Cours 2 : Maven

Excilys
Développeurs de passion

Excilys
Développeurs de passion

SOMMAIRE

Maven

Gestionnaire de build

Gestionnaire de build

Définition :

- De façon générale, un gestionnaire de build est un outil qui permet d'exécuter un script contenant une suite de règles qui servent à accomplir des d'objectifs, souvent appelés cibles ("target").

Gestionnaire de build

Il existe différents gestionnaires de build:

- Ant (Ancêtre de Maven)
- Gradle (Java, Android)
- Maven (Java)
- Make, CMake (All)
- et bien d’autres…

Gestionnaire de build

Maven repose sur l'utilisation de plusieurs concepts :

- Les artéfacts : composants identifiés de manière unique
- Le principe de convention over configuration
- Le cycle de vie et les phases : les étapes de construction d'un projet sont standardisées
- Les dépôts (local et distant)

Les artéfacts

Les artéfacts

- Un artéfact est un composant packagé possédant un identifiant unique appelé aussi coordonnées du projet : un groupId, un artifactId et un numéro de version.

- La gestion des versions est importante pour identifier quel artéfact doit être utilisé : la version est utilisée comme une partie de l'identifiant d'un artéfact.

Les artéfacts

```
<groupId>com.ensta</groupId> // nom du package principal
<artifactId>cours-maven</artifactId> // nom du projet
<version>0.0.1-SNAPSHOT</version>
```

Exemple decoordonnées d'un artéfact Maven

Les artéfacts

- Un projet en cours de développement peut être identifié par un numéro de version contenant le suffixe -SNAPSHOT.

- Maven va systématiquement rechercher une version plus récente pour une dépendance dont le numéro de version est un -SNAPSHOT.

Les archétypes

Les archétypes

Un archétype est un modèle de projet. Maven propose en standard plusieurs archétypes tels que:

- maven-archetype-quickstart : Un archétype pour un exemple de projet Maven
- maven-archetype-webapp: Un archétype pour un projet de type application web

Gestionnaire de build

Répertoire	Contenu
/src	les sources du projet (répertoire qui doit être ajouté dans le gestionnaire de sources)
/src/main	les fichiers sources principaux
/src/main/java	le code source (sera compilé dans /target/classses)
/src/main/resources	les fichiers de ressources (fichiers de configuration, images, ...). Le contenu de ce répertoire est copié dans target/classes pour être inclus dans l'artéfact généré
/src/main/webapp	les fichiers de la webapp
/src/test	les fichiers pour les tests
/src/test/java	le code source des tests (sera compilé dans /target/test-classses)
/src/test/resources	les fichiers de ressources pour les tests
/target	les fichiers générés pour les artéfacts et les tests (ce répertoire ne doit pas être inclus dans le gestionnaire de sources)
/target/classes	les classes compilées
/target/test-classes	les classes compilées des tests unitaires
/target/site	site web contenant les rapports générés et des informations sur le projet
/pom.xml	le fichier POM de description du projet

L'arborescence d'un projet Maven WebApp

Les dépendances

Les dépendances

Une dépendance Maven c’est :

- Une bibliothèque packagée qui permet d'en utiliser le contenu dans le projet.
- Elle peut se présenter sous forme:
  - D’archive (jar/war/ear..).
  - D’un fichier pom regroupant d’autre dépendances.

Les dépendances

La gestion des dépendances de Maven repose sur plusieurs concepts :

- les dépôts : permet de stocker les artéfacts
- la portée (scope) : précise à quelle étape de l'application, la dépendance est ajoutée
- la transitivité : gestion des dépendances de dépendances

Les dépendances de B sont chargées automatiquement par A lorsque ce dernier inclut B dans sa liste de dépendances.

Dépendances transitives

Les dépendances

Maven utilise la notion de portée (scope) pour préciser comment la dépendance sera utilisée. Maven définit quatre portées pour les dépendances :

- **compile** : la dépendance est utilisable pendant toutes les phases et à l'exécution. C'est le scope par défaut.
- **provided** : la dépendance est utilisée pour la compilation mais elle ne sera pas déployée car elle est fournie par l'environnement d'exécution.
- **runtime** : la dépendance n'est pas utile pour la compilation mais elle est nécessaire à l'exécution. C'est par exemple le cas des pilotes JDBC.
- **test** : la dépendance n'est utilisée que lors de la compilation et de l'exécution des tests. C'est par exemple le cas pour la bibliothèque utilisée pour les tests unitaires (JUnit ou pas exemple).

Les dépendances

```
<dependencies>
   <!-- … -->
   <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-lang3</artifactId>
      <version>3.9</version>
      <scope>compile</scope>
   </dependency>
   <!-- … -->
</dependencies>
```

Ajout d’une dépendance au projet

Les dépôts

Les dépôts

- Maven utilise la notion de référentiel ou dépôt (repository) pour stocker les dépendances et les plugins requis pour générer les projets.

- On distingue deux types de dépôts : local et distant (remote). Ces dépôts peuvent être gérés à plusieurs niveaux.

Les dépôts

- Dépôt central : il stocke des dépendances et les plugins utilisables de manière open source.
- Dépôt local : il stocke une copie des dépendances et plugins requis par les projets à générer en local. Ces artéfacts sont soit téléchargés des dépôts centraux, soit créés avec Maven. Le dépôt local se trouve dans le dossier `~/.m2` de l'ordinateur (Sous Unix).
- Un dépôt au niveau entreprise : celui-ci permet de limiter les accès réseau pour les dépendances utilisées dans de multiples projets de l'entreprise. Il joue un rôle de miroir pour le dépôt central.

Les dépôts

Maven recherche un élément dans l'ordre suivant :

- sur le dépôt local
- sur les dépôts «entreprise» configurés dans le projet
- sur le dépôt central

Les dépôts

- La configuration du chemin du dépôt local se fait dans le fichier settings.xml du répertoire .m2 contenu dans le répertoire home de l'utilisateur.
- Le tag <localRepository> permet de préciser le chemin absolu du dépôt local.

Les dépôts

Quelques raisons pour vouloir utiliser un miroir particulier :

- Il existe un miroir plus proche et donc plus rapide.
- On voudrait remplacer le repository distant par un repository interne sur lequel on a un plus grand contrôle.
- On utilise un gestionnaire de repository pour stocker en cache des dépendances.

Le fichier POM

Le fichier POM

- Le fichier POM (Project Object Model) contient la description du projet Maven.
- Le fichier POM doit être à la racine du répertoire du projet.
- Le tag racine est le tag <project>.

Le fichier POM

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.ensta</groupId>
  <artifactId>cours-maven</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>
  <description>Exemple de projet Maven</description>

  <dependencies>
    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-lang3</artifactId>
      <version>3.9</version>
    </dependency>
  </dependencies>
</project>
```

Coordonnées du projet

Dépendances du projet

Un exemple de fichier pom.xml

Le fichier POM

Il est possible de créer des variables grâce au tag <properties> :

```
<properties>
   <my.awesome.variable>Something</my.awesome.variable>
   <variable>42</variable>
</properties>
```

Les variables s'utilisent de la manière suivante :

```
${my.awesome.variable}
```

Les plugins

Les plugins

- Maven en lui-même n'est composé que d'un noyau très léger.

- Toutes les fonctionnalités pour générer un projet sont sous la forme de plugins qui doivent être présents dans le référentiel local ou téléchargés lors de la première utilisation.

- La déclaration et la configuration des plugins à utiliser se fait dans le fichier POM.

Les plugins

- Un plugin est un artéfact Maven.
- Un plugin contient un ou plusieurs goals.
- Il est aussi possible de développer ses propres plugins.

Les plugins

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-checkstyle-plugin</artifactId>
      <version>${maven-checkstyle-plugin.version}</version>
      <configuration>
        <configLocation>src/main/resources/checkstyle.xml</configLocation>
        <failsOnError>true</failsOnError>
        <failOnViolation>true</failOnViolation>
        <violationSeverity>warning</violationSeverity>
        <consoleOutput>true</consoleOutput>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Ajout du plugin maven-checkstyle-plugin

```
$ mvn checkstyle:check
```

Utilisation du **goal** check à partir du **plugin** checkstyle

Cycle de vie du projet

Cycle de vie du projet

La construction d'un projet Maven repose sur la notion de cycle de vie qui définit de manière claire les différentes étapes nommées phases.

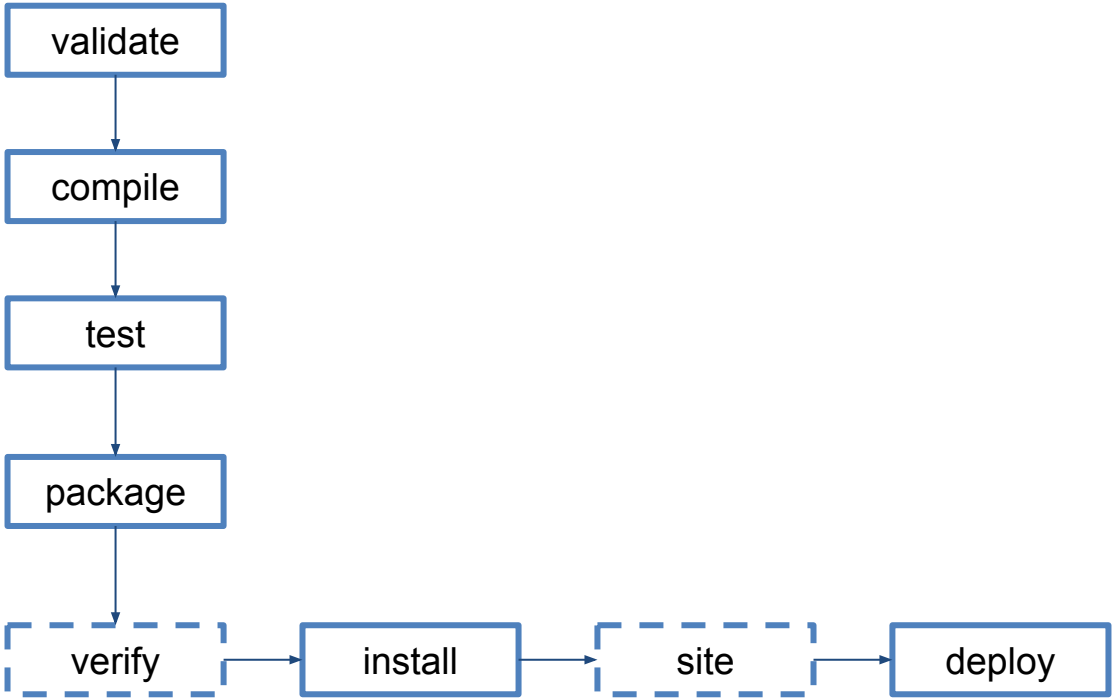
Trois cycles de vie sont définis :

- **default** : il permet de générer et déployer l'artéfact du projet
- **clean** : il permet de nettoyer les fichiers générés du projet
- **site** : il permet de générer un site web pour accéder à la documentation du projet.

Cycle de vie du projet

- Un cycle de vie est composé de phases qui constituent les étapes de la génération de l'artéfact.
- Le cycle de vie par défaut de Maven (build) propose plusieurs phases. Une liste exhaustive ici.
    - Par exemple lancer la commande mvn install va exécuter chaque phase du cycle de vie par défaut dans l'ordre (validate, compile, package, etc.), avant d'exécuter la phase install.

Cycle de vie



Phases du cycle de vie “default”

Cycle de vie

Phase	Rôle
compile	Compiler le code source du projet (génère le bytecode)
package	Générer l'artéfact sous sa forme diffusable (jar, war, ear, ...)
install	Installer l'artefact dans le dépôt local pour qu'il puisse être utilisé comme dépendance d'autres projets
deploy	Envoie le package dans le repository distant défini dans le POM, pour qu'il soit utilisable comme dépendance d'un autre projet

Cycle de vie

- Il est possible de lier l'exécution de **goals** de plugins à une **phase** de Maven.

  Par exemple, on peut vérifier le formattage du code avec le goal **check** du plugin **checkstyle** à chaque fois que la phase **test** est exécutée.

- Pour qu'une phase Maven soit exécutée, il faut qu'au moins un goal de plugin soit lié.

Cycle de vie

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-checkstyle-plugin</artifactId>
      <version>${maven-checkstyle-plugin.version}</version>
      <configuration>
        <configLocation>src/main/resources/checkstyle.xml</configLocation>
        <failsOnError>true</failsOnError>
        <failOnViolation>true</failOnViolation>
        <violationSeverity>warning</violationSeverity>
        <consoleOutput>true</consoleOutput>
      </configuration>
      <executions>
        <execution>
          <phase>test</phase>
          <goals>
            <goal>check</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Nom du plugin

Phase Maven

Goal à exécuter

Lier un goal de plugin à une phase de Maven

Références

- https://maven.apache.org/pom.html
- https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

Excilys
Développeurs de passion