

Génie Logiciel Avancée

Architecture logicielle évoluée :

Framework Spring

01- Outil Maven

Mohamed ZAYANI

2025/2026

Plan

- Mise au contexte
- Exemples de problématiques résolues par Maven
- Présentation du Maven
- Caractéristiques du Maven
- Notion d'artefact
- Gestion de dépendances et plugins
- Repository Maven
- Les propriétés
- Archetypes Maven
- Principe de Fonctionnement
- Cycle de vie d'un projet Maven
- Exemples de commandes Maven

Mise au contexte

- La création d'un projet logiciel **nécessite** souvent des tâches telles que:
 - ❖ le téléchargement des dépendances,
 - ❖ l'alignement des versions compatibles des bibliothèques nécessaires,
 - ❖ la compilation du code source,
 - ❖ l'exécution des tests,
 - ❖ le packaging du projet, etc.
 - Ces opérations peuvent être **fastidieuses et sources d'erreurs** si elles sont effectuées manuellement.
- Les outils de construction automatisent considérablement ces tâches.
- Maven est l'un des outils de construction les plus populaires de l'écosystème Java tels que **Ant** et **Gradle**. (Maven reste le plus répandu dans le monde JEE et Spring : côté Backend)
- Pour d'autres plateformes et langages de programmation, Maven est similaire à:
- **npm ou Yarn** pour JavaScript/TypeScript
 - **NuGet** pour .Net
 - **composer** pour PHP

Problématiques sans Maven

1. Gestion manuelle des bibliothèques (JARs):

- ❖ Le cas où un projet utilise **Hibernate**.
 - ❖ Et Hibernate dépend de plusieurs autres bibliothèques(ex. **antlr**, **dom4j**, **javassist**, **cplib**...).
 - ❖ Sans Maven, le développeur va chercher chaque JAR manuellement sur Internet, les télécharger, les copier dans un dossier lib/.
- Si Hibernate sort une nouvelle version, on doit tout recommencer.

☞ **Problème : perte de temps + erreurs fréquentes (JAR manquant, mauvaise version...)**

2. Conflits de versions:

- ❖ Exemple:
- ❖ Le projet utilise la Bibliothèque A (v1.0) qui dépend de la Bibliothèque C (**v1.0**).
- ❖ Le projet utilise aussi la Bibliothèque B (v2.0) qui dépend de Bibliothèque C (**v2.5**).
- ❖ Résultat: Le projet contient 2 versions différentes de C

→ erreurs de compilation ou de runtime (**ClassNotFoundException**, **NoSuchMethodError**)..

☞ **Problème : sans Maven, il faut résoudre manuellement les conflits de versions**

Problématiques sans Maven

3. Pas de standardisation des projets:

- ❖ Chaque développeur organise ses projets comme il veut : (/src , /bin , /lib , /MesClasses, ..)
 - Difficile pour un nouveau développeur de comprendre la structure.
 - Difficile d'automatiser le build dans une équipe.

☞ **Problème : manque de convention et portabilité**

4. Compilation et exécution fastidieuses:

- ❖ Exemple:

```
javac -cp ".;lib/*"  src/com/ms/app/*.java -d bin
```

```
java -cp ".;bin;lib/*" com.ms.app.Main
```

→ Il est nécessaire d'indiquer manuellement les chemins vers les classes et les JARs.

☞ **Problème : sans Maven, il faut résoudre manuellement les conflits de versions**

5. Pas d'intégration facile avec CI/CD:

- ❖ Sans Maven : pas de cycle de build standardisé.
- Difficile d'automatiser dans Jenkins, GitLab CI, etc., .

☞ **Problème : Chacun doit réinventer son script de build.**

Solutions avec Maven

1. Gérer les bibliothèques (dépendances) :

- ❖ Pour le cas de **Hibernate**, déclarer dans un fichier de configuration (**pom.xml**):

```
<dependencies>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>6.5.2.Final</version>
    </dependency>
</dependencies>
```

- ❖ Ainsi, Maven télécharge Hibernate et toutes ses dépendances transitives automatiquement.
- ❖ Puis met tout ça dans le classpath
→ Plus besoin de chercher à la main.

2. Gestion automatique des versions:

- ❖ Si deux bibliothèques ramènent deux versions différentes de C, Maven détecte le conflit.
- ❖ Et applique une stratégie (plus proche dans l'arborescence ou version la plus récente).
- ❖ Avec la possibilité de forcer une version donnée.

Solutions avec Maven

3. Convention standard de structure:

- ❖ Avec Maven, tout projet suit la même organisation:

- `src/main/java`
- `src/main/resources`
- `src/test/java`
- `src/test/resources`

→ Facile à comprendre et portable d'un développeur à l'autre.

4. Construction simple et portable:

- ❖ Compilation + test + packaging en une seule commande:

`mvn clean package`

→ Maven gère les classpath, JARs et plugins.

5. Intégration CI/CD:

- ❖ Tous les outils (Jenkins, GitLab CI, GitHub Actions...) reconnaissent les commandes Maven.
- ❖ Il suffit de lancer `mvn compile` ou `mvn install` dans un pipeline, sans script compliqué..

Présentation du Maven

- Maven est un outil de gestion de projet logiciel pour Java maintenu par « **l'Apache Software Foundation** ».
- Maven est un outil qui permet de gérer les dépendances (les bibliothèques externes): il assure le téléchargement automatique de ces bibliothèques depuis un dépôt central évitant ainsi leur stockage local
- Maven permet d'automatiser la construction d'un projet JAVA :
 - ❖ compilation, test,
 - ❖ packaging,
 - ❖ déploiement,
 - ❖ production de livrable
 - ❖ gestion de sites web
- Maven permet aussi de générer des documentations (sous forme de rapports) concernant le projet.
→ Maven permet aux développeurs de se concentrer sur l'écriture du code plutôt que sur la gestion des complexités de la configuration du projet.



Caractéristiques Maven

Principe de Convention plutôt que configuration

- Maven établit un certain nombre de conventions afin d'automatiser certaines tâches et rendre la procédure de configuration plus facile.
- Une de ces conventions est de fixer l'arborescence d'un projet. Ainsi, Maven permet de générer une squelette du code du projet en utilisant la notion des « archetypes ».

Approche déclarative

- Maven utilise une approche déclarative où la structure du projet et son contenu sont décrits dans un document XML nommé **POM.xml**
(Project Object Model)

Aspect extensible

- Maven est **extensible** grâce à un mécanisme de plugins qui permettent d'ajouter des fonctionnalités.

Notion d'artefact

- Un **artefact** est un composant packagé possédant un identifiant unique composé de trois éléments : **un groupId, un artifactId et un numéro de version.**
 1. **groupId** : définit l'organisation ou groupe qui est à l'origine du projet. Il est formulé sous la forme d'un package Java
(Exemple : org.jee.maven)
 2. **artifactId** : définit le nom du projet (nom unique dans le groupe)
(Exemple: premierProjet)
 3. **version** : définit la version du projet. Les numéros de version sont souvent utilisés pour des comparaisons et des mises à jour.

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

Exemple

- La déclaration d'une dépendance est spécifiée dans le fichier « **pom.xml** » (cœur de Maven)

- Exemple:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate</artifactId>
    <version>3.3.2</version>
    <scope>compile</scope>
</dependency>
```

- Maven associe à cette dépendance un fichier JAR nommé : **hibernate-3.3.2.jar** selon la convention:

<artifactId>-<version>.jar

- La notion de « **scope** » définit la portée d'une dépendance: Elle permet de préciser dans quel contexte une dépendance est utilisée (**compile, test, provided, ..**):

- ❖ Portée « **test** »: la dépendance est utilisée uniquement dans la phase de test unitaire et elle n'est pas prise dans l'opération de packaging.
- ❖ Portée « **provided** »: la dépendance est utilisée dans toutes les phases et elle n'est pas prise dans l'opération de packaging (elle est remplacée par celle de l'environnement de déploiement).
- ❖ Portée « **compile** » : la dépendance est utilisable par toutes les phases et à l'exécution. C'est le scope par défaut

Fichier « pom.xml »

- Le fichier **POM** (Project Object Model) contient la description du projet Maven. Il contient les informations nécessaires à la génération du projet : (identification de l'artefact, déclaration des dépendances, définition d'informations relatives au projet..). Voici un exemple:

```
<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>com.jee.test</groupId>
  <artifactId>MaWebApp</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Mon application web</name>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.7</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Balises du fichier « pom.xml »

dossier	Description
<code><modelVersion></code>	Préciser la version du modèle de POM utilisée
<code><groupId></code>	Préciser le groupe ou l'organisation qui développe le projet. C'est une des clés utilisée pour identifier de manière unique le projet et ainsi éviter les conflits de noms
<code><artifactId></code>	Préciser la base du nom de l'artéfact du projet
<code><packaging></code>	Préciser le type d'artéfact généré par le projet (jar, war, ear, pom, ...). La valeur par défaut est jar
<code><version></code>	Préciser la version de l'artéfact généré par le projet. Le suffixe - SNAPSHOT indique une version en cours de développement
<code><name></code>	Préciser le nom du projet utilisé pour l'affichage

Balises du fichier « pom.xml »

dossier	Description
<description>	Préciser une description du projet
<url>	Préciser une url qui permet d'obtenir des informations sur le projet
<dependencies>	Définir l'ensemble des dépendances du projet
<dependency>	Déclarer une dépendance en utilisant plusieurs tags fils : <groupId>, <artifactId>, <version> et <scope>

- Le fichier POM doit être à la racine du répertoire du projet.
- La balise racine du fichier « pom.xml » est la balise <**project**>.

Dépendance vs Plugin

- Une dépendance est une bibliothèque externe requise par le projet dans un phase donnée (compilation, test, exécution..)
- Exemple: Besoin d'utiliser la bibliothèque JUnit pour tester le code source

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

- Un plugin sert à étendre Maven: c'est un outil qui ajoute de nouvelles tâches au processus de construction.
- Exemple: Besoin d'un serveur Tomcat embarqué pour une application web

```
<plugins>
  <plugin>
    <groupId>org.apache.tomcat.maven</groupId>
    <artifactId>tomcat7-maven-plugin</artifactId>
    <version>2.2</version>
  </plugin>
<plugins>
```

- Il est possible ainsi d'exécuter l'application web à travers la commande : mvn tomcat7:run

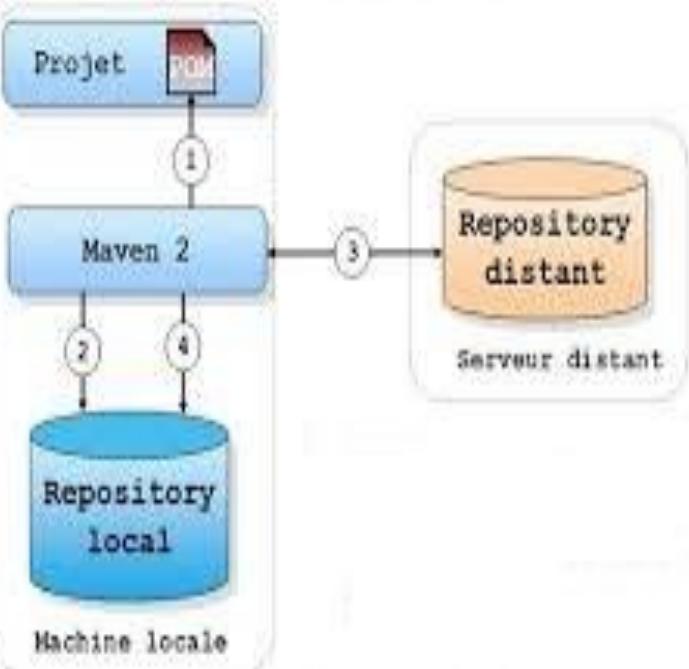
Gestion de dépendances et de plugins

- Maven utilise la notion de référentiel ou dépôt (**repository**) pour stocker les **dépendances** et les **plugins** requis pour générer les projets.
- Un dépôt contient un ensemble d'artéfacts qui peuvent être des livrables, des dépendances, des **plugins**, ...
- Ceci permet de **centraliser** ces éléments qui sont généralement utilisés dans plusieurs projets : c'est notamment le cas pour les **plugins** et les **dépendances**.
- Maven distingue deux types de dépôts : **local** et **distant (remote)**:
 1. **dépôt central (repository central)**: il stocke des dépendances et les **plugins** utilisables par tout le monde car disponible sur le web; ce sont généralement des artéfacts open source
 2. **dépôt local (repository local)** : il stocke une copie des dépendances et **plugins** requis par les projets à générer en local. Ces artéfacts sont téléchargés des dépôts centraux

Dépôt local et dépôt central

- Maven utilise un ou plusieurs dépôts (**repository**) qui peuvent être locaux ou distants
- Si un élément n'est pas trouvé dans le répertoire local, il sera téléchargé dans ce dernier à partir d'un dépôt distant

Gestion des dépendances par Maven 2



1. Lecture du fichier « **pom.xml** » et la liste des dépendances
2. Vérification de l'existence des dépendances (ou plugins) dans le **repository local** (dépôt local)
3. Téléchargement des dépendances non trouvées en accédant au **repository central** (via Internet)
4. Copies de dépendances dans le **repository local**

Repository Maven

- La première exécution d'une commande « Maven », un dossier nommé « **.m2** » est créé dans le répertoire « **HOME** » de l'ordinateur
(Exemple: **C:\Utilisateurs\Ma_Machine**)
- Le dossier « **.m2** » , ainsi créé, comporte un sous-dossier « **repository** » constituant le dépôt local de Maven
- Il est possible de personnaliser l'emplacement du **repository local** en spécifiant le chemin dans un fichier « **settings.xml** » à placer dans le dossier « **.m2** ».



Les propriétés

- Les propriétés personnalisées peuvent faciliter la lecture et la maintenance du fichier pom.xml.
- Dans un cas d'utilisation classique, on utilise des propriétés personnalisées pour définir les versions des dépendances du projet.
- Les propriétés Maven sont des espaces réservés aux valeurs et sont accessibles n'importe où dans un pom.xml en utilisant la notation \${name} où name est la propriété.

```
<properties>
    <spring.version>5.3.16</spring.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>${spring.version}</version>
    </dependency>
</dependencies>
```

Nom de la propriété

Utilisation de la propriété

Archetypes Maven

- Afin de générer la squelette d'un projet, Maven s'appuie sur des archétypes (ou modèles).
- Un Archetype est un outil pour faire des templates de projet.
- Un projet généré via un Archetype est dit **projet Maven**.
- Un Archetype est un modèle de projet à partir duquel d'autres projets sont créés.
- L'utilisation d'archétypes a pour principal avantage de normaliser le développement de projets et de permettre aux développeurs de suivre facilement les meilleures pratiques tout en débutant leurs projets plus rapidement.
- Maven fournit, aux utilisateurs, une très grande liste de différents types de modèles de projet en utilisant le concept d'Archetype.

Exemples d'Archetypes Maven

archetype	Description
quickstart	Contient un exemple projet Maven standard
simple	Contient un simple projet Maven
webapp	Contient un exemple projet Maven d'une application web
j2ee-simple	Contient un exemple projet Maven d'une application JEE

- **Maven** permet de créer la structure d'un projet selon un modèle donné (archetype) en utilisant la commande suivante:

mvn archetype:generate

Structure d'un projet Maven

dossier	Description
/src	les sources du projet
/src/main	les fichiers sources principaux
/src/main/java	le code source (sera compilé dans /target/classes)
/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'artefact généré
/src/main/webapp	les fichiers de l'application web
/src/test	les fichiers pour les tests
/src/test/java	le code source des tests (sera compilé dans /target/test-classes)

Structure d'un projet Maven

dossier	Description
/src/test/resources	les fichiers de ressources pour les tests
/target	les fichiers générés pour les artefacts et les tests
/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

NB: l'arborescence d'un projet Maven est par défaut imposée par l'outil Maven
(par convention)

Principe de fonctionnement du Maven 3

- Toutes les fonctionnalités décrites ici font partie de la version **3** de Maven.
- Une connexion à internet est nécessaire pour permettre le téléchargement des plugins requis et des dépendances.
- Pour installer Maven:
 - ❖ Télécharger l'archive sur le site:
<http://maven.apache.org/download.html>
 - ❖ Décompresser l'archive dans un répertoire du système
 - ❖ Créer la variable d'environnement **M2_HOME** qui pointe sur le répertoire contenant Maven
 - ❖ Ajouter le chemin **M2_HOME/bin** à la variable **PATH** du système
 - ❖ Pour vérifier l'installation, il suffit de lancer la commande:

mvn -version

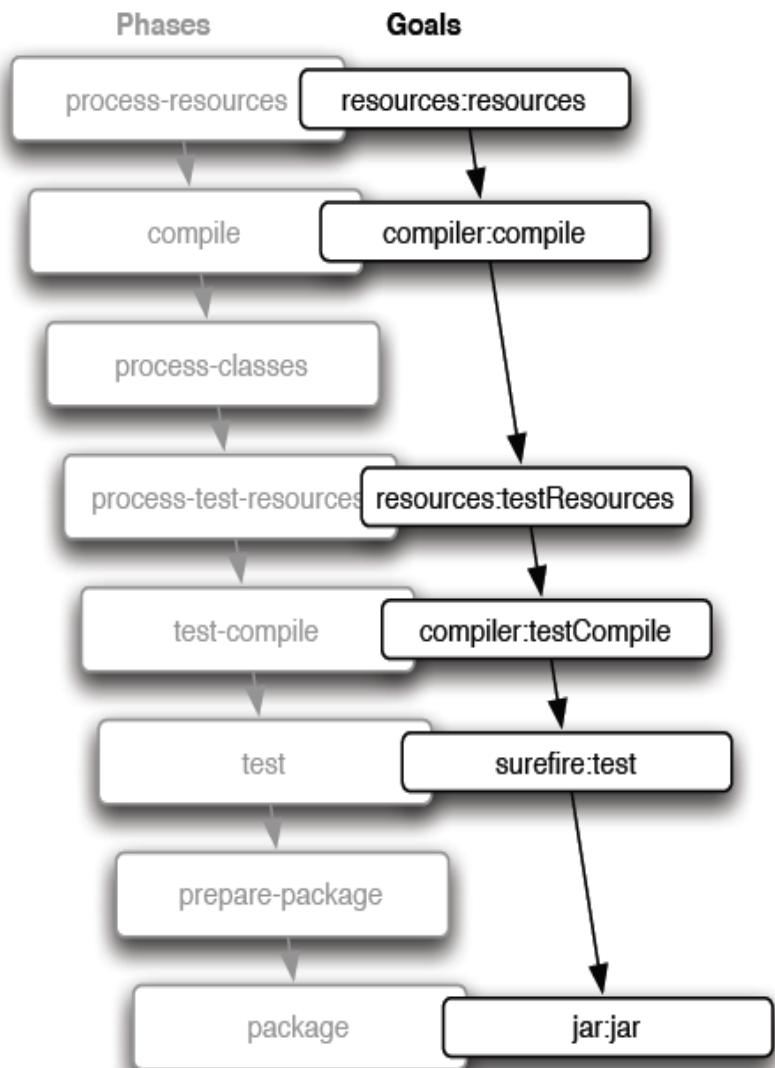
Cycle de vie d'un projet Maven

- Dans le cycle de vie 'par défaut' d'un projet Maven, les phases les plus utilisées sont:

- ❖ **validate**: vérifie les prérequis d'un projet maven
- ❖ **compile**: compilation du code source
- ❖ **test**: lancement des tests unitaires
- ❖ **package**: assemble le code compilé en un livrable
- ❖ **install**: partage le livrable pour d'autres projets sur le même ordinateur
- ❖ **deploy**: publie le livrable pour d'autres projets dans le « repository » distant

- Les phases s'exécutent de façon **séquentielle** de façon à ce qu'une phase dépende de la phase précédente.

- Par exemple, le lancement par l'utilisateur de la phase test (**mvn test**) impliquera le lancement préalable par maven des phases « validate » et « compile ».



Commandes Maven 3

- Toutes les fonctionnalités décrites font partie de la version **3** de Maven.
- Une commande Maven3 s'utilise en ligne de commande sous la forme suivante :

mvn plugin:goal ou **mvn plugin**

- Exemple:

mvn archetype:generate

- Il est possible d'utiliser des options précédées par « - »

- Exemple:

mvn -version

Exemples de commandes Maven 3

commande	Description
<code>mvn package</code>	Construire le projet pour générer l'artefact
<code>mvn site</code>	Générer le site de documentation dans le répertoire target/site
<code>mvn clean</code>	Supprimer les fichiers générés par les précédentes générations
<code>mvn install</code>	Générer l'artefact et le déployer dans le dépôt local
<code>mvn eclipse:eclipse</code>	Générer des fichiers de configuration Eclipse à partir du POM (notamment les dépendances)
<code>mvn javadoc:javadoc</code>	Générer la Javadoc
<code>mvn test</code>	Exécuter les tests unitaires