Maven : les concepts clés

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-11-02

Table des matières

1 Avant de commencer 1
2 Objectifs 1
3 1. Le POM décrit le projet 2
4 2. Les coordonnées dictent l'organisation du dépôt local 2
5 3. Cycle de vie : les phases sont ordonnées 3
6 4. Phases, goals et plugins 3
7 5. Le scope répond à « quand cette dépendance est-elle nécessaire ? » 4
8 6. Une dépendance en apporte d'autres 5
9 7. Héritage et agrégation ne sont pas la même relation 5
10 8. Le wrapper fige la version de Maven dans le dépôt 6
11 9. Lire un BUILD FAILURE 6
12 À retenir 7
13 Auto-évaluation 7
13.1 Éléments de réponse 8
14 Étape suivante 8

1 Avant de commencer

Avant Maven, construire un projet Java signifiait écrire soi-même un script de compilation, maintenir à la main la liste des bibliothèques du classpath et rejouer l'opération sur chaque machine. Le résultat est fragile : il dépend de la mémoire du scripteur et de l'état de son poste de travail.

Maven change de contrat. Au lieu de décrire comment construire, on décrit ce qu'est le projet : ses coordonnées, ses dépendances, sa structure. Maven en déduit une procédure de construction standard. Cette leçon pose le vocabulaire et la logique sur lesquels le TP s'appuie ; elle n'énumère pas de commandes.

2 Objectifs

À l'issue de cette leçon, vous devriez être capable de :

- expliquer ce qu'un POM déclaratif remplace par rapport à un script manuel ;
- identifier les coordonnées d'un artefact et en déduire son emplacement dans le dépôt local ;
- situer une invocation dans un cycle de vie, et distinguer phase, goal et plugin ;
- choisir un scope de dépendance selon le moment où la bibliothèque est nécessaire ;
- prévoir l'effet d'une dépendance transitive et la rendre lisible ;
- distinguer héritage (`<parent>`) et agrégation (`<modules>`) ;
- lire un `BUILD FAILURE` et isoler la cause utile.

3 1. Le POM décrit le projet

Le fichier `pom.xml`, appelé **POM** (Project Object Model), est la seule configuration obligatoire d'un projet Maven. Il répond à trois questions : qui êtes-vous, de quoi avez-vous besoin, comment devez-vous être construit.

```
<project xmlns="http://maven.apache.org/POM/4.0.0">
  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.example</groupId>
  <artifactId>billing</artifactId>
  <version>1.0.0</version>
  <packaging>jar</packaging>
</project>
```

Le triplet `groupId` / `artifactId` / `version` forme les **coordonnées** du projet : il l'identifie de façon unique aux yeux de Maven, comme une adresse à trois niveaux. Une quatrième brique, `packaging` (`jar`, `war`, `pom`), précise la forme de l'artefact produit. Pour un projet Java standard, `<packaging>jar</packaging>` est la forme attendue ; `jar` étant la valeur par défaut, l'élément peut être omis sans changer le résultat — on le rend explicite quand il faut signaler une autre nature d'artefact.

4 2. Les coordonnées dictent l'organisation du dépôt local

Les dépendances ne vivent pas dans le projet. Elles sont téléchargées puis rangées dans le **dépôt local**, par défaut `~/.m2/repository`, dont l'arborescence reproduit exactement les coordonnées :

```
~/.m2/repository/
└── fr/example/billing/1.0.0/
    ├── billing-1.0.0.jar
    └── billing-1.0.0.pom
```

Trois conséquences pratiques :

1. une machine neuve doit tout télécharger : la première construction est lente, la seconde quasi instantanée ;
2. le dépôt local est un cache, jamais une source : on peut le vider sans rien perdre, et il n'est jamais versionné ;
3. une dépendance « introuvable » est d'abord une question de coordonnées.

5 3. Cycle de vie : les phases sont ordonnées

Maven définit plusieurs cycles de vie indépendants (`clean`, `default`, `site`). Une **phase** est une étape nommée et ordonnée : invoquer une phase exécute implicitement toutes celles qui la précèdent.

```
mvn package        # exécute validate, compile, test, puis package
```

Phase	Rôle	Ce qui est produit
`validate`	vérifier la cohérence du POM	rien
`compile`	compiler `src/main/java`	`target/classes`
`test`	exécuter les tests de `src/test/java`	rapports de test
`package`	assembler l'artefact	`target/*.jar`
`verify`	contrôles complémentaires sur l'artefact	rien de publié
`install`	publier l'artefact dans le dépôt local	entrée dans `~/.m2`
`deploy`	publier l'artefact dans un dépôt distant	artefact partagé

> ⚠ Piège classique
>
> `clean` **n'appartient pas** au cycle `default`. Écrire `mvn clean package` ne place pas une phase avant les autres : cela invoque deux cycles distincts dans l'ordre donné sur la ligne de commande.

6 4. Phases, goals et plugins

Une phase n'accomplit rien par elle-même : elle est reliée à des **goals** (objectifs) apportés par des **plugins**. Formule à retenir : le cycle ordonne, le plugin fournit, le goal exécute.

```
mvn compiler:compile     # goal 'compile' du plugin maven-compiler-plugin
mvn dependency:tree      # goal 'tree' : affiche l'arbre des dépendances
```

Certains goals, comme `dependency:tree`, sont des outils de diagnostic invocables hors cycle. La configuration d'un plugin, elle, se déclare dans le POM et non sur la ligne de commande, sans quoi elle n'existerait que pour la frappe du jour :

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <release>21</release>   <!— version de Java cible du projet —>
      </configuration>
    </plugin>
  </plugins>
</build>
```

7 5. Le scope répond à « quand cette dépendance est-elle nécessaire ? »

Le **scope** d'une dépendance borne les étapes où elle apparaît dans le classpath, et décide de sa propagation transitive. Pour un `jar` Maven standard, l'artefact final n'embarque pas les dépendances ; l'emballage dans une archive exécutable dépend d'un autre mécanisme (`war`, `shade`, `assembly`).

Scope	Compilation	Tests	Exécution	Transitif
`compile` (défaut)	oui	oui	oui	oui
`provided`	oui	oui	non	non
`runtime`	non	oui	oui	oui
`test`	non	oui	non	non

- `provided` : l'environnement d'exécution fournit la bibliothèque ; elle sert à compiler, elle ne doit pas être fournie à l'exécution ni empaquetée dans un artefact d'exécution sous peine de doublon.
- `runtime` : inutile pour compiler, indispensable pour exécuter — un pilote JDBC que le code ne nomme jamais en est l'exemple type.
- `test` : strictement confinée aux tests.

Deux scopes supplémentaires existent (`system`, déprécié ; `import`, réservé au `<dependencyManagement>`) et ne sont pas mobilisés ici.

8 6. Une dépendance en apporte d'autres

En déclarant une dépendance, vous héritez aussi de ses propres dépendances : ce sont les **dépendances transitives**. Le scope se propage : une dépendance `compile` expose ses dépendances `compile` et `runtime` en `compile` ; une dépendance `test` d'un tiers reste confinée à ce tiers.

```
billing
└── fr.example:common:1.2.0          (compile)
    └── org.slf4j:slf4j-api:2.0.9    (compile)
```

```
mvn dependency:tree -Dverbose      # révèle l'arbre et les arbitrages de version
```

Bénéfice : ne déclarer que ce dont le code se sert directement. Contrepartie : des versions que vous n'avez pas choisies peuvent être sélectionnées à votre place. Cette lecture est le premier réflexe lorsqu'un `NoSuchMethodError` apparaît à l'exécution alors que la compilation passait.

9 7. Héritage et agrégation ne sont pas la même relation

Critère	Héritage	Agrégation
Déclaration	`<parent>` dans l'enfant	`<modules>` dans le père
Effet	reprend propriétés, versions et configurations	construit plusieurs projets d'un seul appel (le reactor)
Implique une arborescence ?	non	oui : chaque module est un chemin déclaré

```
<!-- côté enfant : je reprends une politique de construction -->
<parent>
  <groupId>fr.example</groupId>
  <artifactId>platform-parent</artifactId>
  <version>3.1.0</version>
</parent>
```

```
<!-- côté père : je suis responsable de ces modules -->
<modules>
  <module>common</module>
  <module>billing</module>
</modules>
```

Les deux relations se logent souvent dans le même fichier, d'où la confusion courante. Règle de conduite : le parent porte la politique (versions, encodage, options de compilation), chaque module garde la description de ce qu'il est. Un POM purement agrégateur se déclare en général `<packaging>pom</packaging>`.

10 8. Le wrapper fige la version de Maven dans le dépôt

`mvn` dépend d'une installation locale : deux machines, deux versions, deux comportements possibles. Le **wrapper** — `mvnw`, `mvnw.cmd` et le répertoire `.mvn/wrapper/` — est un lanceur versionné qui télécharge et exécute la version de Maven inscrite dans le dépôt, et non celle laissée sur le poste.

```
./mvnw package       # même version de Maven pour tout le monde
```

Dans un projet versionné qui fournit un wrapper, les consignes doivent mentionner `./mvnw` plutôt que `mvn`. Dans un TP où le projet est créé à la main et où le wrapper n'existe pas encore, les commandes peuvent rester écrites avec `mvn` ; générer le wrapper ensuite est une amélioration facultative. Le wrapper fige l'outil ; il ne dispense en rien de comprendre ce que la commande fait.

11 9. Lire un BUILD FAILURE

Un échec Maven est bavard, mais structuré :

```
[INFO] BUILD FAILURE
[ERROR]     Failed     to     execute     goal     o.a.m.plugins:maven-compiler-
plugin:3.11.0:compile
        (default-compile) on project billing: Compilation failure
[ERROR] /home/you/dev/billing/src/main/java/App.java:[12,9] cannot find symbol
[ERROR]   symbol: class Price
```

Quatre questions, dans cet ordre :

1. Une **goal** est-elle nommée ? Si oui, elle apparaît en tête de la première ligne (`Failed to execute goal …`) et localise l'étape en cause. Si aucune goal n'est citée, l'échec est antérieur à l'exécution d'une tâche : il relève en général de la **résolution** — dépendances ou plugins introuvables — et c'est cette piste qu'il faut suivre d'emblée.
2. Quel **module** est en cause ? Réponse dans `on project …`, lorsqu'un projet est identifié.
3. La **première** ligne `[ERROR]` porte la cause ; les suivantes sont souvent des conséquences. On ne lit qu'elle.

4. S'agit-il d'un problème de **résolution** ou de **compilation / test** ? Dans le premier cas, l'enquête porte sur les coordonnées et les dépôts ; dans le second, sur le code.

> **Erreur fréquente**
>
> `Could not resolve dependencies` ne signifie pas « cette bibliothèque n'existe pas », mais « Maven ne l'a pas trouvée **aux coordonnées indiquées**, dans **les dépôts qu'il connaît** ». Relire la déclaration dans le POM est plus efficace que de vider le dépôt local, qui ne fait que masquer le problème et le faire rejouer. Pour trancher sans lancer une construction complète, `dependency:resolve` vérifie que chaque dépendance déclarée est bien atteignable ; `dependency:tree` en affiche le sous-arbre.

12 À retenir

> **Sept idées à garder**
>
> 1. Le POM décrit, Maven déduit.
> 2. Coordonnées = `groupId` : `artifactId` : `version` — et l'arborescence du dépôt local en est la conséquence mécanique.
> 3. Une phase entraîne toutes les phases précédentes ; `clean` est un autre cycle.
> 4. Un scope se choisit en demandant quand la bibliothèque est nécessaire.
> 5. Toute dépendance apporte son propre sous-arbre.
> 6. `<parent>` est une relation d'héritage, `<modules>` une relation d'agrégation.
> 7. Dans un échec, le premier `[ERROR]` est celui qui compte.

13 Auto-évaluation

Répondez en une ou deux phrases avant d'ouvrir le TP.

1. Qu'est-ce qu'un POM déclaratif remplace, par rapport à une construction manuelle ?
2. Où Maven range-t-il l'artefact `fr.example:billing:1.0.0` dans le dépôt local ?
3. `mvn test` compile-t-il les sources principales ? Justifiez.
4. Vous avez besoin de JUnit uniquement depuis `src/test/java` : quel scope ?
5. Un pilote JDBC est requis à l'exécution mais jamais nommé dans le code : quel scope ?
6. Une dépendance déclarée en `test` contamine-t-elle l'artefact final ?
7. Un POM père déclare `<modules>` et chaque enfant déclare `<parent>` : qui relève de l'agrégation, qui relève de l'héritage ?

8. `mvn clean package` échoue sur `Could not resolve dependencies` : citez deux causes distinctes à tester dans l'ordre.

13.1 Éléments de réponse

1. Le script de construction et la gestion manuelle du classpath.
2. `~/.m2/repository/fr/example/billing/1.0.0/billing-1.0.0.jar`, avec son `.pom`.
3. Oui : `test` est postérieure à `compile`, qui est donc exécutée par la même invocation.
4. `test`.
5. `runtime`.
6. Non : une dépendance en scope `test` n'est ni transitive, ni embarquée dans un `jar` standard.
7. `<modules>` agrège, `<parent>` hérite.
8. D'abord les coordonnées (artefact, groupe ou version mal écrits, version inexistante) ; ensuite le dépôt censé héberger l'artefact — un échec de téléchargement mis en cache se relance avec `-U`.

14 Étape suivante

Le TP 01_java_P_maven_lab.qmd met ces notions à l'épreuve sur un projet que vous manipulez vous-même : ses consignes fixent le détail des commandes, la présente leçon en fournit la logique. Revenez y chercher une définition au premier message d'erreur obscur — la cause est le plus souvent une coordonnée, un scope ou une phase mal interprétée.