TP — Manipuler un projet Maven

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

2026-11-09

Table des matières

1 Prérequis 2
2 Objectifs 2
3 Étape 0 — Préparation du poste 2
4 Étape 1 — Un projet sans générateur 3
5 Étape 2 — Coordonnées et dépôt local 4
6 Étape 3 — Phases et répertoire `target/` 4
7 Étape 4 — Ajouter une dépendance, choisir un scope 5
8 Étape 5 — L'arbre des dépendances 7
9 Étape 6 — Provoquer et lire un BUILD FAILURE 8
10 Étape 7 — Mini multi-module : héritage et agrégation 9
11 Livrables 11
12 Erreurs fréquentes 11
13 Pour aller plus loin (facultatif) 12
14 Où chercher une définition 12

> **i** Mode d'emploi
>
> Rien n'est exécuté par ce document : tous les blocs sont des exemples à recopier dans votre terminal. Dans un bloc `bash`, le `$` introduit la commande (ne le tapez pas) ; dans un bloc `text`, il s'agit d'une sortie **attendue**, à comparer avec la vôtre — les libellés exacts varient avec la version de Maven.
>
> Ce TP **applique** la leçon 01_java_L_maven_introduction.qmd : il n'en redonne pas les définitions. Ouvrez-la à côté et allez y chercher la notion dès qu'un message d'erreur vous arrête.

> **mvn ou ./mvnw ?**
>
> La leçon (§ 8) recommande le wrapper `./mvnw` dans un vrai projet versionné. Ici, vous créez le projet à la main et le wrapper n'existe pas au départ : **toutes les commandes du TP sont donc écrites avec `mvn`**. Quand votre POM sera en place, générer le wrapper (`mvn wrapper:wrapper`) puis remplacer `mvn` par `./mvnw` partout est un excellent exercice facultatif. Générer le wrapper n'empêche pas `mvn` de continuer à fonctionner ; faites cet exercice seulement une fois le TP terminé, sinon les commandes recopiées ne correspondront plus à celles du TP.

1 Prérequis

- Un JDK 21 ou supérieur (`java -version`) et Maven 3.9+ installés.
- Un terminal et un éditeur de texte. Aucun IDE requis.
- De la place disque : les téléchargements vont dans `~/.m2/repository`.

> **Sur les versions citées**
>
> Les versions d'artefacts de ce TP (`org.slf4j:slf4j-api`, `org.junit.jupiter:junit-jupiter`, `maven-surefire-plugin`) sont **indicatives**. Si une résolution échoue sur une version, remplacez-la par une version publiée : la manipulation et les conclusions ne changent pas.

2 Objectifs

À l'issue du TP, vous savez :

1. écrire à la main un POM minimal et en lire les coordonnées ;
2. retrouver un artefact dans le dépôt local à partir de ces coordonnées ;
3. prévoir, puis vérifier, ce qu'une phase du cycle de vie dépose dans `target/` ;
4. choisir un scope de dépendance en justifiant par l'usage, et le vérifier ;
5. provoquer un `BUILD FAILURE` de résolution et en isoler la cause utile ;
6. distinguer héritage (`<parent>`) et agrégation (`<modules>`) dans un vrai projet ;
7. rendre le projet reproductible sur une autre machine.

3 Étape 0 — Préparation du poste

```
$ mvn -v
Apache Maven 3.9.x
Maven home: /opt/maven
Java version: 21.x, vendor: ..., runtime: /Library/Java/...
```

Les chemins `Maven home` et Java runtime varient selon l'installation et le système d'exploitation ; comparez les versions, pas les chemins.

`mvn -v` affiche la version de Maven et celle du JDK utilisée : ce sont les deux variables d'environnement qui rendent deux machines dissemblables. Cette commande n'affiche **pas** l'emplacement du dépôt local : par défaut, c'est `~/.m2/repository`. Regardez son contenu :

```
$ ls ~/.m2/repository
```

Si ce répertoire n'existe pas, c'est normal sur une machine neuve : il est créé à la première construction.

- ☐ `mvn -v` répond (et non `command not found`).
- ☐ Vous savez dire où Maven range les artefacts qu'il télécharge.

> ℹ Si le dépôt local a été déplacé
>
> Pour connaître le chemin réellement actif sur votre machine : `mvn help:evaluate -Dexpression=settings.localRepository -q -DforceStdout`. Remplacez alors `~/.m2/repository` par la valeur affichée dans tout le TP.

4 Étape 1 — Un projet sans générateur

Créez l'arborescence suivante à la main (aucune commande de génération, aucun archétype : l'objectif est de voir ce que Maven **attend**, pas de déléguer la frappe) :

```
billing/
├── pom.xml
└── src/
    ├── main/java/fr/example/billing/App.java
    └── test/java/fr/example/billing/
```

`billing/pom.xml` :

```
<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>
```

`src/main/java/fr/example/billing/App.java` :

```java
package fr.example.billing;

public class App {
    public static void main(String[] args) {
        System.out.println("billing prêt");
    }
}
```

Ajoutez un `.gitignore` contenant au minimum `target/` : le dépôt local et le répertoire de construction ne se versionnent jamais.

Questions. (a) Quelles lignes du POM sont des coordonnées, laquelle est la forme de l'artefact ? (b) Que perdriez-vous si vous supprimiez `<packaging>` ? (c) Quel répertoire Maven va-t-il compiler sans qu'on le lui dise ?

- ☐ `mvn validate` se termine par `BUILD SUCCESS`.
- ☐ Le projet est initialisé en Git et ne contient aucun `target/` versionné.

5 Étape 2 — Coordonnées et dépôt local

Prédisez d'abord, avant toute commande, le chemin complet que `mvn install` va faire apparaître dans le dépôt local pour votre propre artefact `fr.example:billing:1.0.0`. Écrivez-le, puis installez et vérifiez :

```
$ mvn install
$ ls ~/.m2/repository/fr/example/billing/1.0.0/
billing-1.0.0.jar  billing-1.0.0.pom  ...
```

Le chemin parcouru est exactement le `groupId` découpé en sous-répertoires, puis l'`artifactId`, puis la `version`.

1. Écrivez de mémoire ce même chemin pour `fr.example:common:1.0.0`. Cet artefact n'existe pas encore : gardez votre prédiction, vous la vérifierez à l'étape 7.
2. Expliquez, en deux phrases, pourquoi la **seconde** exécution de `mvn compile` est quasi instantanée alors que la première a téléchargé.

- ☐ Votre prédiction pour `billing` correspond au `ls` observé.
- ☐ Vous savez dire pourquoi vider `~/.m2` ne casse rien définitivement.

6 Étape 3 — Phases et répertoire `target/`

Prévoyez **avant** de lancer chaque commande ce qui va apparaître dans `target/`, puis vérifiez. Complétez le tableau :

Commande	Phases réellement exécutées	Nouveautés observées dans `target/`
`mvn compile`		
`mvn test`		
`mvn package`		
`mvn clean package`		

Regardez le contenu de l'artefact produit :

```
$ jar tf target/billing-1.0.0.jar
META-INF/MANIFEST.MF
fr/example/billing/App.class
```

Questions. (a) `mvn test` a-t-il compilé `src/main/java` ? Comment le voyez-vous ? (b) Pourquoi `clean` apparaît-il « avant » `package` alors qu'il n'appartient pas au même cycle ? (c) Pour l'instant, aucune dépendance n'est déclarée. Prédisez ce que contiendra ce jar **après** l'étape 4 : les classes de `slf4j-api` ou de JUnit y figureront-elles ? Vous vérifierez à la fin de l'étape 4.

- ☐ Vous justifiez chaque ligne du tableau par une observation, pas par une opinion.
- ☐ Vous distinguez phase invoquée et phases entraînées.

7 Étape 4 — Ajouter une dépendance, choisir un scope

Deux besoins, deux moments différents : `slf4j` est utilisé par le code de production, JUnit uniquement par les tests.

```
  <dependencies>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>2.0.9</version>
    </dependency>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter</artifactId>
      <version>5.10.0</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

Utilisez-les vraiment. Dans `App.java` :

```java
package fr.example.billing;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class App {
    private static final Logger LOG = LoggerFactory.getLogger(App.class);

    public static void main(String[] args) {
        LOG.info("billing prêt");
    }
}
```

Dans `src/test/java/fr/example/billing/SmokeTest.java` :

```java
package fr.example.billing;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

class SmokeTest {
    @Test
    void journalisable() {
        Assertions.assertNotNull(App.class);
    }
}
```

Pour rendre l'exécution des tests reproductible, épinglez la version du `maven-surefire-plugin` dans le POM plutôt que de compter sur la version héritée de votre installation : c'est exactement ce que dit la leçon sur la configuration des plugins. Selon la version de Maven utilisée, les tests peuvent passer sans cette précision ; l'épinglage évite surtout qu'ils dépendent d'un comportement local.

```xml
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>3.2.5</version>
      </plugin>
    </plugins>
  </build>
```

```
$ mvn clean package
... Tests run: 1, Failures: 0 ... [INFO] BUILD SUCCESS
```

Expérience. Le scope se prouve par l'observation, pas par la mémorisation. Pour chaque valeur de `slf4j-api`, lancez `mvn clean compile`, puis `mvn clean test`. Testez `compile`, `runtime`, puis `provided` ; remettez `compile` à la fin.

Scope de `slf4j-api`	`mvn clean compile` passe	`mvn clean test` passe	Message obtenu
`compile`			
`runtime`			
`provided`			

> ⚠️ Lire les résultats avec Maven
>
> Si `mvn clean test` échoue avec le scope `runtime`, l'échec a lieu pendant la phase `compile`, avant la phase `test`. Le tableau de la leçon indique que `runtime` est disponible pour les tests une fois cette phase atteinte, pas que la commande complète `mvn clean test` réussit quand le code source principal utilise la dépendance.

Après avoir remis `slf4j-api` en `compile`, régénérez l'artefact :

```
$ mvn clean package
$ jar tf target/billing-1.0.0.jar
```

Aucune classe de dépendance n'y apparaît : un `jar` Maven standard n'embarque pas ses dépendances. Un scope ne se lit donc pas dans le contenu du `jar` ; il se lit dans les phases atteintes ou bloquées par les commandes `mvn clean compile` et `mvn clean test`.

- ☐ Vous expliquez l'échec obtenu en `runtime` par la classe présente dans le code.
- ☐ Vous rattachez chaque observation du tableau à une colonne du tableau de cours.

8 Étape 5 — L'arbre des dépendances

```
$ mvn dependency:tree
[INFO] fr.example:billing:jar:1.0.0
[INFO] +- org.slf4j:slf4j-api:jar:2.0.9:compile
[INFO] \- org.junit.jupiter:junit-jupiter:jar:5.10.0:test
[INFO]    +- org.junit.jupiter:junit-jupiter-api:jar:...:test
```

```
[INFO]    +- org.junit.jupiter:junit-jupiter-params:jar:...:test
[INFO]    \- org.junit.jupiter:junit-jupiter-engine:jar:...:test
```

Questions. (a) Avez-vous déclaré `junit-jupiter-api` ? Pourquoi est-il là ? (b) Quel scope se propage aux enfants de `junit-jupiter` ? (c) Combien de lignes de dépendances auriez-vous écrites sans la transitivité ?

- ☐ Vous lisez une ligne d'arbre au format `groupe:artefact:packaging:version:scope`.
- ☐ Vous savez nommer un cas où cette lecture est votre premier réflexe.

9 Étape 6 — Provoquer et lire un BUILD FAILURE

Déclarez une dépendance volontairement fausse, puis construisez :

```
    <dependency>
      <groupId>fr.example</groupId>
      <artifactId>introuvable</artifactId>
      <version>9.9.9</version>
    </dependency>
```

```
$ mvn compile
```

Sortie typique (condensée) :

```
[INFO] BUILD FAILURE
[ERROR] Failed to execute goal on project billing: Could not resolve dependencies
        for project fr.example:billing:jar:1.0.0
[ERROR] dependency: fr.example:introuvable:jar:9.9.9 (compile)
[ERROR]   fr.example:introuvable:jar:9.9.9 was not found ... this failure was
        cached in the local repository
```

Appliquez les questions de la leçon, par écrit. Observez un détail : ici, la ligne d'en-tête dit `Failed to execute goal on project billing` **sans nommer de goal** (`compiler:compile` n'apparaît pas). Question 1 des quatre questions du § 9 de la leçon : à quoi ce détail vous dit-il que le problème est de **résolution** et non de compilation ? Répondez aussi : quel projet est en cause, quelle est la **première** ligne `[ERROR]`, que faut-il relire en priorité (le code ou le POM) ?

Deux investigations complémentaires :

```
$ ls ~/.m2/repository/fr/example/introuvable/9.9.9/
introuvable-9.9.9.pom.lastUpdated      # l'échec est mis en cache
```

```
$ mvn -U compile                          # -U : relance la résolution en ignorant
le cache
```

Le build échoue encore : l'artefact n'existe vraiment pas. Ce qui change est que Maven retente l'accès réseau au lieu de rejouer l'échec mis en cache.

Variante — isoler l'axe « version » de l'axe « dépôt ».

1. Gardez la dépendance `fr.example:introuvable` et changez seulement sa version en `99.0.0`. Relancez `mvn compile` : l'erreur change-t-elle de nature ?
2. Retirez la dépendance `fr.example:introuvable`, puis changez la version de `org.slf4j:slf4j-api` de `2.0.9` à `99.0.0`. Relancez : même famille d'erreur, alors que le groupe et l'artefact, eux, existent bien.
3. Remettez `org.slf4j:slf4j-api` en `2.0.9` et retrouvez un `BUILD SUCCESS`.

- ☐ Vous écrivez en une phrase ce que `Could not resolve dependencies` veut dire.
- ☐ Vous savez pourquoi vider `~/.m2` ne serait pas ici la bonne première réponse.

10 Étape 7 — Mini multi-module : héritage et agrégation

Rangez deux projets sous un même parent. Arborescence cible :

```
platform/
├── pom.xml                  # agrégateur ET parent
├── common/pom.xml
└── billing/                 # le projet des étapes 1 à 6, déplacé tel quel
```

Placez votre dépôt Git et le `.gitignore` à la racine `platform/` : si `billing/` contient déjà un `.git` ou un `.gitignore`, déplacez-les à la racine `platform/` ou réinitialisez le dépôt à ce niveau.

`platform/pom.xml` :

```
<project xmlns="http://maven.apache.org/POM/4.0.0">
  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.example</groupId>
  <artifactId>platform-parent</artifactId>
  <version>1.0.0</version>
  <packaging>pom</packaging>
  <modules>
    <module>common</module>
    <module>billing</module>
  </modules>
  <properties>
    <maven.compiler.release>21</maven.compiler.release>
```

```
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>
</project>
```

`platform/common/pom.xml` et sa classe `fr.example.common.Money` :

```
<project xmlns="http://maven.apache.org/POM/4.0.0">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>fr.example</groupId>
    <artifactId>platform-parent</artifactId>
    <version>1.0.0</version>
  </parent>
  <artifactId>common</artifactId>
</project>
```

```
package fr.example.common;

public final class Money {
    private final long cents;
    public Money(long cents) { this.cents = cents; }
    public long cents() { return cents; }
}
```

Modifiez `billing/pom.xml` : ajoutez en tête le **même** bloc `<parent>` que `common`, retirez les lignes `<groupId>` et `<version>` désormais héritées, et conservez `<artifactId>billing</artifactId>` ainsi que `<packaging>jar</packaging>`. Déclarez ensuite le besoin sur `common` :

```
    <dependency>
      <groupId>fr.example</groupId>
      <artifactId>common</artifactId>
      <version>1.0.0</version>
    </dependency>
```

Depuis la **racine** `platform/` :

```
$ mvn clean install
[INFO] Reactor Summary for platform-parent 1.0.0:
[INFO] platform-parent ..................................... SUCCESS
[INFO] common .............................................. SUCCESS
```

```
[INFO] billing ..................................... SUCCESS
[INFO] BUILD SUCCESS
```

Vérifiez ici la prédiction écrite à l'étape 2 pour `fr.example:common:1.0.0` :

```
$ ls ~/.m2/repository/fr/example/common/1.0.0/
```

Questions. (a) Qu'avez-vous retiré du POM de `billing`, et qui le fournit désormais ? (b) Quelles propriétés du parent remplacent ce que vous auriez pu écrire dans chaque enfant ? (c) Dans quel ordre le reactor a-t-il construit les modules et qu'est-ce qui dicte cet ordre ? (d) Supprimez `~/.m2/repository/fr/example/common`, puis lancez `mvn package` **depuis `billing/` seul** : que constatez-vous et pourquoi ? (e) Relancez depuis `platform/` : le problème a disparu, pourquoi ?

- ☐ Vous classez chaque déclaration rencontrée : héritage ou agrégation.
- ☐ `mvn dependency:tree` dans `billing` montre `common` et son scope.

11 Livrables

À déposer :

1. l'arborescence complète de `platform/`, **sans** `target/` ni dépôt local ;
2. le tableau de l'étape 3 et celui de l'étape 4 complétés ;
3. les réponses rédigées des questions des étapes 5, 6 et 7 (une ou deux phrases chacune) ;
4. trois copies de console : `mvn -v`, un `dependency:tree`, un `BUILD FAILURE` de résolution annoté (projet, première ligne `[ERROR]`, indice « absence de goal ») ;
5. un `compte-rendu.md` de dix lignes maximum : ce que vous prévoyiez, ce que vous avez observé, l'écart le plus surprenant.

12 Erreurs fréquentes

Symptôme	Cause probable	Réflexe
`mvn: command not found`	Maven non installé ou absent du `PATH`	installer Maven 3.9+, vérifier le `PATH`
`No compiler is provided in this environment`	JDK non installé, `JAVA_HOME` sur un JRE	`java -version`, corriger `JAVA_HOME`
`cannot find symbol: class Logger`	dépendance sortie du classpath de compilation	vérifier le scope, pas le code

Symptôme	Cause probable	Réflexe
`Could not resolve dependencies`	coordonnée ou version incorrecte, échec mis en cache	relire le POM, puis `-U`
`Tests are skipped` / aucun test exécuté	surefire hérité non épinglé ou non compatible avec JUnit 5	épingler la version du plugin
`common` introuvable depuis `billing` seul	jamais `install` dans le dépôt local	relancer depuis la racine agrégée

13 Pour aller plus loin (facultatif)

- Comparez votre POM à `mvn help:effective-pom` : vous verrez tout ce que Maven ajoute sans que vous l'écriviez. C'est le prix du déclaratif.
- Générez le wrapper dans `billing/` (`mvn wrapper:wrapper`), versionnez `mvnw`, `mvnw.cmd` et `.mvn/`, puis reprenez le TP avec `./mvnw` : vous appliquerez le § 8 de la leçon.
- Sans rien exécuter d'autre, prédisez puis vérifiez l'effet de `mvn verify` et de `mvn install` sur `target/` et sur `~/.m2`.

14 Où chercher une définition

Ce TP ne définit rien : les définitions, les tableaux de scopes et la méthode de lecture d'un échec sont dans 01_java_L_maven_introduction.qmd. Si une consigne vous paraît arbitraire, c'est presque toujours qu'une notion de la leçon (coordonnée, phase, goal, scope, héritage) a été relue trop vite.