Dependency Management with Maven

[DT-0540] Metodi di sviluppo agile

Daniele Di Pompeo

daniele.dipompeo@univaq.it

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

A.A. 2024-2025

Introduction to Dependency Management

What is Dependency Management?

- Dependency management automates the process of adding, upgrading, and managing software libraries and packages needed by a project.

Why Dependency Management?

- Ensures compatibility between libraries and frameworks.
- Reduces manual effort of managing complex library versions.
- Helps avoid "dependency hell" by resolving conflicts and maintaining a consistent environment.

Early Days of Java Dependency Management

Manual Management:

- Initially, Java developers managed dependencies manually by downloading and adding '.jar' files to projects.
- Dependencies were stored locally in project directories, making it difficult to manage and share libraries across projects.

Challenges:

- Version conflicts (known as "JAR Hell") when multiple libraries required different versions of the same dependency.
- No standardized way to handle transitive dependencies (dependencies of dependencies).

Emergence of Build Tools (Ant and Ivy)

Apache Ant (2000):

- Ant was introduced as a build automation tool, allowing developers to script the build process.
- However, Ant required developers to manually specify dependencies and did not handle transitive dependencies.

Apache Ivy (2004):

- Ivy was introduced as an extension to Ant to manage dependencies.
- Ivy provided better dependency resolution, allowing projects to define and resolve transitive dependencies.

Limitations:

- Ant and Ivy configurations were verbose, requiring XML scripting.
- Dependency management was still not fully standardized.

Modern Solutions (Maven and Gradle)

Apache Maven (2004):

- Maven introduced a convention-over-configuration approach, reducing manual setup.
- Dependencies were managed through a centralized repository system, and transitive dependencies were automatically resolved.
- ‘pom.xml‘ (Project Object Model) file standardized project and dependency configuration.

Gradle (2009):

- Gradle combined the best of Maven and Ant, offering a more flexible, scriptable approach to dependency management.
- Utilizes a Groovy (or Kotlin) DSL, making configurations more concise and powerful.

Modern Solutions (Maven and Gradle)

Impact:

- Modern dependency management tools improved reproducibility, automation, and collaboration in Java projects.

Maven Overview

What is Maven?

- Maven is a popular build automation and dependency management tool for Java projects.
- Uses an XML file (‘pom.xml‘) to configure project dependencies, build lifecycle, and plugins.

Benefits of Using Maven:

- Simplifies dependency management.
- Standardizes the build process across projects.
- Integrates with a wide range of plugins for added functionality.

The `pom.xml` File

Purpose of `pom.xml`:

- The ‘pom.xml‘ file (Project Object Model) is the core configuration file for a Maven project.
- Defines project metadata, dependencies, plugins, build settings, and other configurations.

Common Sections in `pom.xml`:

- `<dependencies>` - Lists libraries and frameworks required by the project.
- `<build>` - Specifies build configuration, including plugins.
- `<properties>` - Defines project-wide properties (e.g., Java version).

Maven Build Lifecycle Phases

Maven Lifecycles:

- **Clean**: Removes previous build artifacts.
- **Default (Build)**: Compiles, tests, and packages the code.
- **Site**: Generates project documentation.

Key Phases:

- Each lifecycle consists of sequential phases, which Maven executes in order.
- Most commonly used lifecycle is the Default (Build) lifecycle.

Detailed Phases of the Default Lifecycle

Phases of Default Lifecycle:

- **validate**: Validates the project structure and dependencies.
- **compile**: Compiles the source code.
- **test**: Runs unit tests using a testing framework (e.g., JUnit).
- **package**: Packages the compiled code (e.g., into a ‘.jar‘ or ‘.war‘ file).
- **verify**: Runs checks to validate the package.
- **install**: Installs the package into the local Maven repository.
- **deploy**: Deploys the package to a remote repository for sharing.

Note: Running ‘mvn install‘ will execute all phases up to ‘install‘.

Transitive Dependencies in Maven

What are Transitive Dependencies?

- Transitive dependencies are dependencies of dependencies.
- Maven automatically includes required dependencies for the libraries you specify.

Example:

- Adding a dependency on *spring-core* may automatically include *spring-context*, *spring-beans*, etc.

Benefits:

- Reduces the need for manually specifying every library.
- Simplifies management of complex dependency trees.

Managing Dependency Scope in Maven

Dependency Scopes:

- **compile** (default): Available in all phases; included in final package.
- **provided**: Available at compile time but not included in the final package (e.g., ‘javax.servlet‘).
- **runtime**: Not required at compile time but needed during runtime (e.g., JDBC drivers).
- **test**: Used only during the test phase, not included in final package.
- **system**: Requires an explicit path on the system; rarely used.

Purpose of Scopes:

- Helps control where and when dependencies are used in the build lifecycle.

Example: Full Maven Build Process

Typical Commands in Maven Build Process:

- `mvn clean` - Cleans the project, removing any previously compiled files.
- `mvn compile` - Compiles the project's source code.
- `mvn test` - Runs the unit tests.
- `mvn package` - Packages the compiled code into a *.jar* or *.war*.
- `mvn install` - Installs the package into the local repository.
- `mvn deploy` - Deploys the package to a remote repository.

End-to-End Example:

- Running `mvn install` will execute the entire build lifecycle up to 'install'.

Example: Dependency Management in Maven

Adding a Dependency:

- Dependencies are defined in ‘pom.xml‘ using the ‘¡dependencies¿‘ tag.
- Each dependency specifies its group ID, artifact ID, and version.

Example: Adding JUnit as a Dependency

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Example: Dependency Management in Maven

Explanation:

- <scope>test</scope> specifies that JUnit is only used during the test phase.

Best Practices for Dependency Management with Maven

- **Use Dependency Scopes**: Define scopes to limit dependency usage to necessary phases.
- **Avoid Conflicting Versions**: Ensure no conflicting versions of libraries are added.
- **Use a Repository Manager**: Use tools like Nexus or Artifactory to manage dependencies centrally.
- **Specify Dependency Versions**: Lock versions to avoid unexpected updates and maintain consistency.
- **Regularly Update Dependencies**: Ensure dependencies are up-to-date for security and performance.

What is a Maven Plugin?

Definition:

- A Maven plugin is an extension to Maven that provides additional functionality to automate tasks in the build lifecycle.
- Plugins allow Maven to compile code, run tests, package code, and perform many other tasks.

Why Use Maven Plugins?

- Plugins extend Maven's capabilities beyond just compiling and packaging.
- Automate repetitive tasks and integrate various tools into the Maven build process.

Types of Maven Plugins

Two Main Types:

- **Build Plugins:**
  - Execute during the build lifecycle phases (e.g., compile, test, package).
  - Examples: *maven-compiler-plugin*, *maven-surefire-plugin*.
- **Reporting Plugins:**
  - Generate reports, documentation, and metrics after the build.
  - Examples: *maven-site-plugin*, *maven-javadoc-plugin*.

Execution:

- Plugins are bound to specific phases in the build lifecycle (e.g., compile, test).

Common Maven Plugins

Examples of Popular Maven Plugins:

- **maven-compiler-plugin:** Compiles Java source code.
  - Example: *mvn compile*
- **maven-surefire-plugin:** Runs unit tests.
  - Example: *mvn test*
- **maven-jar-plugin:** Packages code into a JAR file.
  - Example: *mvn package*
- **maven-clean-plugin:** Removes the ‘target‘ directory, clearing out compiled files.
  - Example: *mvn clean*

Configuring a Maven Plugin

Configuring Plugins in pom.xml:

- Plugins are defined in the *<build>* section of the *pom.xml*.
- You can specify the version, configuration, and execution goals.

Configuring a Maven Plugin

Example: Configuring the Compiler Plugin

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.8.1</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Benefits of Using Maven Plugins

- **Automates Repetitive Tasks:** Reduces manual work by automating common tasks like compiling, testing, and packaging.
- **Improves Build Consistency:** Plugins ensure that builds are consistent across environments.
- **Extends Functionality:** Allows integration with external tools and additional functionalities.
- **Customizable Builds:** Plugins can be customized to suit specific project needs.

Summary of Maven Plugins

- Maven plugins provide extended functionality to automate and customize the build process.
- They are divided into build and reporting plugins, bound to specific lifecycle phases.
- Common plugins include the compiler, surefire (testing), and jar (packaging) plugins.
- Configuring plugins in the ‘pom.xml‘ makes it easy to set up build automation for Java projects.

Summary

- Maven automates dependency management, simplifying the inclusion and updating of libraries.
- The Default Lifecycle provides sequential phases from validation to deployment.
- Using proper dependency scopes and managing transitive dependencies reduces conflicts and ensures efficiency.
- Following best practices helps maintain a stable and secure project environment.

Example

Creating a Default Maven Project with JUnit 5

Bash Command:

```
mvn archetype:generate -DgroupId=com.example \
  -DartifactId=my-app \
  -DarchetypeArtifactId=maven-archetype-quickstart \
  -DarchetypeVersion=1.4 \
  -DinteractiveMode=false
```

Creating a Default Maven Project with JUnit 5

Explanation:

- `-DgroupId=com.example`: Sets the group ID (package structure) for the project.
- `-DartifactId=my-app`: Names the project as "my-app".
- `-DarchetypeArtifactId=maven-archetype-quickstart`: Uses the quickstart archetype to generate a simple project structure.
- `-DarchetypeVersion=1.4`: Specifies the version of the archetype.
- `-DinteractiveMode=false`: Runs the command in non-interactive mode.

Creating a Default Maven Project with JUnit 5

Adding JUnit 5 Dependency to `pom.xml`

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.7.0</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Creating a Default Maven Project with JUnit 5

Explanation:

- The `junit-jupiter-engine` provides the JUnit 5 test engine.
- `<scope>test</scope>` ensures that JUnit is only used in the test phase.

Final Command:

- Run `mvn test` to verify the setup and execute tests.

Overview of Maven Project Structure

Overview of Maven Project Structure

Maven's Default Directory Layout:

- Maven follows a standardized project structure, making projects consistent and easier to navigate.
- The root directory contains source code, resources, tests, and the 'pom.xml' file for configuration.

Key Components:

- `src/` - Contains source code and resources.
- `pom.xml` - Project Object Model file, defines dependencies, plugins, and build configuration.

`src/main` Directory

Purpose of `src/main`:

- The ‘src/main‘ directory is used for production code and resources.

Subdirectories:

- `src/main/java` - Stores Java source files for the application.
- `src/main/resources` - Contains non-Java resources such as configuration files, properties, XML files, and other assets needed by the application.

`src/test` Directory

Purpose of `src/test`:

- The ‘src/test‘ directory holds test code and resources.
- Used for unit and integration tests, helping ensure code quality and functionality.

Subdirectories:

- `src/test/java` - Stores Java test files, typically using frameworks like JUnit or TestNG.
- `src/test/resources` - Contains resources required during testing, like test configuration files or mock data.

Example Maven Project Structure

Complete Example Structure:

```
my-app/
|-- pom.xml
|-- src/
    |-- main/
    |   |-- java/
    |   |   |-- com/example/App.java
    |   |-- resources/
    |       |-- application.properties
    |-- test/
        |-- java/
        |   |-- com/example/AppTest.java
        |-- resources/
            |-- test-config.properties
```

Example Maven Project Structure

Explanation:

- `src/main/java` - Production source code.
- `src/main/resources` - Production resources.
- `src/test/java` - Test code.
- `src/test/resources` - Test resources.
- `pom.xml` - Project configuration.

Benefits of Maven Project Structure

- **Consistency Across Projects:** Standard structure makes it easier for developers to navigate and understand the layout.
- **Build Automation Compatibility:** Integrates seamlessly with Maven's lifecycle, allowing automated builds and dependency management.
- **Scalability and Modularity:** Separates production and test code, making the project easier to scale and manage.
- **Tool Support:** Widely supported by IDEs and CI/CD tools, enabling smooth development workflows.

Dependency Management with Maven

[DT-0540] Metodi di sviluppo agile

Daniele Di Pompeo

daniele.dipompeo@univaq.it

UNIVERSITÀ DEGLI STUDI DELL'AQUILA

A.A. 2024-2025