maven de résoudre les dépendances transitives. Le support pour les dépendances transitives est une de fonctionnalités les plus puissantes de Maven.
- Il est possible de définir différents **périmètres** de dépendances. Par exemple, la dépendance *junit:junit:jar:3.8.1* n'existe que dans le périmètre de test.
  - Le périmètre fourni donne des indications au plugins (inclure la dépendance dans le classpath, le packaging, ...).



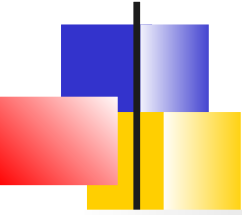


# Dépendances transitives

---

- Maven permet de s'affranchir des dépendances transitives. Si par exemple, votre projet dépend d'une librairie B qui dépend elle-même d'une librairie C, il n'est pas nécessaire d'indiquer la dépendance sur C
- Lorsqu'une dépendance est téléchargée vers le dépôt local, Maven récupère également le *pom* de la dépendance et est donc capable d'aller chercher les autres artefacts dépendants. Ces dépendances sont alors ajoutées aux dépendances du projet courant.
- Si ce comportement n'est pas désirable, il est possible d'exclure certaines dépendances transitives

# Ajout de dépendances

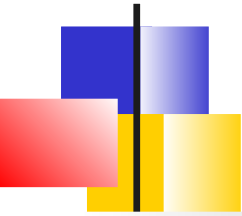


---

L'ajout de dépendance est réalisé en ajoutant une balise **<dependency>** sous la balise **<dependencies>** et en indiquant les coordonnées Maven

```
<dependencies>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.14</version>
  </dependency>
</dependencies>
```

# Périmètres

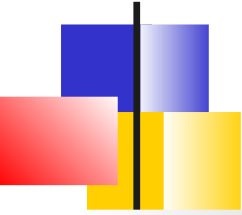


La balise `<dependency>` peut spécifier un périmètre via la balise ***scope***.

5 périmètres sont disponibles :

- ***compile*** : Le périmètre par défaut, les dépendances sont tout le temps dans le classpath et sont packagées
- ***provided*** : Dépendance fournie par le container. Présent dans le classpath de compilation mais pas packagée
- ***runtime*** : Présent à l'exécution et lors des tests mais pas à la compilation Ex : Une implémentation JDBC
- ***test*** : Disponible seulement pour les tests
- ***system*** : Fournie par le système (pas recherché dans un repository)

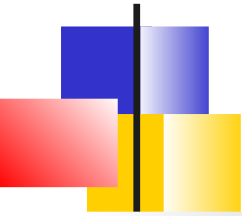
# Example



---

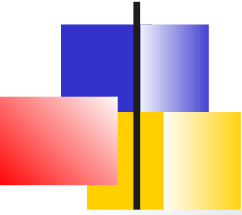
```
<dependency>  
  <groupId>org.apache.commons</groupId>  
  <artifactId>commons-io</artifactId>  
  <version>1.3.2</version>  
  <scope>test</scope>  
</dependency>
```

# Exclusion de dépendances



```
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate</artifactId>
    <version>3.2.5.ga</version>
    <exclusions>
      <exclusion>
        <groupId>javax.transaction</groupId>
        <artifactId>jta</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
  <dependency>
    <groupId>org.apache.geronimo.specs</groupId>
    <artifactId>geronimo-jta_1.1_spec</artifactId>
    <version>1.1</version>
  </dependency>
</dependencies>
```

# Où trouver les dépendances ?

- 
- 
- Le site <http://www.mvnrepository.com> permet de retrouver les *groupid* et les *artifactId* nécessaires pour indiquer les dépendances
    - Il fournit une interface de recherche vers le dépôt public Maven.
    - Lorsque l'on recherche un artefact, il liste l'*artifactId* et toutes ses versions connues.
    - En cliquant sur une version, on peut récupérer l'élément `<dependency>` à inclure dans son POM



# Version du projet

---

- La version d'un projet Maven permet de grouper et ordonner les différentes releases.
- Elle est constituée de 4 parties :

`<major version>.<minor version>.<incremental version>[-<qualifier>]`

Exemple : *1.3.5-beta-01*

- En respectant ce format, on permet à Maven de déterminer quelle version est plus récente
- SNAPSHOT indique qu'une version est en cours de développement et nécessite de récupérer le code souvent

# Gestion des versions des dépendances



---

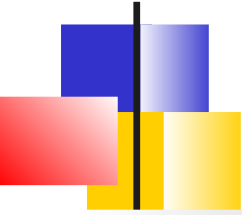
Pour éviter de disperser dans tous les projets de l'entreprise les mêmes dépendances classiques ... et d'avoir des soucis de mis à jour lors d'une montée de version

Il est possible de définir via la balise ***<dependencyManagement>*** les versions des dépendances


Cela est généralement effectué dans un POM parent



# <dependencyManagement>

- 
- L'élément <dependencyManagement> permet de centraliser les n° de versions des librairie utilisée.
  - Un projet enfant n'a alors plus besoin d'indiquer un n° de version .

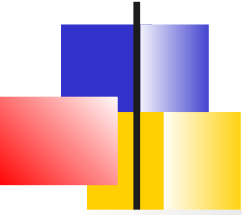
# POM parent



---

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.2</version>
    </dependency>
    ...
  </dependencies>
</dependencyManagement>
```

# POM enfant



---

```
<parent>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>a-parent</artifactId>
  <version>1.0.0</version>
</parent>
<artifactId>project-a</artifactId>
...
<dependencies>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
  </dependency>
</dependencies>
```



# Importation de dépendances BOM

---

Un BOM (Bills Of Material) est un POM spécifique utilisé uniquement pour contrôler les versions des dépendances d'un projet

Il fournit ainsi un point central de configuration.

Si un projet enfant importe le BOM, il n'a plus besoin de spécifier les versions de ces dépendances.



# Example

---

```
<project ...>
  <modelVersion>4.0.0</modelVersion>
  <groupId>baeldung</groupId>
  <artifactId>Baeldung-BOM</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>pom</packaging>
  <description>parent pom</description>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>test</groupId>
        <artifactId>a</artifactId>
        <version>1.2</version>
      </dependency>
      <dependency>
        <groupId>test</groupId>
        <artifactId>b</artifactId>
        <version>1.0</version>
        <scope>compile</scope>
      </dependency>
      <dependency>
        <groupId>test</groupId>
        <artifactId>c</artifactId>
        <version>1.0</version>
        <scope>compile</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
</project>
```



# 2 utilisations

---

## Héritage simple du BOM ou mieux importation

```
<project ...>
  ...
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>baeldung</groupId>
        <artifactId>Baeldung-BOM</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
</project>
```



# Gestion des versions des plugins


---

Le même mécanisme est utilisé pour fixer la version des plugins utilisés

Dans un POM parent, la balise **<pluginManagement>** permet de fixer les versions des plugins utilisables ainsi que des propriétés de configuration

Les POMs enfants utilisent alors les mêmes versions de plugins

# Exemple *pluginManagement*



---

```
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>hibernate3-maven-plugin</artifactId>
        <version>2.1</version>
      </plugin>
    </plugins>
  </pluginManagement>
  ...
```





# Configuration

---

Gestion des dépendances

**Adaptation du cycle de build**

Profiles de build

Packaging war

Projets multi-modules



# Adaptation

---

2 mécanismes sont possibles pour adapter le cycle de construction Maven aux spécificités d'un projet

- **Configurer** les plugins utilisés en surchargeant la configuration par défaut.

*Voir la doc du plugin pour trouver les options de configuration*

- **Attacher** des objectifs de nouveaux plugins à des phases du build



# Exemple *compiler:compile*

---

Par défaut, cet objectif compile tous les sources de *src/main/java* vers *target/classes*

Le plugin *Compile* utilise alors *javac* et suppose que les sources sont en Java 1.3 et vise une JVM 1.1 !


Pour changer ce comportement, il faut spécifier les versions visées dans le *pom.xml*



# Example

---

```
<project> ...  
<build> ...  
<plugins>  
  <plugin>  
    <artifactId>maven-compiler-plugin</artifactId>  
    <configuration>  
      <source>1.5</source>  
      <target>1.5</target>  
    </configuration>  
  </plugin>  
</plugins> ...  
</build> ...  
</project>
```

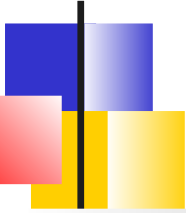


# Association objectif/phase.

## L'exemple du plugin Assembly

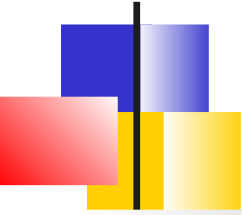
- Le plugin **Assembly** permet de créer des distributions du projets sous d'autres formats que le types d'archives standard (*.jar*, *.war*, *.ear*)
  - Il est ainsi possible de générer un fat jar (exécutable du projet)
- Par défaut, ce plugin n'est pas utilisé par le cycle de construction

# Attacher assembly avec package



On peut configurer Maven afin qu'il exécute l'assemblage lors du cycle de vie par défaut en attachant l'objectif d'assemblage à la phase de packaging

# Example



---

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <configuration>
        <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>
```



# Configuration

---

Gestion des dépendances  
Adaptation du cycle de build

**Profils de build**

Packaging war  
Projets multi-modules





# Profils

---

Les **profils** permettent de personnaliser un build en fonction d'un environnement (test, production, ...)

Maven permet de définir autant de profils que désirés qui **surchargent** la configuration par défaut

Les profils sont **activés** manuellement, en fonction de l'environnement ou du système d'exploitation

Les profils, configurés dans le *pom.xml*, sont associés à des **identifiants**



# Example

---

```
<profiles>
  <profile>
    <id>production</id>
    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <configuration>
            <debug>>false</debug>
            <optimize>>true</optimize>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```



# `<profiles>`

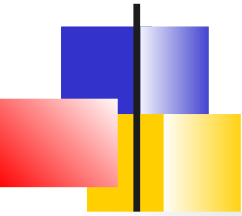
---

La balise **`<profiles>`** liste les profils du projet, elle se trouve en fin du fichier *pom.xml*

Chaque profile a un élément **`<id>`**, l'activation du profile en ligne de commande s'effectue via **`-P<id>`**

Un élément **`<profile>`** peut contenir les balises qui se trouve habituellement sous `<project>`

# Activation automatique des profils



Il est possible d'activer automatiquement un profil en fonction de :

- ✓ La version du JDK
- ✓ Les propriétés de l'OS
- ✓ Les propriétés Maven (ou l'absence de propriété)
- ✓ La présence d'un fichier

# Exemple



```
<profile>
  <id>dev</id>
  <activation>
    <activeByDefault>false</activeByDefault>
    <jdk>1.5</jdk>
    <os>
      <name>Windows XP</name>
      <family>Windows</family>
      <arch>x86</arch>
      <version>5.1.2600</version>
    </os>
    <property><name>mavenVersion</name><value>2.0.5</value></property>
    <property><name>!environnement.type</name></property>
    <file>
      <exists>file2.properties</exists>
      <missing>file1.properties</missing>
    </file>
  </activation>
  ...
</profile>
```

---



# Configuration

---

Gestion des dépendances  
Adaptation du cycle de build  
Profils de build  
**Packaging war**  
Projets multi-modules



# Introduction

---

Maven est également adapté pour produire des packages JavaEE

En précisant la balise `packaging` à `war`, `ejb` ou `ear`, il adapte le cycle de build pour ces formats

- En prenant en compte les descripteurs de déploiement standard :  
*META-INF/ejb-jar.xml*,  
*WEB-INF/web.xml*
- En packagant d'autre répertoire ressources  
*src/main/webapp*



# Structure d'un projet war

---

```
| -- pom.xml
|-- src
|   |-- main
|       |-- java
|           |-- com
|               |-- example
|                   |-- projects
|                       |-- SampleAction.java
|       |-- resources
|           |-- images
|               |-- sampleimage.jpg
|   |-- webapp
|       |-- WEB-INF
|           |-- web.xml
|       |-- index.jsp
|       |-- jsp
|           |-- webservice.jsp
```





# war généré

---

documentedproject-1.0-SNAPSHOT.war

```
|-- META-INF
|   |-- MANIFEST.MF
|   |-- maven
|       |-- com.example.projects
|           |-- documentedproject
|               |-- pom.properties
|               |-- pom.xml
|-- WEB-INF
|   |-- classes
|       |-- com
|           |-- example
|               |-- projects
|                   |-- SampleAction.class
|       |-- images
|           |-- sampleimage.jpg
|   |-- web.xml
|-- index.jsp
|-- jsp
    |-- webservice.jsp
```



# Ressources web

---

Il est possible de définir des alternatives pour les répertoires contenant les ressources web (css, images, etc.)

```
<plugin>
  <artifactId>maven-war-plugin</artifactId>
  <version>3.3.2</version>
  <configuration>
    <webResources>
      <resource>
        <directory>resource2</directory>
        <includes>
          <include>/**/*.jpg</include>
        </includes>
      </resource>
    </webResources>
  </configuration>
</plugin>
```



# Plugin Jetty

---

Le plugin Jetty permet de rapidement tester l'application sans la packager

```
<plugin>
  <groupId>org.mortbay.jetty</groupId>
  <artifactId>maven-jetty-plugin</artifactId>
  <version>6.1.10</version>
  <configuration>
    <scanIntervalSeconds>10</scanIntervalSeconds>
    <connectors>
      <connector implementation="org.mortbay.jetty.nio.SelectChannelConnector">
        <port>8080</port>
        <maxIdleTime>60000</maxIdleTime>
      </connector>
    </connectors>
  </configuration>
</plugin>
```



# Configuration

---

Gestion des dépendances  
Adaptation du cycle de build  
Profils de build  
Packaging war  
**Projets multi-modules**

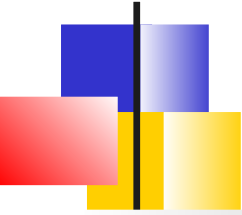
# Projets multi-modules



---

- Un projet multi-modules est défini par un POM parent qui référence plusieurs sous-modules :
- La définition s'effectue via les balises **<modules>** et **<module>**


# POM parent



---

```
<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/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.ch07</groupId>
  <artifactId>simple-parent</artifactId>
  <packaging>pom</packaging>
  <version>1.0</version>
  <name>Chapter 7 Simple Parent Project</name>
  <modules>
    <module>simple-model</module>
    <module>simple-persist</module>
    <module>simple-webapp</module>
  </modules>
</project>
```

# POM d'un projet multi-modules

- 
- Le POM parent définit les coordonnées Maven
  - Il ne crée pas de JAR ni de WAR, il consiste seulement à référencer les autres projets Maven, le packaging a alors la valeur ***pom***.
  - Dans l'élément **<modules>**, les sous-modules correspondant à des sous-répertoires sont listées
  - Dans chaque sous-répertoire, Maven pourra trouver les fichiers *pom.xml* et inclura ces sous-modules dans le *build*
  - Enfin, le POM parent peut contenir des configurations qui seront héritées par les sous-modules

# POM des sous-modules

- Le POM des sous-modules référence le parent via ses coordonnées
- Les sous-modules ont souvent des dépendances vers d'autres sous-modules du projet

```
<parent>
```

```
  <groupId>org.sonatype.mavenbook.ch07</groupId>
```

```
  <artifactId>simple-parent</artifactId>
```

```
  <version>1.0</version>
```

```
</parent>
```

```
...
```

```
<dependencies>
```

```
  <dependency>
```

```
    <groupId>org.sonatype.mavenbook.ch07</groupId>
```

```
    <artifactId>simple-model</artifactId>
```

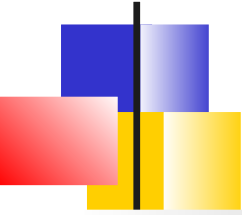
```
    <version>1.0</version>
```

```
  </dependency>
```

```
...
```



# Exécution de Maven

- 
- 
- Lors de l'exécution du build, Maven commence par charger le POM parent et localise tous les POMs des sous-modules
  - Ensuite, les dépendances des sous-modules sont analysées et déterminent l'ordre de compilation et d'installation
  - La commande est naturellement exécutée sur le projet parent



# Dépôts et Releasing

---

## **Les dépôts d'artefacts** Processus de Release



# Dépôts

---

Avec Maven, il existe donc différents types de dépôts utilisés pour récupérer les dépendances et pour y installer des artefacts.

- Le dépôt **local**
- Les dépôts **publics** fournis par Maven, JBoss, Sun, ...
- Les dépôts **miroirs** généralement configurés dans le fichier *settings.xml*
- Les dépôts **privés**, propres à une organisation ou entreprise



# Dépôt privé

---

L'utilisation de Maven dans le contexte d'une entreprise nécessite souvent la mise en place d'un dépôt interne à l'entreprise :

- sécurité,
- bande passante
- et surtout possibilité d'y publier les artefacts produits par la société.

Les artefacts produits peuvent être récupérés via *http*, *scp*, *ftp* ou *cp*

Les outils spécialisés sont *Nexus*, *Artifactory*, *Archiva*



# Utilisation d'un dépôt interne

---

Il suffit d'indiquer :

- une balise **<repository>** dans le fichier *pom.xml* qui référence un id de serveur défini *dans* une balise **<server>** dans du fichier *settings.xml*
- Dans *settings.xml*, on trouve les créidentiels d'accès au dépôt interne



# Balise repository dans *pom.xml*

---

```
<project>
  ...
  <repositories>
    <repository>
      <id>my-internal-site</id>
      <url>http://myserver/repo</url>
    </repository>
  </repositories>
  ...
</project>
```



# Balise *<server>*

---

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
    http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <servers>
    <server>
      <id>my-internal-site</id>
      <username>my_login</username>
      <password>my_password</password>
      <privateKey>${user.home}/.ssh/id_dsa</privateKey>
      <passphrase>some_passphrase</passphrase>
      <filePermissions>664</filePermissions>
      <directoryPermissions>775</directoryPermissions>
      <configuration></configuration>
    </server>
  </servers>
  ...
</settings>
```



# *Deploy* plugin

---

Le plugin ***deploy*** permet de déployer un artefact dans un dépôt distant.

En général, un dépôt d'entreprise.

Le plugin offre 2 objectifs principaux :

***deploy*** : attaché à la phase *deploy*, il publie l'artefact dans un dépôt distant

***deploy-file*** : Permet de déployer un fichier qui n'a pas été construit par Maven





# *<distributionManagement>*

---

Afin que l'objectif soit effectif, il faut spécifier dans le POM une balise ***<distributionManagement>*** qui indique le dépôt à utiliser

```
<distributionManagement>
  <repository>
    <id>internal.repo</id>
    <name>MyCo Internal Repository</name>
    <url>Host to Company Repository</url>
  </repository>
</distributionManagement>
```



# SNAPSHOTS et RELEASES

---

En général, 2 types d'artefacts sont stockés dans les **dépôts**

- Les **SNAPSHOTS**, ils fournissent le dernier état (plutôt stable) de la version en cours de développement  
=> Les projets dépendants peuvent alors anticiper les futures versions
- Les **releases**, ils sont immuables.  
Ils sont taggés avec le même tag que les sources les ayant produits.



# Dépôts et Releasing

---

Les dépôts d'artefacts

**Ex : Nexus**

Processus de Release



# Nexus

---

***Nexus*** est un gestionnaire de dépôt qui offre principalement 2 fonctionnalités :

- Proxy d'un dépôt distant (Maven central) et cache des artefacts téléchargés
- Hébergement de dépôt interne (privé à une organisation)

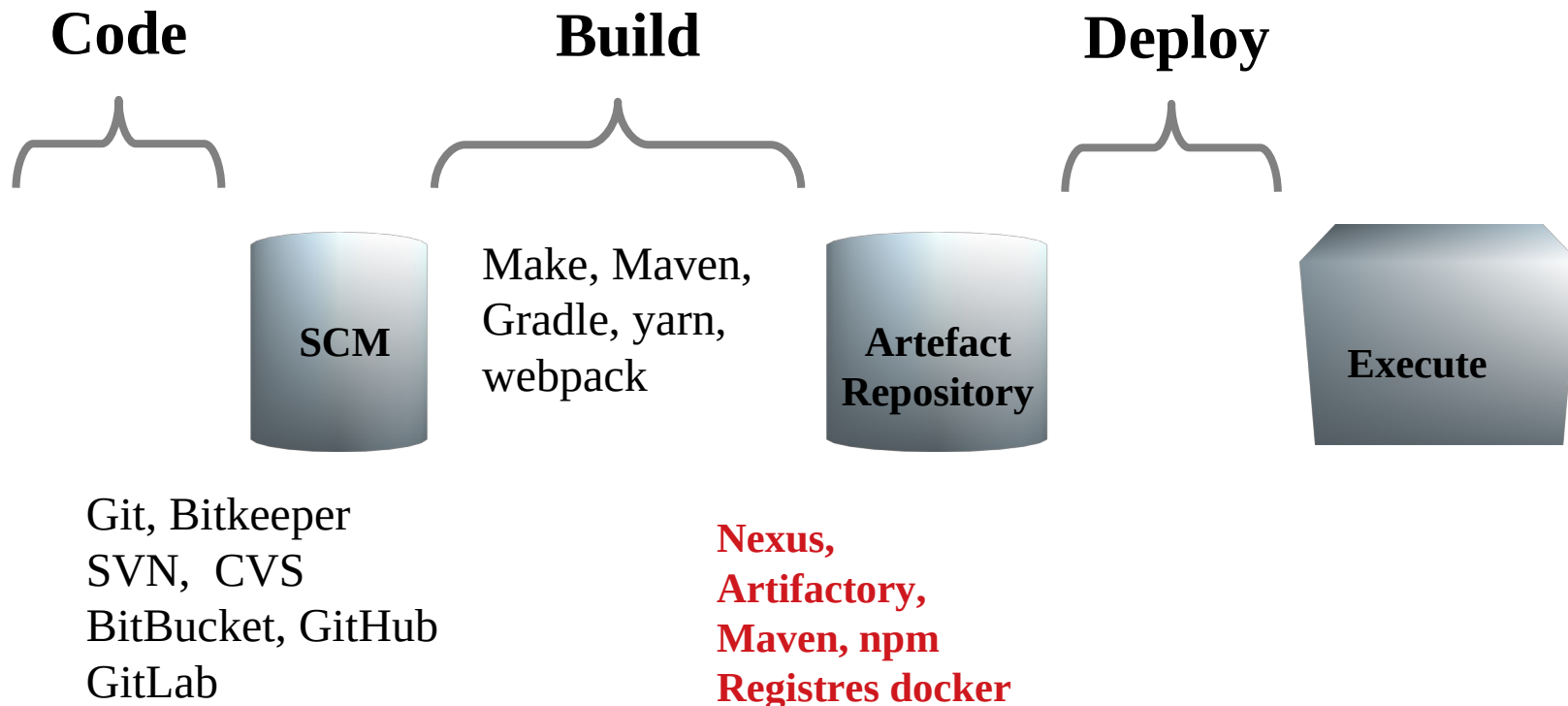
*Nexus* support de nombreux types de dépôts ... et bien sûr les dépôts Maven



# Les gestionnaires de dépôts dans le cycle de vie

Plateforme d'intégration continue

Plateforme de livraison





# Concepts

---

Nexus manipule des **dépôts** et des **composants**

- Les composants sont des artifacts identifiés par leurs coordonnées. Nexus permet d'y attacher des méta-données Pour être générique envers tous les types de composants. Nexus utilise le terme asset
- Les dépôts peuvent être des dépôts Maven, npm, RubyGems, ...



# Types de dépôt

---

Nexus permet de définir différents types de dépôts :

- **Proxy** : Nexus se comporte alors proxy d'un dépôt distant avec des fonctionnalités de cache
- **Hosted** : Nexus stocke des artefacts produits par l'entreprise
- **Group** : Permet de grouper sous une URL unique plusieurs dépôts



# Interface utilisateur

---

*Nexus* fournit un accès anonyme pour la recherche de composants

Si on est loggé, on a accès aux fonctions d'administration (gestion des dépôts, sécurité, traces, API Rest, ...)

La fonctionnalité de recherche propose de nombreuses options





# Installation par défaut

---

La création d'un dépôt dans Nexus s'effectue avec un login Administrateur

*Administration → repositories → Add*

L'installation par défaut a déjà créé le dépôt groupe nommé **maven-public** qui regroupe :

- *maven-releases* : Pour stocker les releases internes
- *maven-snapshots* : Pour stocker les snapshots internes
- *maven-central* : Proxy de Maven central



# Intégration Maven

---

La configuration Maven consiste à modifier le fichier *settings.xml* pour utiliser Nexus comme :

- Proxy de Maven Central
- Dépôt des artefacts snapshots
- Dépôt des artefacts de releases
- Définir un dépôt groupe permettant une URL unique pour les dépôts précédents



# Example

---

```
<servers>
  <server>
    <id>nexus-snapshots</id>
    <username>admin</username>
    <password>admin123</password>
  </server>
  <server>
    <id>nexus-releases</id>
    <username>admin</username>
    <password>admin123</password>
  </server>
</servers>

<mirrors>
  <mirror>
    <id>central</id>
    <name>central</name>
    <url>http://localhost:8081/repository/maven-public/</url>
    <mirrorOf>*</mirrorOf>
  </mirror>
</mirrors>
```



# Configuration projet

---

Au niveau du projet Maven, il faut indiquer

- La balise repository pour télécharger les dépendances à partir de Nexus
- la balise *<distributionManagement>* pour déployer vers nexus



# Configuration projet

---

```
<project>
...
<repositories>
  <repository>
    <id>maven-group</id>
    <url>http://your-host:8081/repository/maven-group/</url>
  </repository>
</repositories>

<distributionManagement>
  <snapshotRepository>
    <id>nexus-snapshots</id>
    <url>http://your-host:8081/repository/maven-snapshots/</url>
  </snapshotRepository>
  <repository>
    <id>nexus-releases</id>
    <url>http://your-host:8081/repository/maven-releases/</url>
  </repository>
</distributionManagement>
```



# Dépôts et Releasing

---

Les dépôts d'artefacts

Ex : Nexus

**Processus de Release**



# Production d'une release

---

La processus de production d'une release consiste généralement en :

- Affecter/Modifier le n° de version de la version SNAPSHOT en cours
- Générer l'artefact et s'assurer que les tous les tests sont OK
- Commiter et Tagger le SCM avec un n° de version
- Publier l'artefact généré dans le dépôt d'artefact et lui donner le même n° de version
- Incrémenter le n° version (et lui apposer le suffixe SNAPSHOT). Committer

Ce processus devra également être automatisé dans le contexte de pipeline CD.



# Automatisation

---

L'automatisation de ce processus peut se faire

- Via des scripts utilisant, les commandes git, l'outil de build et éventuellement l'outil de déploiement vers le dépôt
- Certains plugins d'outils de build implémentent tout le processus (*Maven Release plugin, Gradle Release plugin*)





# Maven et SCM

---

Maven supporte les interactions avec un système de contrôle de version.

L'emplacement du SCM doit être configuré dans le *pom.xml* via la balise **<scm>**

Ainsi, certains plugins (*SCM*, *Release*) peuvent utiliser ces informations pour automatiser des opérations avec le SCM



# Exemple GIT

---

```
<scm>  
<url>https://github.com/kevinsawicki/github-maven-  
  example</url>  
<connection>scm:git:git://github.com/kevinsawicki/  
  github-maven-example.git</connection>  
<developerConnection>scm:git:git@github.com:kevinsawic  
  ki/github-maven-example.git</developerConnection>  
</scm>
```



# Plugin *Release*

---

Le plugin ***Release*** permet d'effectuer des releases en automatisant les modifications manuelles des balises *<versions>* et les opérations avec le SCM.

Une release est effectuée en 2 étapes principales :

***prepare*** : Préparation

***perform*** : Réalisation de la release



# Préparation

---

La préparation effectue les étapes suivantes :

- ✓ Vérification que toutes les sources ont été committées
- ✓ Vérification qu'il n'y a pas de dépendance vers des versions SNAPSHOT
- ✓ Change la versions dans les POM de *x-SNAPSHOT* vers une nouvelle version saisie par l'utilisateur
- ✓ Transforme l'information SCM dans le POM pour inclure le tag de destination de la release
- ✓ Exécute les tests avec les POMs modifiés
- ✓ Commit les POMs modifiés
- ✓ Tag le code dans le SCM avec le nom de version
- ✓ Change les versions dans les POMs avec une nouvelle valeur : *y-SNAPSHOT*
- ✓ Commit les POMs modifiés
- ✓ Génère un fichier *release.properties* utilisé par l'objectif *perform*



# Exemples de commande

---

Reprend la commande ou elle s'est arrêtée

**mvn release:prepare**

Reprend la commande depuis le début

**mvn release:prepare -Dresume=false**

Ou

**mvn release:clean release:prepare**



# *Perform*

---

L'objectif ***perform*** effectue les traitements suivants :

- ✓ Effectue un check-out à partir d'une URL lue dans le fichier *release.properties*
- ✓ Appel de l'objectif prédéfini *deploy*



# Process de release

---

Le process de *release* consiste à appeler successivement les objectifs :

*release:prepare*

*release:perform*

Pour réussir, on doit s'assurer que :

Le *pom* effectif n'a pas de dépendances sur des versions SNAPSHOTS

Tous les changements locaux ont été commités

Tous les tests passent

Tout cela peut se faire en une seule commande maven :

***mvn release:prepare release:perform -B***



# Tests

---

## **Tests unitaires et d'intégration** Couverture des tests





# Introduction

---

Maven intègre les tests dans son cycle de vie

Avec la version 2.2, il fournit un support natif pour l'exécution des tests JUnit avec les plugins ***Surefire*** et ***FailSafe***

- *Surefire* est dédié à l'exécution des tests unitaires
- *FailSafe* aux tests d'intégration



# Configuration JUnit4

---

```
<build>
  <plugins><plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.22.2</version>
  </plugin><plugin>
    <artifactId>maven-failsafe-plugin</artifactId>
    <version>2.22.2</version>
  </plugin></plugins>
</build>
<!-- ... -->
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.vintage</groupId>
    <artifactId>junit-vintage-engine</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```



# Configuration JUnit5

---

```
<build>
  <plugins><plugin>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.22.2</version>
  </plugin><plugin>
    <artifactId>maven-failsafe-plugin</artifactId>
    <version>2.22.2</version>
  </plugin></plugins>
</build>
<!-- ... -->
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```



# Cycle complet Maven

---

*validate* : Valider que le projet est correct

*initialize* : initialiser l'état du build.

*generate-sources* : Générer des sources

*process-sources* : Traite les sources du projet.

*generate-resources* : Génère des ressources .

*process-resources* : Traite ls ressources.

*compile* : compile les sources.

*process-classes* : Traite les classes.

**generate-test-sources** : Génère le code source de test.

**process-test-sources** : Traite les sources.

**generate-test-resources** : Crée des ressources pour le test.

**process-test-resources** : Traite les ressources de test.

**test-compile** : Compilation du code de test

**process-test-classes** : Traitement des .class de test

**test** : Exécution des tests.

*prepare-package* : Préparation au packaging

*package* : Packaging

**pre-integration-test** : Actions avant les tests d'intégration

**integration-test** : Exécutions des tests d'intégration

**post-integration-test** : Actions après les tests d'intégration

**verify** : Phase terminant les tests d'intégration .

*install* : Installation dans le repository local

*deploy* : Déploiement ou release s.



# Plugin *Surefire*

---

Le plugin *Surefire* a un seul objectif :  
***surefire:test*** qui est associé par  
défaut à la phase test de Maven

L'objectif génère des rapports de tests  
en 2 formats dans le répertoire  
*target/surefire-reports*

- Plain text (\*.txt)
- XML (\*.xml)



# Paramètres *Surefire*

---

Paramètres requis :

- ***testSourceDirectory*** : Répertoire des sources de test

Paramètres optionnels :

- ***disableXmlReport*** : Ne pas générer les rapports XML
- ***includes/excludes*** : Inclusion/Exclusion des classes de tests
- ***groups/excludedGroups*** : Groupes ou tags à inclure/exclure
- ***parallel*** : Utilisation de threads
- ***forkCount, forkMode*** : Nbre de threads à démarrer
- ***printSummary*** : Résumé pour les suites de test
- ***reportFormat, reportDirectory*** : Options pour le reporting
- ***testFailureIgnore*** : Continue le cycle Maven même si un test a échoué



# Propriétés de configuration

---

2 autres propriétés passées en ligne de commande sont fréquemment utilisés :

- ***skipTests*** : Permet de faire un build sans exécuter les tests
- ***maven.test.failure.ignore=true*** : Permet de continuer le build même si des tests échouent



# Distinction entre tests unitaires et d'intégration

---

Il n'y a pas d'approche standard pour différencier les tests unitaires des tests d'intégration. Différentes techniques peuvent être utilisées :

1. Convention de nommage: Tous les tests d'intégration peuvent se terminer par “*IntegrationTest*”, ou être placés dans un package particulier.
2. Avec JUnit5, les tests peuvent être différenciés par des tags

Il faut en plus configurer FailSafe afin qu'il exécute les objectifs voulus aux bonnes phases





# Plugin FailSafe

*FailSafe* s'appuie sur *Surefire*.

- Cependant si un test échoue, à la différence de *Surefire* il permet n'arrête pas le cycle Maven et permet d'exécuter la phase *post-integration* dédié à l'arrêt des serveurs (bd, serveur web) utilisés durant les tests

Il comporte 2 objectifs :

- ***failsafe:integration-test*** : Exécution des tests d'intégration avec Surefire. Attaché par défaut à la phase *integration-test*
- ***failsafe:verify*** : Vérification de l'exécution des tests d'intégration. Attaché par défaut à la phase *verify*

Après configuration de *FailSafe*, les tests d'intégration doivent donc se lancer avec :

```
mvn verify
```



# Convention de nommage

---

```
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <executions><execution>
    <goals><goal>test</goal></goals>
    <configuration>
      <excludes><exclude>**/*IntegrationTest.java</exclude></excludes>
    </configuration>
  </execution></executions>
</plugin>
<plugin>
  <artifactId>maven-failsafe-plugin</artifactId>
  <executions><execution>
    <goals> <goal>integration-test</goal> <goal>verify</goal> </goals>
    <configuration>
      <includes><include>**/*IntegrationTest.java</include></includes>
    </configuration>
  </execution></executions>
</plugin>
```



# Séparation selon les tags

---

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.22.2</version>
      <configuration>
        <groups>acceptance | !feature-a</groups>
        <excludedGroups>integration, regression</excludedGroups>
      </configuration>
    </plugin>
  </plugins>
</build>
```



# Tests

---

Tests unitaires et d'intégration  
**Couverture des tests**



# Introduction

La **couverture de code** est une métrique logicielle utilisée pour mesurer le nombre de lignes de code exécutées lors de tests automatisés.

Dans le monde Java, l'outil le plus couramment utilisé est **JaCoCo**

*JaCoCo* fournit un plugin Maven

Il existe également un plugin Eclipse  
*EclEmma*



# Plugin Maven

---

Le plugin Maven fournit 2 principaux objectifs :

- ***prepare-agent*** : Initialisation de JaCoCo, attachement d'un agent à la JVM exécutant le build.  
S'effectue dans la phase initialize
- ***report*** : Génération du rapport de couverture.  
S'effectue après les tests, dans la phase prepare-package par exemple



# Pom.xml

---

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.7.7.201606060606</version>
  <executions>
    <execution>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
      <id>report</id>
      <phase>prepare-package</phase>
      <goals>
        <goal>report</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```



# Mécanisme

Lors de l'exécution des avec JUnit JaCoCo crée un rapport de couverture :




***target/jacoco.exec***

- Ce format est compris par certains outils comme Sonar Qube.

L'objectif ***jacoco:report*** génère des rapports de couverture de code dans plusieurs formats lisibles - par ex. HTML, CSV et XML.



# Résultats

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
 <a href="#">Palindrome</a>		21%		17%	3	5	4	7	0	2	0	1
Total	30 of 38	21%	5 of 6	17%	3	5	4	7	0	2	0	1

```
1. package com.baeldung.testing.jacoco;
2.
3. public class Palindrome {
4.
5.     public boolean isPalindrome(String inputString) {
6.         if (inputString.length() == 0) {
7.             return true;
8.         } else {
9.             char firstChar = inputString.charAt(0);
10.            char lastChar = inputString.charAt(inputString.length() - 1);
11.            String mid = inputString.substring(1, inputString.length() - 1);
12.            return (firstChar == lastChar) && isPalindrome(mid);
13.        }
14.    }
15. }
```



# Analyse

---

Le rapport montre

- le pourcentage d'instructions (bytecode) couvert par les tests,
- le pourcentage de branches couvertes par les tests
- Le nombre de chemins linéairement indépendants couverts par les tests (la complexité cyclomatique)

Une aide visuelle à l'analyse est fournie par des diamants de couleurs pour les branches et des couleurs d'arrière-plan pour les lignes:

- Rouge : aucune branche/lignes n'a été couverte.
- Jaune : Couverture partielle .
- Vert : Couverture totale.



# Seuil minimal

---

*JaCoCo* via l'objectif ***check*** offre un moyen simple de déclarer les exigences minimales à respecter

Si ces exigences ne sont pas respectées, le build échoue

L'objectif *check* est associé par défaut à la phase *verify* de Maven.



# Configuration seuil

---

```
<execution>
  <id>jacoco-check</id>
  <goals>
    <goal>check</goal>
  </goals>
  <configuration>
    <rules>
      <rule>
        <element>PACKAGE</element>
        <limits>
          <limit>
            <counter>LINE</counter>
            <value>COVEREDRATIO</value>
            <minimum>0.50</minimum>
          </limit>
        </limits>
      </rule>
    </rules>
  </configuration>
</execution>
```



# Analyse statique

---

**CheckStyle**  
Sonarqube



# Introduction

---

Le plugin **Checkstyle** génère un rapport concernant le style de code utilisé par les développeurs.

Il apporte 3 objectifs :

- **checkstyle:checkstyle** : Effectue une analyse et génère un rapport sur les violations.
- **checkstyle:checkstyle-aggregate** : Effectue une analyse et génère un rapport HTML agrégé sur les violations dans un projet multi-module.
- **checkstyle:check** : Effectue une analyse et génère des violations ou un nombre de violations dans la console, ce qui peut faire échouer le build. Peut également être configuré pour réutiliser une analyse antérieure.



# Usage

---

```
<project>
  ...
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-checkstyle-plugin</artifactId>
        <version>3.2.1</version>
        <reportSets><reportSet><reports>
          <report>checkstyle</report>
        </reports></reportSet></reportSets>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

mvn site

OU

mvn checkstyle:checkstyle



# Exemple : Configuration du build

---

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-checkstyle-plugin</artifactId>
  <version>3.2.1</version>
  <configuration>
    <!-- Configuration des règles de checkstyle -->
    <configLocation>checkstyle.xml</configLocation>
    <encoding>UTF-8</encoding>
    <consoleOutput>true</consoleOutput>
    <failOnError>true</failOnError>
    <linkXRef>false</linkXRef>
  </configuration>
  <executions>
    <execution>
      <id>validate</id>
      <!-- Association à la phase validate -->
      <phase>validate</phase>
      <goals><goal>check</goal></goals>
    </execution>
  </executions>
</plugin>
```



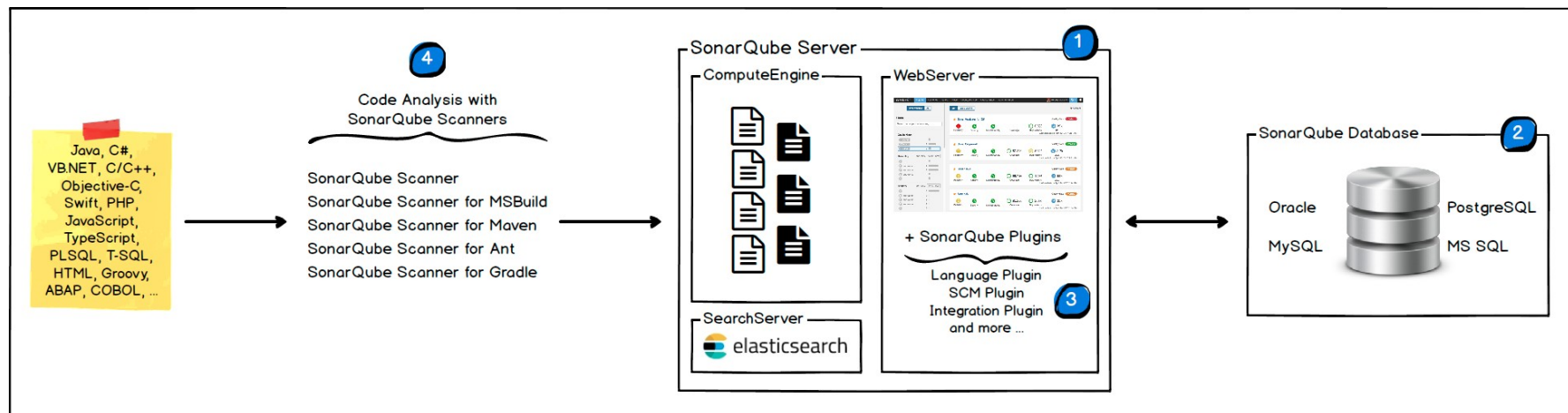


# Analyse statique

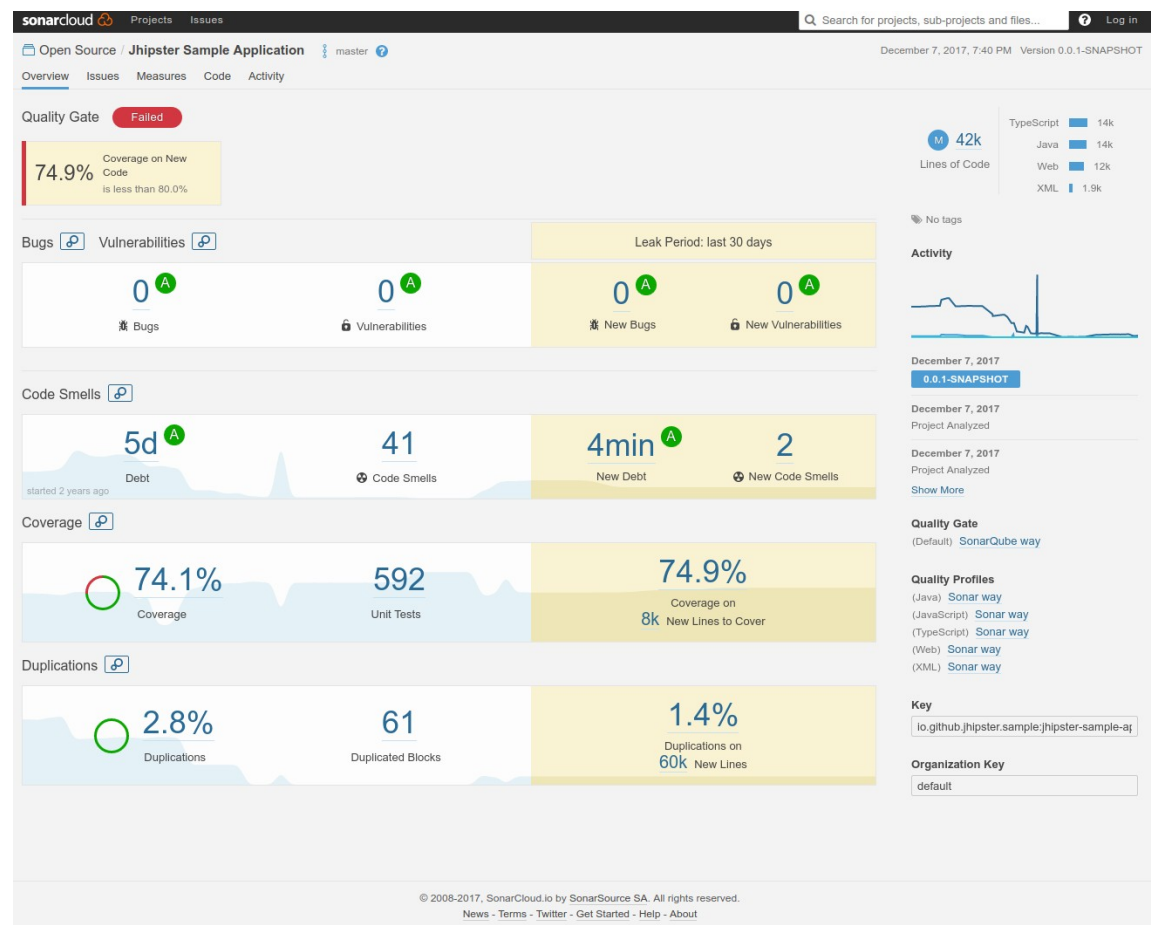
---

CheckStyle  
**Sonarqube**

# Architecture



# Sonar Dashboard



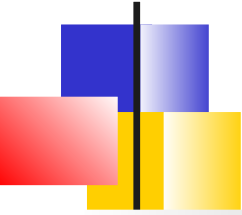


# Analyse et Scanners

---

*Sonar* permet de démarrer une analyse facilement grâce à ses ***Scanners*** fournis

- MSBuild: Projets .Net
- Maven
- Gradle
- Ant
- Jenkins
- Commande en ligne



# Configuration serveur Sonar et groupe de plugins

---

```
<settings>
  <pluginGroups>
    <pluginGroup>org.sonarsource.scanner.maven</pluginGroup>
  </pluginGroups>
  <profiles>
    <profile>
      <id>sonar</id>
      <activation>
        <activeByDefault>true</activeByDefault>
      </activation>
      <properties>
        <sonar.host.url>
          http://myserver:9000
        </sonar.host.url>
      </properties>
    </profile>
  </profiles>
</settings>
```



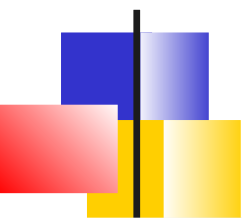
# Analyse

---

L'objectif ***sonar:sonar*** démarre l'analyse

- Il faut en général passer un jeton qui permet de publier l'analyse sur le serveur Sonarqube
- Si l'on veut publier des métriques sur la couverture de test l'analyse doit s'effectuer après l'exécution des tests

```
mvn clean verify sonar:sonar  
-Dsonar.login=authenticationToken
```



# Configuration de l'analyse

---

L'analyse est configurée via des propriétés sonar qui peuvent être indiquées :

- Dans le pom.xml
- Via un fichier properties (par défaut sonar-project.properties)
- En ligne de commande avec -D

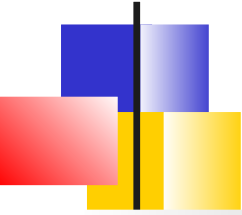


# Exemple Configuration analyse

---

```
<properties>
<java.version>1.8</java.version>
<project.testresult.directory>${project.build.directory}/test-results</
  project.testresult.directory>
<jacoco-maven-plugin.version>0.7.9</jacoco-maven-plugin.version>
<sonar.jacoco.itReportPaths>${project.testresult.directory}/coverage/
  jacoco/jacoco-it.exec</sonar.jacoco.itReportPaths>
  <sonar.jacoco.reportPaths>${project.testresult.directory}/coverage/jacoco/
    jacoco.exec</sonar.jacoco.reportPaths>
  <sonar.java.codeCoveragePlugin>jacoco</sonar.java.codeCoveragePlugin>
<sonar.coverage.exclusions>**/*JWT*.java,**/SecurityConfig.java,**/
  SwaggerConfig.java,**/dto/*.java,**/jwt/*.java</sonar.coverage.exclusions>
<sonar.issue.ignore.multicriteria>S1659</sonar.issue.ignore.multicriteria>
<sonar.issue.ignore.multicriteria.S1659.resourceKey>**/model/*.java</
  sonar.issue.ignore.multicriteria.S1659.resourceKey>
<sonar.issue.ignore.multicriteria.S1659.ruleKey>squid:S1659</
  sonar.issue.ignore.multicriteria.S1659.ruleKey>
</properties>
```





# Exemple Configuration de build

---

```
<build>
<plugins>
  <plugin>
    <groupId>org.sonarsource.scanner.maven</groupId>
    <artifactId>sonar-maven-plugin</artifactId>
    <version>${sonar-maven-plugin.version}</version>
    <executions>
      <execution>
        <id>quality-analysis</id>
        <phase>prepare-package</phase>
        <goals>
          <goal>sonar</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
</plugins>
</build>
```



# Intégration Jenkins

---

Maven plugin, wrapper, docker  
**Pipeline complète**



# Introduction

---

De nombreuses alternatives sont possibles

L'enjeu est d'utiliser une version précise de Maven afin d'obtenir un build reproductible

- Pré-installation de Maven sur les machines de build et définition d'un outil Jenkins par les administrateurs et éventuellement installation du plugin Maven
- Utilisation d'un wrapper Maven commité dans le dépôt qui télécharge automatiquement la bonne version de Maven sur la machine de build
- Utilisation d'une image docker défini dans le Jenkinsfile



# Installation outil

## Maven

### Maven installations

List of Maven installations on this system

Add Maven

≡ Maven

Name

M3

☒ Install automatically ?

≡ Install from Apache

Version

3.8.7

Add Installer ▾



# Exemple Jenkinsfile

---

```
pipeline {
  agent any
  tools {
    // Install the Maven version configured as "M3" and add it to the path.
    maven "M3"
  }
  stages {
    stage('Build') {
      steps {
        // Get some code from a GitHub repository
        git 'https://github.com/jglick/simple-maven-project-with-tests.git'

        // Run Maven on a Unix agent.
        sh "mvn -Dmaven.test.failure.ignore=true clean package"
      }
    }
    post {
      // If Maven was able to run the tests, even if some of the test
      // failed, record the test results and archive the jar file.
      success {
        junit '**/target/surefire-reports/TEST-*.xml'
        archiveArtifacts 'target/*.jar'
      }
    }
  }
}
```

# Plugin Maven

## Gestion des modules

Grâce au plugin Maven, Jenkins comprend la structure en module d'un projet multi-modules et ajoute des entrées dans l'interface pour chaque module

=> Possibilité de visualiser la structure en modules, d'accéder au détail d'un module, de lancer le build d'un module particulier en isolation, ou de configurer spécifiquement un module



The screenshot shows the Jenkins Hudson interface. The top navigation bar includes the 'Hudson' logo, a search bar, and a link to 'ENABLE AUTO REFRESH'. The left sidebar contains a list of navigation links: 'Back to Dashboard', 'Status', 'Changes', 'Workspace', 'Build Now', 'Delete Project', 'Configure', 'Modules' (highlighted), 'Promotion Status', and 'Subversion Polling Log'. The main content area is titled 'Modules' and displays a table with the following data:

S	W	Job ↓	Last Success	Last Failure	Last Duration	
		<a href="#">gameoflife</a>	21 hr (#73)	1 mo 13 days (#41)	1.7 sec	
		<a href="#">gameoflife-core</a>	21 hr (#73)	2 days 12 hr (#68)	11 sec	
		<a href="#">gameoflife-web</a>	21 hr (#73)	2 days 12 hr (#68)	21 sec	
		<a href="#">gameoflife-webservice</a>	21 hr (#73)	2 days 12 hr (#68)	0.61 sec	
		<a href="#">gameoflife-cli</a>	21 hr (#73)	2 days 12 hr (#68)	0.32 sec	



# Sections de configuration d'un job Maven

---

La configuration d'un job est simplifiée. Elle consiste en

- Des configurations générales : Nom, conservation des vieux builds, ...
- L'association à un SCM
- La définition des déclencheurs de build
- Les étapes avant le build : Initialisation, etc.
- Les étapes du build : Appel d'une cible Maven avec passage des options
- Les étapes après le build : Fermeture des ressources, ...
- Actions à la suite du build : Archivage automatique d'artefacts, Publication des résultats, démarrage d'autre build, ...



# Exemple Jenkinsfile avec wrapper

---

```
pipeline {
  agent any

  stages {
    stage('Build') {
      steps {
        // Get some code from a GitHub repository
        git 'https://github.com/jglick/simple-maven-project-with-tests.git'

        // Run Maven on a Unix agent with a wrapper
        sh ".mvnw -Dmaven.test.failure.ignore=true clean package"

      }
    }
  }

  post {
    // If Maven was able to run the tests, even if some of the test
    // failed, record the test results and archive the jar file.
    success {
      junit '**/target/surefire-reports/TEST-*.xml'
      archiveArtifacts 'target/*.jar'
    }
  }
}
```





# Exemple avec plugin docker

---

```
pipeline {  
  agent {  
    docker {  
      image 'maven:3-alpine'  
      args '-v $HOME/.m2:/root/.m2'  
    }  
  }  
  stages {  
    stage('Build') {  
      Steps { sh 'mvn -B' }  
    }  
  }  
}
```



# Intégration Jenkins

---

Maven plugin, wrapper, docker  
**Pipeline complète**