Gen C Maven

Table of Contents

1. 1. Introduction to Maven

What is Maven? - Maven is a build automation tool primarily used for Java projects. It helps manage project dependencies, compile code, run tests, and package applications.

Maven is a build automation tool used primarily for Java projects.

It helps you:

- Create Project from Scratch
- Build your project
- Manage project dependencies (external libraries)
- Run tests
- Package your code (into .jar or .war)
- Deploy your application
- Simplifies the build process
- Manages dependencies efficiently
- Provides a standardized project structure
- Integrates with various IDEs

- Generate project reports and documentation
- Integrate with CI/CD tools like Jenkins, GitHub Actions, etc

2. 2. Maven Installation

Prerequisites - Java Development Kit (JDK) installed

Steps to Install Maven 1. Download Maven from the [official website](https://maven.apache.org/download.cgi). 2. Extract the downloaded archive. 3. Add the `bin` directory of the extracted folder to the system's `PATH` environment variable. 4. Verify the installation by running `mvn -v` in the command line.

3. 3. Maven Project Structure

Creating Maven Project

```
mvn archetype:generate -DgroupId=com.example -DartifactId=My-Maven-Project
-DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

Standard Directory Layout

```
my-app
|-- src
|   |-- main
|   |   `-- java
|   |       `-- com
|   |           `-- mycompany
|   |*    `-- app
|   |*        `-- App.java
|   `-- test
|       `-- java
|           `-- com
|*    `-- mycompany
|*        `-- app
|*            `-- AppTest.java
|-- pom.xml
```

4. 4. POM File

What is a POM File? - The Project Object Model (POM) file (`pom.xml`) is the fundamental unit of work in Maven. It contains information about the project and configuration details used by Maven to build the project.

Basic Structure of a POM File

```
<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/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.mycompany.app</groupId>
    <artifactId>my-app</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <!-- Dependencies go here -->
    </dependencies>
</project>
```

5. 5. Dependencies

Adding Dependencies - Dependencies are external libraries required by the project. They are specified in the `<dependencies>` section of the POM file.

Example Dependency

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
```

6. 6. Build Lifecycle

Phases of the Build Lifecycle - Validate - Compile - Test - Package - Verify - Install - Deploy

```
mvn clean install
```

7. 7. Plugins

What are Plugins? - Plugins are used to perform specific tasks such as compiling code, running tests, packaging, and deploying.

Commonly Used Plugins

```
<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>
```

8. 8. Example Maven Project

Creating a Simple Maven Project

```
mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=my-app
-DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

Adding Dependencies - Add the following dependency to the `pom.xml` file:

```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.12.0</version>
</dependency>
```

Building the Project

```
mvn clean install
```

9. Maven Project Structure

Project Structure:

```
my-app
|-- src
|   |-- main
|   |   `-- java
|   |       `-- com
|   |           `-- mycompany
|   |*    `-- app
|   |*        `-- App.java
|   `-- test
```

```
|       `-- java
|           `-- com
|*    `-- mycompany
|*        `-- app
|*            `-- AppTest.java
|-- pom.xml
```

App.java:

```
package com.mycompany.app;

public class App {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

AppTest.java:

```
package com.mycompany.app;

import org.junit.Test;
import static org.junit.Assert.assertTrue;

public class AppTest {
    @Test
    public void testApp() {
        assertTrue(true);
    }
}
```

pom.xml:

```
<!-- Root element of the Maven Project Object Model (POM) -->
<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/xsd/maven-4.0.0.xsd">

    <!-- POM model version (always keep this as 4.0.0) -->
    <modelVersion>4.0.0</modelVersion>

    <!-- Group ID uniquely identifies your project group or organization -->
    <groupId>com.mycompany.app</groupId>

    <!-- Artifact ID is the name of the project or module -->
    <artifactId>my-app</artifactId>
```

```
    <!-- Version of your project -->
    <version>1.0-SNAPSHOT</version>

    <!-- Dependencies section - where you declare all external libraries you need -->
    <dependencies>
        <!-- JUnit dependency for writing unit tests -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <!-- 'test' scope means this dependency is only used during test phase -->
            <scope>test</scope>
        </dependency>

        <!-- Apache Commons Lang - provides helper methods for working with strings,
numbers, etc. -->
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-lang3</artifactId>
            <version>3.12.0</version>
        </dependency>
    </dependencies>

    <!-- Build section for configuring plugins -->
    <build>
        <plugins>
            <!-- Maven Compiler Plugin to set Java version for compiling the code -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <!-- Java version used to compile the code -->
                    <source>1.8</source>
                    <!-- Java version used for running the compiled code -->
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

This structure and these examples should provide a comprehensive introduction to Maven, covering its key features and demonstrating its usage through a simple project.

10. Effective POM

What is Effective POM?

- The **Effective POM** is the final version of the POM (Project Object Model) file that Maven uses after combining:
- Your project's `pom.xml`
- Parent POM (if any)
- Super POM (default Maven settings)
- Settings from profiles and plugins
- It helps you understand all inherited and default configurations that affect your project.

Why is it Useful?

- To debug issues with dependencies, plugins, and configurations.
- To understand what values Maven is **actually** using during build.
- To see inherited settings from the parent or the default Super POM.

How to View Effective POM

Use this command:

```
mvn help:effective-pom
```

This will print the effective POM in your terminal, showing merged values from all sources.

Example

Suppose your `pom.xml` looks like:

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.1.2</version>
    </parent>
    <groupId>com.example</groupId>
    <artifactId>my-app</artifactId>
    <version>1.0.0</version>
</project>
```

You may not see build plugins or dependency versions in your `pom.xml`, but when you run:

```
mvn help:effective-pom
```

You will see all inherited configuration like:

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.10.1</version>
      <configuration>
        <source>17</source>
        <target>17</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Summary

- **Effective POM** shows the complete configuration used by Maven.
- Helps in debugging and understanding project settings.
- Use `mvn help:effective-pom` to see it.

11. Understanding `<parent>` Tag

Purpose of `<parent>` Tag

- The `<parent>` tag is used to inherit configuration from another POM file.
- This allows you to reuse common build settings, plugin configurations, dependency versions, and properties across multiple projects.

Example

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.4.4</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
```

Explanation of Each Element

- `groupId` – Group ID of the parent project (here, Spring Boot).
- `artifactId` – Artifact ID of the parent project.

- `version` – Version of the parent project.
- `relativePath` – Tells Maven where to find the parent POM:
  - If left empty (`<relativePath/>`), Maven will **not** look for a local file but instead fetch it from the repository (like Maven Central).
  - Default value is `../pom.xml`, which Maven uses to look for a parent one directory above.

How It Works

- When you specify a parent POM:
  - Your project automatically inherits configurations (like plugin versions, encoding, Java version, dependency management, etc.) from the parent.
  - You don't need to redefine common configurations in every project.
- In this case, `spring-boot-starter-parent` provides:
  - Default versions for commonly used dependencies.
  - Plugin configurations like `maven-compiler-plugin`.
  - Sensible defaults (UTF-8 encoding, Java version compatibility, etc.).

Why Use It?

- Reduces duplication across multiple Spring Boot projects.
- Ensures consistency in builds and dependency versions.
- Simplifies maintenance and upgrades.

Summary

- The `<parent>` tag is key to inheriting configurations in Maven.
- Spring Boot's parent POM simplifies setup for Spring-based projects.
- Use `relativePath` if the parent POM is in a local directory, or leave it blank to fetch from a remote repository.

12. What is Maven Lifecycle?

- A **build lifecycle** is a well-defined sequence of phases used to build and deploy a project.
- Each phase performs a specific task.
- Maven has **three built-in lifecycles**:
  - `default` – handles your project deployment.
  - `clean` – handles project cleaning.
  - `site` – handles project documentation.

Default Lifecycle Phases (Most Common)

The `default` lifecycle contains the following key phases:

Phase	Description
`validate`	Validates the project structure and configuration.
`compile`	Compiles the Java source code.
`test`	Runs unit tests using a suitable testing framework (like JUnit).
`package`	Packages the compiled code into a JAR or WAR file.
`verify`	Runs any checks on test results or integration tests.
`install`	Installs the built artifact into the local Maven repository (`~/.m2`).
`deploy`	Copies the final package to a remote repository for sharing.

Clean Lifecycle

Phase	Description
`pre-clean`	Perform tasks before cleaning.
`clean`	Deletes previously built artifacts.
`post-clean`	Tasks to perform after cleaning.

Site Lifecycle

Phase	Description
`pre-site`	Prepares for site generation.
`site`	Generates project documentation.
`post-site`	Final adjustments to documentation.
`site-deploy`	Deploys the generated site to a web server.

Running Maven Phases

You can run Maven phases using the command:

```
mvn clean install
```

This command runs the following in order: `clean` → `validate` → `compile` → `test` → `package` → `install`

Note

- Each phase runs all previous phases automatically in the correct order.
- You don’t need to run each phase manually.

Summary

- Maven lifecycle simplifies project build and deployment.

- Just run one command and Maven does everything for you in sequence.
- The three key lifecycles are: **default**, **clean**, and **site**.

13. Maven `validate` Phase Failure

What does the `validate` phase do?

- The `validate` phase checks if the project is correctly structured.
- It ensures all required configuration is present before moving forward to compile or test.

When does `validate` fail?

It fails if: * Mandatory elements in the `pom.xml` are missing (like `groupId`, `artifactId`, or `version`). * The directory structure is invalid. * Plugins or dependencies have incorrect syntax or invalid references.

Example Scenario

Suppose you have a `pom.xml` that looks like this:

```
<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/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <!-- Missing groupId -->
    <artifactId>my-app</artifactId>
    <version>1.0</version>
</project>
```

Result

If you run:

```
mvn validate
```

You will see an error like:

```
[ERROR] The groupId cannot be empty.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
```

Fix

Add the missing `groupId`:

```
<groupId>com.example</groupId>
```

Summary

The `validate` phase is useful for catching configuration mistakes early before wasting time compiling or testing.

14. Maven Commands

1. mvn clean

- Deletes the `target` directory, which contains compiled code and temporary files.
- Use this to start a clean build.

```
mvn clean
```

2. mvn compile

- Compiles the source code of the project.

```
mvn compile
```

3. mvn test

- Runs the unit tests using a testing framework like JUnit.

```
mvn test
```

4. mvn package

- Compiles code, runs tests, and packages it into a `.jar` or `.war` file.

```
mvn package
```

5. mvn install

- Installs the package (JAR/WAR) into your local Maven repository.
- This allows other local projects to use it as a dependency.

```
mvn install
```

6. mvn verify

- Runs checks on test results to ensure they meet the criteria for success.

```
mvn verify
```

7. mvn clean install

- Commonly used to do everything from scratch: clean, compile, test, and install the package locally.

```
mvn clean install
```

8. mvn dependency:tree

- Displays the project's dependency hierarchy.

```
mvn dependency:tree

mvn dependency:resolve
```

9. mvn help:effective-pom

- Shows the final `pom.xml` after inheritance and interpolation.

```
mvn help:effective-pom
```

10. mvn site

- Generates a project website with reports.

```
mvn site
```

15. References

- Maven Central Repository
- Spring Repository