



Maven

David THIBAU – 2021

david.thibau@gmail.com



Agenda

Introduction

- Outils de build
- Principes Maven

Configuration

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

Dépôts et releases

- Les dépôts d'artefacts
- Le processus de release

Particularités SpringBoot

- Starters spring-boot
- Plugin SpringBoot
- Exemple typique



Introduction

Outils de build
Principes Maven



Pré-requis d'un build

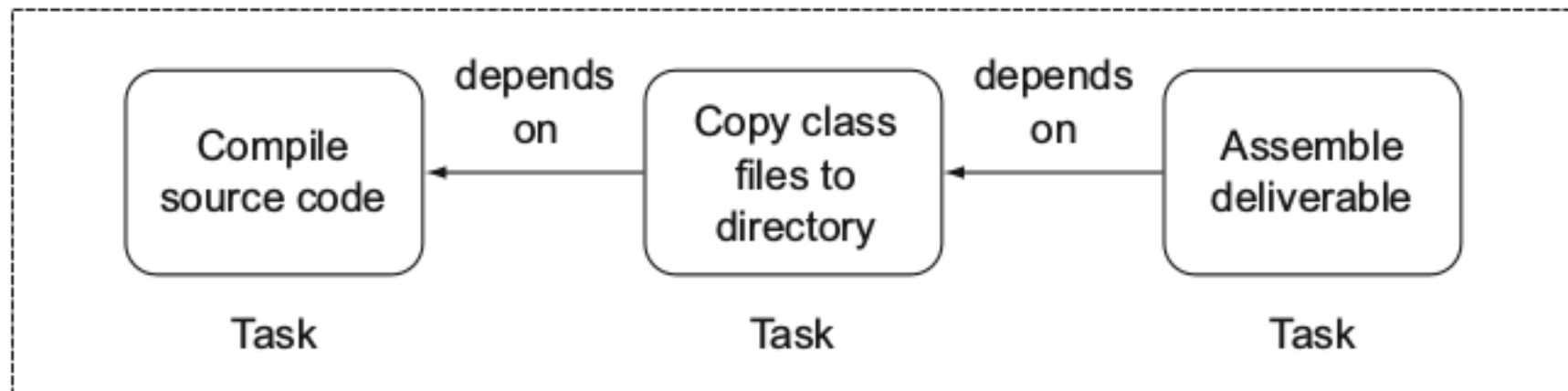
- ♦ **Proscrire les interventions manuelles** sujettes à erreur et chronophage
- ♦ Créer des builds **reproductibles** : Pour tout le monde qui exécute le build
- ♦ **Portable** : Ne doit pas nécessiter un OS ou un IDE particulier, il doit être exécutable en ligne de commande
- ♦ **Sûr** : Confiance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

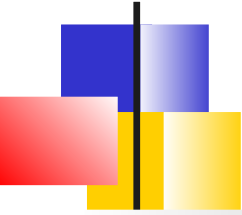
Outils de build
Principes Maven



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



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

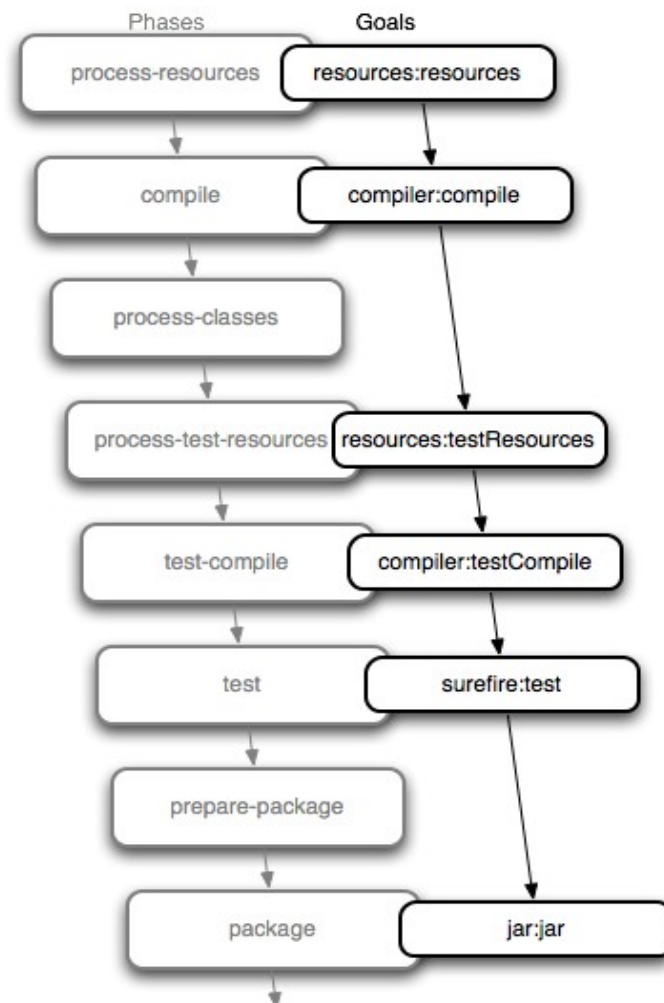
- Le cycle de construction est une **séquence ordonnée de phases** impliquées dans la construction d'un projet
- Pour chaque packaging, Maven supporte différents cycles de vie. Le cycle de construction par défaut est le plus souvent utilisé, pour un packaging jar il commence avec une phase validant l'intégrité du projet et termine avec une phase déployant le projet en production
- 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>
```



Configuration

Gestion des dépendances

Adaptation du cycle de build

Profils de build

Projets multi-modules



Gestion des dépendances

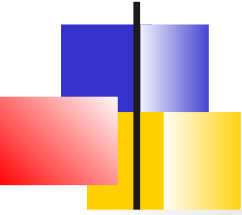
- La possibilité de localiser un artefact à partir de ces coordonnées permet d'indiquer les dépendances dans le projet POM.
- Le fait que le POM associé à tout artefact soit stocké dans le dépôt permet à Maven de résoudre les dépendances transitives. Le support pour les dépendances transitives est une 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

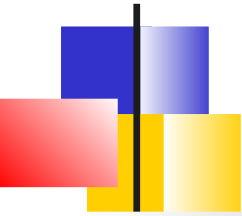
Ajout de dépendances



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 :

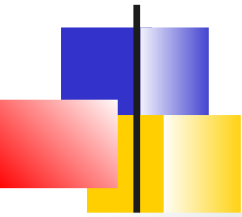
- ***compile*** : Le périmètre par défaut, les dépendances sont tout le temps dans le classpath et sont packagées
- ***provided*** : Dépendance fournie par le container. Présent dans le classpath de compilation mais pas packagée
- ***runtime*** : Présent à l'exécution et lors des tests mais pas à la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

Example



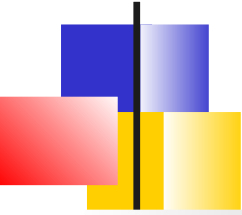
```
<dependency>  
  <groupId>org.apache.commons</groupId>  
  <artifactId>commons-io</artifactId>  
  <version>1.3.2</version>  
  <scope>test</scope>  
</dependency>
```

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

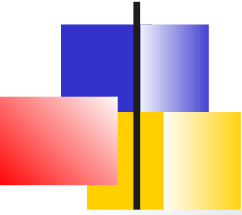


Pour éviter de disperser dans tous les projets de l'entreprise les mêmes dépendances classiques ... et d'avoir des soucis de mis à jour lors d'une montée de version

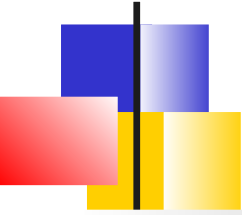
Il est possible de définir via la balise **<dependencyManagement>** les versions des dépendances

Cela est généralement effectué dans un POM parent

<dependencyManagement>

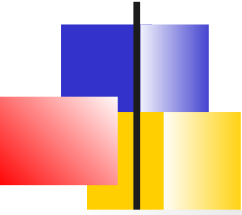
- 
-
- L'élément <dependencyManagement> permet de centraliser les n° de versions des librairie utilisée.
 - Un projet enfant n'a alors plus besoin d'indiquer un n° de version .

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>
```



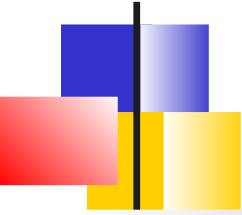
Gestion des versions des plugins

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

Propriétés
Gestion des dépendances
Adaptation du cycle de build
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>
```

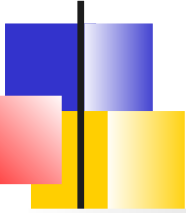


Association objectif/phase.

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
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**



Exemple

```
<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>
```



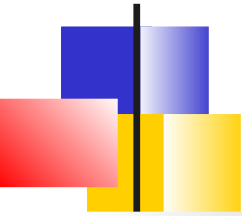
`<profiles>`

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



Configuration

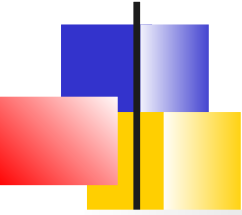
Gestion des dépendances
Adaptation du cycle de build
Profils de build
Projets multi-modules

Projets multi-modules



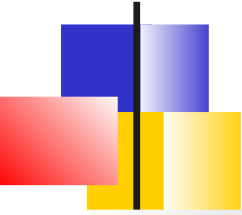
- Un projet multi-modules est défini par un POM parent qui référence plusieurs sous-modules :
- La définition s'effectue via les balises **<modules>** et **<module>**

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>
```

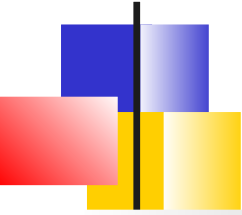
```
    <artifactId>simple-model</artifactId>
```

```
    <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)

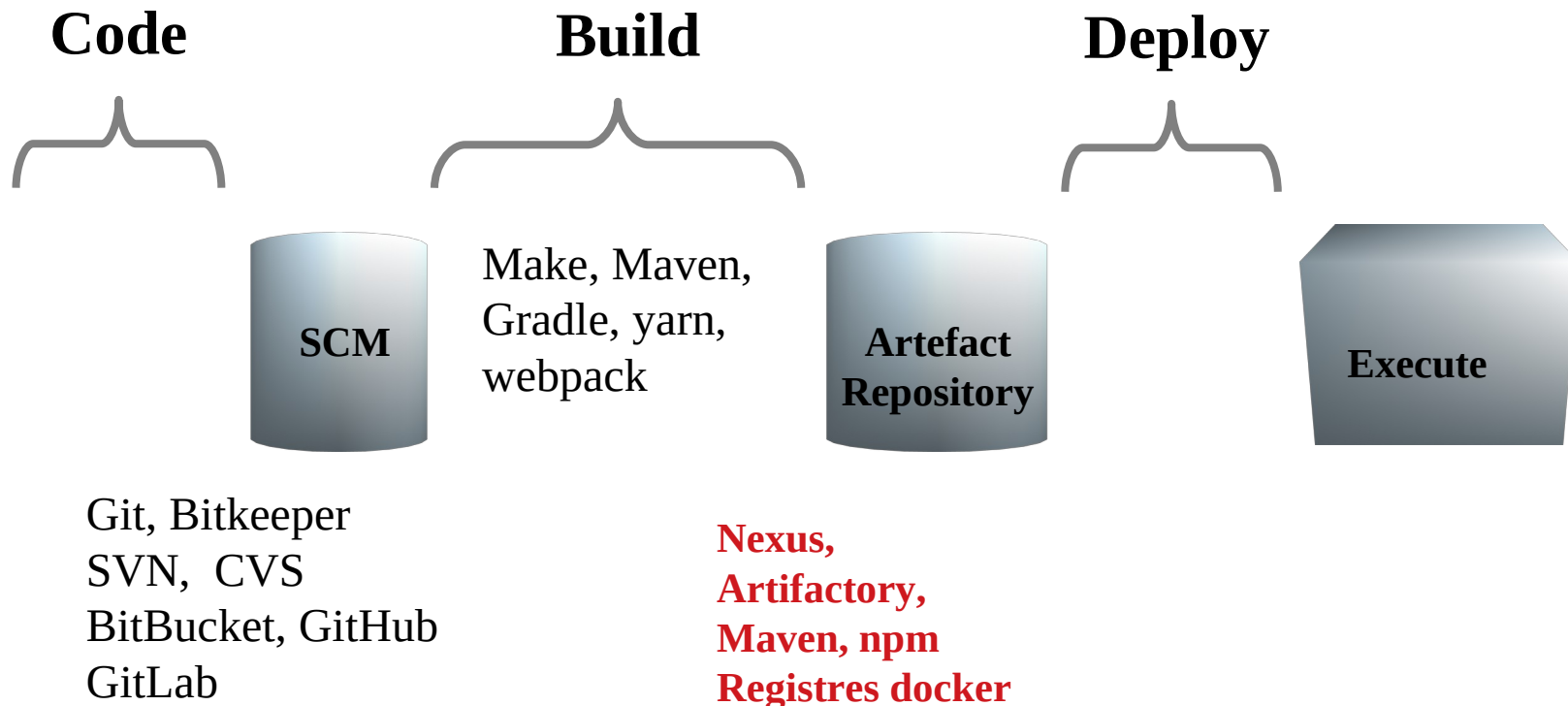
Nexus support de nombreux types de dépôts ... et bien sûr les dépôts Maven



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 commitées
- ✓ 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



Particularités SpringBoot

Starter et gestion des versions

Plugin SpringBoot

Configurations typiques



Java

Gestion des dépendances

Dans un environnement Java, Spring Boot simplifie la gestion de dépendances et de leurs versions :

- Il organise les fonctionnalités de Spring en modules.
=> Des groupes de dépendances peuvent être ajoutés à un projet en important des **"starter" modules**.
- Il fournit un POM parent dont les projets héritent qui gère les versions des dépendances.
=> Le projet ne gère alors qu'un seul n° de version , celui de SpringBoot
- Il propose **"Spring Initializr"**, qui peut être utilisée via un navigateur ou un IDE, qui permet de générer des configurations Maven ou Gradle



Starter Modules

spring-boot-starter-web: librairies de Spring MVC + configuration automatique d'un serveur embarqué (Tomcat, Undertow, ...).

spring-boot-starter-reactive-web: librairies Spring WebFlux + configuration automatique d'un serveur embarqué (Netty).

spring-boot-starter-data-* : Librairies d'accès aux données persistantes (JPA, NoSQL, SolR, ...). Facilite la configuration des sources de données et l'implémentation de la couche DAO

spring-boot-starter-security : librairies de SpringSecurity + configuration simpliste de la sécurité

spring-boot-starter-actuator : Permet l'exposition de points de surveillance via HTTP ou JMX (métriques de performances, sondes, audit sécurité, traces HTTP, ...).



Un unique n° de version

Chaque version de Spring Boot fournit sa liste de dépendances.

- 1 seule numéro de version fixe l'ensemble des versions des dépendances utilisées .
- Lors d'une mise à jour de SpringBoot, toutes les dépendances sont mises à niveau de manière cohérente.

La liste des dépendances est importée dans le projet via le *pom* parent ou le plugin *Gradle*



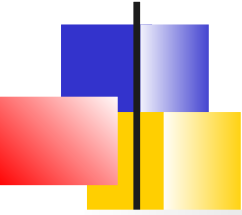
Fichier *pom* Parent

Les projets Maven spring-boot hérite du pom parent **spring-boot-starter-parent** qui contient :

- Un niveau de compilation Java en Java6 ou Java8
- L'encodage UTF-8
- L'importation des dépendances
- Des filtres de ressources. En particulier pour les fichiers de configuration *application-** servant aux configurations spécifiques à des *profils* de build
- La configuration de plugins Maven (*exec plugin, surefire, Git commit ID, shade*)

Pour les projets Gradle, ce sont les plugins

'org.springframework.boot' et **'io.spring.dependency-management'** qui apportent ces fonctionnalités



Exemple Maven

```
<!-- Héritage de Spring Boot -->
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.4.2.RELEASE</version>
</parent>
<properties>
  <java.version>1.8</java.version>
</properties>
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```



Particularités SpringBoot

Starter et gestion des versions

Plugin SpringBoot

Configurations typiques



Introduction

Les plugins Maven/Gradle permettent de :

- créer un package d'archives jar ou war exécutables,
- d'exécuter des applications Spring Boot,
- de générer des informations de build
- de démarrer une application Spring Boot avant d'exécuter des tests d'intégration.



Mise en place Maven

```
<!-- Héritage de Spring Boot -->
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.4.2.RELEASE</version>
</parent>
<properties>
  <java.version>1.8</java.version>
</properties>
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```



Objectifs Maven

spring-boot:repackage : Repackage le jar applicatif en un fat jar

spring-boot:build-image : Package une application en image OCI (docker) en utilisant buildpack.

spring-boot:build-info : Génère un fichier *build-info.properties* contenant les informations de version du pom.xml

Et aussi :

- *spring-boot:help* : Aide
- *spring-boot:run* : Exécute l'application
- *spring-boot:start* : Exécute l'application en background, permet l'exécution de tests d'intégration
- *spring-boot:stop* : Arrête l'application en background



Particularités SpringBoot

Starter et gestion des versions
Plugin SpringBoot
Exemple typique