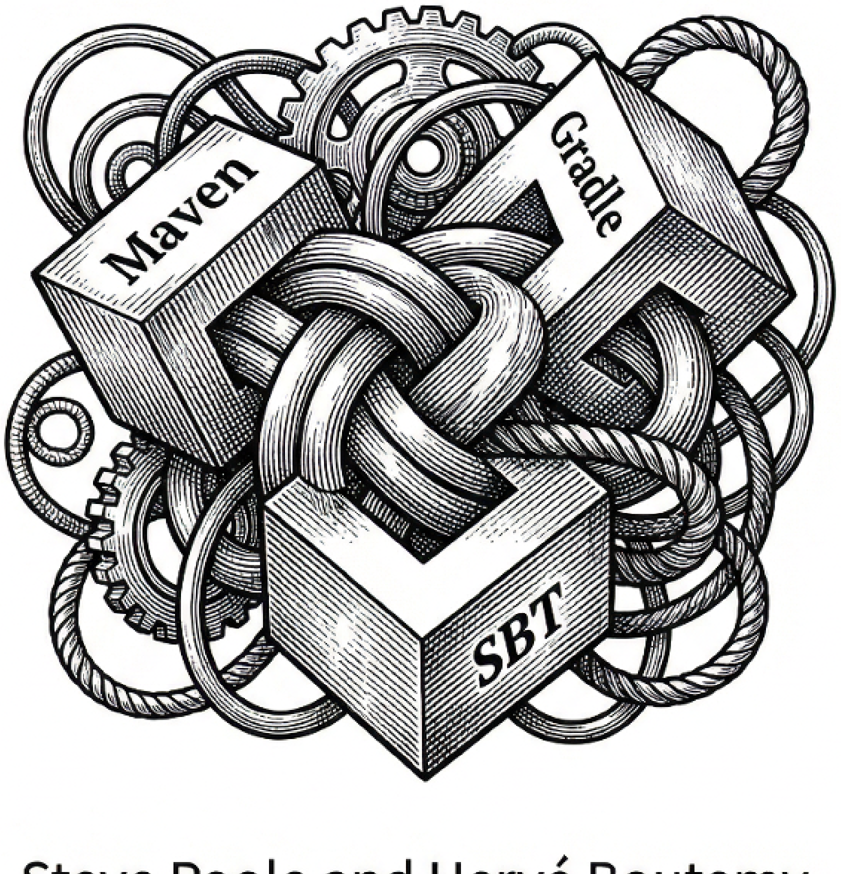
Practical Dependency Resolution

A Field Guide



Steve Poole and Hervé Boutemy

Sponsored By herodevs

Welcome To the sample of the Dependency Management Troubleshooting Guide

This sample is rough and ready. We reserve every right to correct any mistakes This QR code takes you to the HeroDevs website where you get the latest version of this sample and register for the complete book when available.

Figure 1.: QR Code

Who This Book Is For Dependency management often behaves as you would guess enough times to earn your trust. Until it doesn't.

Most trouble stems not from misunderstanding Maven or Gradle syntax, but from making assumptions that conflict with how build tools actually operate.

This book is about the mechanical rules that govern those operations.

If you have stared at a dependency tree and thought, "that doesn't make sense," this book is for you.

It is written for

- **Experienced Developers:** You know how to use exclusions and BOMs, but have learned that today's fix can become tomorrow's mystery.
- **Tech Leads & Build Owners:** You are responsible for build stability and must explain why a version changed or why "just upgrade it" isn't a strategy.
- **Platform & Security Engineers:** You recognize these failure modes as consequences of real-world dependency resolution, not just devel-

oper mistakes.
- **Junior Developers:** This guide will help you avoid persistent misconceptions early in your career.

How to Use This Guide This guide is designed for reactive use. When a version changes unexpectedly or a class disappears at runtime, scan the scenario titles. Each scenario provides rapid orientation: expected intuition, actual results, the underlying mechanism, and safe resolutions.

Alternatively, read several scenarios to understand the system's shape. You will notice recurring patterns—these represent the core mechanics of dependency resolution.

This is not a reference manual or a replacement for official documentation. It does not provide exhaustive command sequences.

Instead, it explains the "why." Dependency resolution is not intuitive or opinionated; it is mechanical, structural, and remarkably consistent. Once you understand the rules, the behavior becomes predictable.

4. Start Here for Troubleshooting Dependency Issues

Has something in your build has started behaving differently? A version changed, a class disappeared, or the runtime no longer matches what you expected?

You're in the right place. Don't read this book front to back.

Treat it like a field guide: find the symptom that looks closest to your situation and jump straight to that scenario.

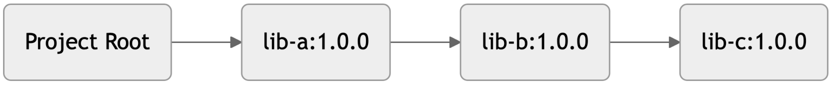
4.0.1. What to do next

- Version changed unexpectedly ⟶ S2: Version Conflict Resolution
- Explicit version not honoured ⟶ S2, then S6
- Compiles but fails at runtime ⟶ S7: Dependency Scopes, also check S5
- Build worked yesterday but not today ⟶ S9: Version Ranges, then S11
- Unwanted library keeps appearing ⟶ S5, then S15
- Fixed a version but it keeps reverting ⟶ S6, and S14
- Dependency tree looks wrong but not sure why ⟶ S1, then S3

Table of contents

1. Basic Transitive Resolution

What to do when a Dependency Brings in Versions You Didn't Expect:
This scenario demonstrates the fundamental concept of transitive dependency resolution.
It shows how a build tool automatically discovers and includes dependencies of your direct dependencies, forming a dependency graph.
This is the "happy path" where no conflicts exist.



```xml
<dependencies>
    <dependency>
        <groupId>com.demo</groupId>
        <artifactId>lib-a</artifactId>
        <version>1.0.0</version>
    </dependency>
</dependencies>
```

Reader Expectation: Declaring `lib-a` automatically includes its transitive dependencies `lib-b` and `lib-c` on the classpath.

Why This Happens: Build tools traverse the dependency graph and collect all unique nodes. In the absence of version conflicts, resolution is straightforward and follows the project's dependency structure.

How Developers Commonly "Fix" This: No fix is needed here. This is the desired behavior.

Safer Ways to Take Control: Even in simple scenarios, using a BOM (Bill of Materials) or dependency constraints can prevent future surprises if new dependencies are added.

Deterministic Checks: **Fast check (Maven):**

```
mvn dependency:tree
```

What to look for: A clean tree with no unexpected versions or `omitted` nodes.

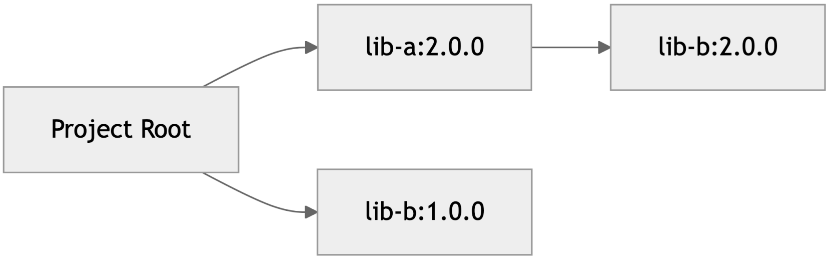
Conclusion: The dependency graph is healthy and resolving as expected.

How This Scales: In large systems, this simple chain `A → B → C` is repeated thousands of times, creating a complex web. The "happy path" becomes rare as diamond dependencies emerge.

Key Takeaway: Transitive dependency resolution is the mechanism that allows us to build on top of giants without manually managing every single library.

2. Version Conflict Resolution

What to Do When the Version You Get Isn't the One You Chose: This scenario demonstrates the "Diamond Dependency" problem where two different versions of the same library are requested.
It highlights the fundamental difference in resolution strategies between Maven ("Nearest Wins") and Gradle/SBT ("Newest Wins").



```
    <dependencies>

        <dependency>
            <groupId>com.demo</groupId>
            <artifactId>lib-a</artifactId>
            <version>2.0.0</version>
        </dependency>
        <dependency>
            <groupId>com.demo</groupId>
            <artifactId>lib-b</artifactId>
            <version>1.0.0</version>
        </dependency>
    </dependencies>
```

Reader Expectation: The newer version (`2.0.0`) wins to provide features for `lib-a`, or the direct dependency (`1.0.0`) wins because it was explicitly requested.

Actual Resolution Results: Resolves `lib-b:1.0.0` due to "Nearest Wins": depth 1 (direct) beats depth 2 (transitive).

Classpath Reality:

- **Maven**: Contains `lib-b-1.0.0.jar`. `lib-a` may crash if it requires API only in `2.0.0`.
- **Gradle/SBT**: Contains `lib-b-2.0.0.jar`. Project code may crash if it requires API removed in `2.0.0`.

Why This Happens: Maven prioritizes predictability; the closest declaration is assumed to be the intentional one. This "nearest wins" strategy is why the same graph behaves differently across tools.

Conflicts at the Same Depth: When depth is equal, Maven uses **declaration order**. The version from the dependency declared first in the `pom.xml` wins. This makes ordering significant and potentially fragile.

How Developers Commonly "Fix" This: Reordering dependencies (fragile), adding explicit exclusions (verbose), or managing versions in a parent POM. * **Gradle**: Forcing versions or using strict constraints.

Operational Risk:

- **Low risk:** Use `<dependencyManagement>` to pin versions explicitly.
- **Medium risk:** Add an explicit direct dependency to shorten distance.
- **High risk:** Reorder `<dependency>` blocks to trigger tie-breaking; this is fragile and often misunderstood.

Safer Ways to Take Control: Use `<dependencyManagement>` to fix the version regardless of depth. Gradle uses `constraints` or `platform` (BOM) for similar control.

Deterministic Checks: **Fast check (Maven):** `mvn dependency:tree -Dverbose` **Fast check (Gradle):** `./gradlew dependencyInsight --dependency <id>` **Fast check (SBT):** `sbt dependencyTree`

What to look for: Lines containing `omitted for conflict` (Maven), `(by conflict resolution)` (Gradle), or evicted nodes (SBT).

Conclusion: The winning version is "nearest" to the root, not necessarily the newest or most compatible.

How This Scales: This is the primary cause of "Dependency Hell." As graphs grow, "nearest" becomes arbitrary and "newest" can trigger unintended upgrades.

Key Takeaway: Maven optimizes for "distance", while Gradle and SBT optimize for "freshness" — knowing which strategy your tool uses is critical for stability.

Related Scenarios:

- Scenario 3: Dependency Management - How to fix this centrally.

Seen in the Wild

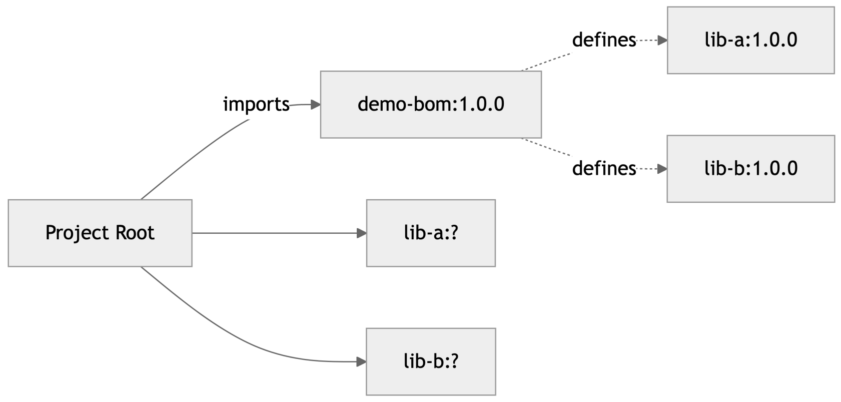
A number of financial institutions discovered vulnerable versions of Apache Commons Collections deep in their transitive graphs, often buried three levels deep in vendor SDKs.

In many cases, they couldn't upgrade the SDK itself. Because Maven's "nearest wins" logic often favored the older, vulnerable version (due to it being slightly "closer" to the root).

They utilized `<dependencyManagement>` to pin the patched version globally. This allowed them to satisfy security audits without extensive regression testing of the vendor SDK.

3. Dependency Management (BOM)

What to Do When Your BOM Isn't Actually in Control: This scenario demonstrates how to use a Bill of Materials (BOM) or `<dependencyManagement>` block to centralize version definitions. This decouples "what I need" from "which version I need".



```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>com.demo</groupId>
            <artifactId>demo-bom</artifactId>
            <version>1.0.0</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <!-- Versions are managed by the BOM -->
    <dependency>
        <groupId>com.demo</groupId>
        <artifactId>lib-a</artifactId>
    </dependency>
    <dependency>
```

```
<!-- Uses managed version (1.0.0) -->
    <dependency>
        <groupId>com.demo</groupId>
        <artifactId>lib-b</artifactId>
    </dependency>
</dependencies>
```

Reader Expectation: Explicitly declared versions win over managed suggestions.

Why This Happens: The `<dependencies>` section is the final authority for the project; `<dependencyManagement>` acts only as a default. This priority is why the same graph behaves differently across tools.

How Developers Commonly "Fix" This: They might try to change the BOM version, which affects everything. Or they use exclusions.

Safer Ways to Take Control: Be careful when overriding BOMs. You might break compatibility with other libraries managed by that BOM. If you must override, ensure you test thoroughly.

Deterministic Checks: **Fast check (Maven):** `mvn dependency:tree`
Fast check (Gradle): `./gradlew dependencyInsight --dependency <id>`
Fast check (SBT): `sbt dependencyTree`

What to look for: The explicitly declared version in the tree, rather than the BOM or platform recommendation.

Conclusion: A hardcoded version in `<dependencies>` always overrides `<dependencyManagement>`.

How This Scales: Overriding BOMs is common when patching a vulnerability before the BOM maintainers release an update.

Key Takeaway: Explicit declarations generally trump managed recommendations, allowing for targeted overrides.

Related Scenarios:

- Scenario 6: Forcing Transitive Versions via Dependency Management - How to force a version even against explicit declarations.

Seen in the Wild

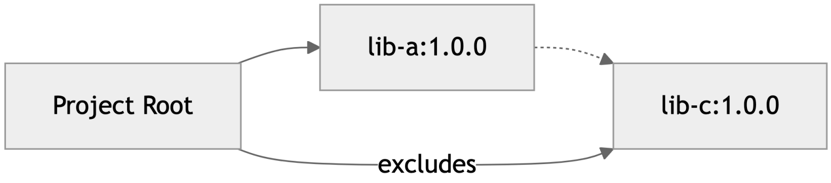
Many teams using the Spring Boot BOM encountered cases where a patched library version was required before an updated BOM was available.

By declaring the patched version explicitly in their `<dependencies>` block, they overrode the BOM's "safe" suggestion.

This allowed them to ship a fix in hours rather than waiting days for the platform maintainers to release a synchronized update.

5. Dependency Exclusions

What to Do When Excluding a Dependency Fixes the Build but Breaks Runtime: This scenario demonstrates how to surgically remove a transitive dependency from the graph.
This is a common "sledgehammer" fix for conflicts.



```
<dependencies>
    <dependency>
        <groupId>com.demo</groupId>
        <artifactId>lib-a</artifactId>
        <version>1.0.0</version>
        <exclusions>
            <exclusion>
                <groupId>com.demo</groupId>
                <artifactId>lib-c</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>
```

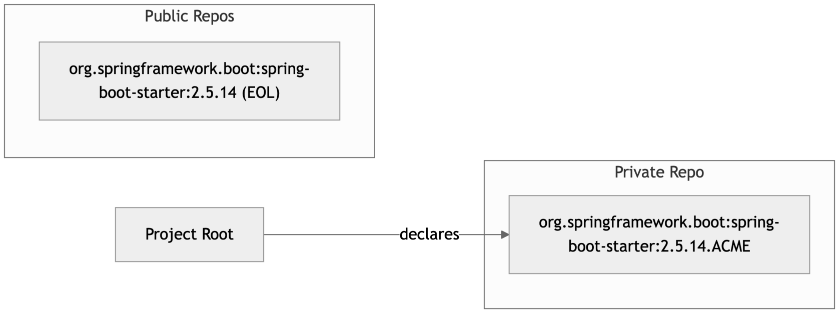
Reader Expectation: Excluding `lib-c` removes it and its transitive subtree from the graph entirely.

Why This Happens: Maven respects exclusion metadata and prunes the specified branch during graph traversal. This "sledgehammer" fix for conflicts is why the same graph behaves differently across tools.

How Developers Commonly "Fix" This: This *is* the fix for many problems (e.g., excluding `log4j` to use `slf4j`).

17. Private Repository & Direct Patch Override

What to Do When You Need a One-Off Patch: This scenario shows a more direct way to use a private, third-party repository. Instead of using a vendor-supplied BOM, we explicitly request the patched version of a dependency in our project's `<dependencies>` block.
This approach is simpler than the BOM method but can be more verbose if many dependencies need to be overridden. It is suitable for one-off patches or when a full BOM is not available.


Public Repos
org.springframework.boot:spring-boot-starter:2.5.14 (EOL)
Private Repo
Project Root
declares
org.springframework.boot:spring-boot-starter:2.5.14.ACME

The `pom.xml` adds the private repository and then directly declares a dependency on the specific patched version.

```
<repositories>
    <repository>
        <id>third-party-repo</id>
        <url>https://private-repo.com</url>
    </repository>
</repositories>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
        <version>2.5.14.ACME</version>
    </dependency>
</dependencies>
```

Reader Expectation Explicitly declaring a patched version from a private repository ensures that specific artifact is chosen over any public release.

Why This Happens A direct version declaration in `<dependencies>` is Maven's strongest versioning directive. When combined with a private repository that makes the artifact available, it overrides all other mechanisms like BOMs or transitive versioning.

Key Takeaway A direct version declaration for a dependency is the most straightforward way to ensure a specific version is used, provided that version is available in one of the configured repositories. This method overrides all other mechanisms like BOMs or transitive versions.

Related Scenarios

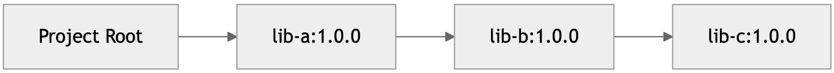
- Scenario 16: Private Repository & BOM Override - A more scalable approach using a vendor BOM.

Part II.

Gradle Scenarios

1. Basic Transitive Resolution

Or what to Do When Gradle Resolves More Than You Declared This scenario demonstrates fundamental concept of transitive dependency resolution. It shows how it automatically discovers and includes dependencies of your direct dependencies.



```
dependencies {
    implementation 'com.demo:lib-a:1.0.0'
}
```

Reader Expectation Declaring `lib-a` automatically includes its transitive dependencies `lib-b` and `lib-c`.

Why This Happens Gradle performs a depth-first traversal of the graph, following every edge in POM or Gradle module metadata files. In the absence of version conflicts, it collects every unique artifact to build the complete classpath.

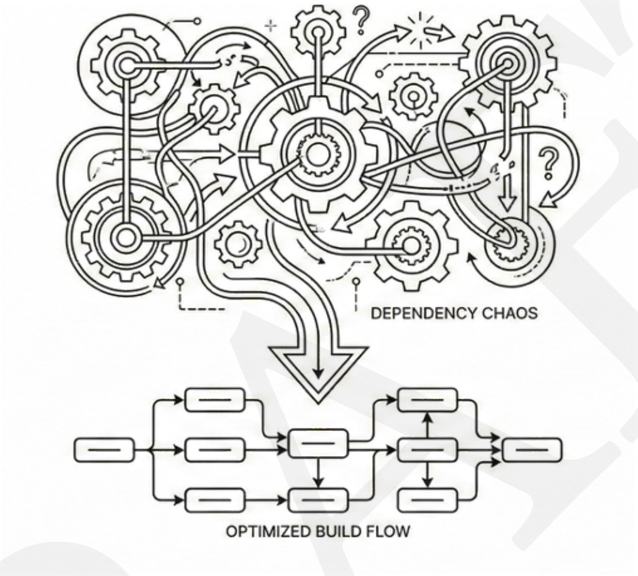
How This Scales in Real Systems In production projects, this simple chain (A $\longrightarrow$ B $\longrightarrow$ C) is replicated thousands of times. A typical Spring Boot application resolves 200–400 transitive dependencies from a handful of direct declarations. The "happy path" of zero conflicts becomes increasingly rare as the graph grows; diamond dependencies emerge naturally.

Deterministic Checks **Fast check (Gradle):**

```
./gradlew dependencies
```

What to look for: A clean tree with no unexpected versions or `omitted` nodes.

Conclusion: The dependency graph is healthy and resolving as expected.


DEPENDENCY CHAOS
OPTIMIZED BUILD FLOW