Contents

[Maven 1](#_Toc157361216)

[Key concepts and features of Maven include: 1](#_Toc157361217)

[POM (Project Object Model): 1](#_Toc157361218)

[Build Lifecycle: 1](#_Toc157361219)

[Dependency Management: 1](#_Toc157361220)

[Plugins 1](#_Toc157361221)

[Repository System: 1](#_Toc157361222)

[Convention Over Configuration: 1](#_Toc157361223)

[maven is widely adopted for project building, reporting, documentation. 2](#_Toc157361224)

[Reporting and Documentation: 2](#_Toc157361225)

[What is a build tool? 2](#_Toc157361226)

[Generate Source Code: 2](#_Toc157361227)

[Generate Documentation from Source Code: 2](#_Toc157361228)

[Compile the Source: 2](#_Toc157361229)

[Compile Code Packages into JAR or ZIP Files: 2](#_Toc157361230)

[Install the Packages in Repositories: 2](#_Toc157361231)

[Maven Build Lifecycle 3](#_Toc157361232)

[Default Lifecycle: 3](#_Toc157361233)

[Clean Lifecycle: 3](#_Toc157361234)

[Why Clean? 4](#_Toc157361235)

[Site Lifecycle: 4](#_Toc157361236)

[Why Site Lifecycle? 5](#_Toc157361237)

[Goals 5](#_Toc157361238)

[Maven Repositories 6](#_Toc157361239)

[Local Repository: 6](#_Toc157361240)

[Central Repository: 6](#_Toc157361241)

[Remote Repositories: 6](#_Toc157361242)

[When They Are Used: 6](#_Toc157361243)

# Maven

Maven is a widely-used build automation and project management tool primarily used for Java projects, although it can be used for projects in other languages as well. It provides a comprehensive set of conventions and plugins for managing the entire build lifecycle, including tasks such as compiling source code, running tests, packaging artifacts, managing dependencies, and more.

## Key concepts and features of Maven include:

POM (Project Object Model): Maven uses a declarative XML file called pom.xml to define the project structure, dependencies, plugins, and build configurations. The POM file serves as the project's configuration and management file.

Build Lifecycle: Maven defines a set of standard build phases (e.g., compile, test, package, install) and binds plugins to these phases. This standardization helps ensure consistency across projects.

Dependency Management: Maven simplifies dependency management by automatically downloading and managing project dependencies from repositories. Dependencies are declared in the POM file, and Maven resolves and retrieves them from remote repositories.

Plugins: Maven uses plugins to extend its functionality. Plugins can be used to perform various tasks such as compiling code, running tests, creating JAR or WAR files, generating documentation, and more.

Repository System: Maven relies on a centralized repository system where it stores and retrieves artifacts (compiled libraries, plugins, etc.). There are public repositories like Maven Central, and organizations can set up their private repositories.

Convention Over Configuration: Maven follows the principle of "Convention Over Configuration," meaning that it provides default configurations and project structures based on conventions. This helps developers by reducing the amount of configuration they need to write.

By adhering to these principles and providing a standardized build process, Maven aims to simplify project management, improve collaboration, and streamline the build and release processes in software development projects.

## maven is widely adopted for project building, reporting, documentation.

Maven defines a standard build lifecycle with common phases such as compile, test, package, install, and deploy. This consistency simplifies the build process across different projects. Plugins can be bound to these lifecycle phases, making it easy to extend or customize the build process.

## Reporting and Documentation:

Maven supports the generation of project reports and documentation through plugins. Tools like Apache Maven Site Plugin and Doxygen can generate websites, documentation, and reports, providing a convenient way to document and share project information.

## What is a build tool?

a build tools generate source code, generate documentation from source code, compile the source, compile code packages into jar of the zip file, install the packages in repositories

### Generate Source Code:

Some build tools, especially those used in certain frameworks or languages, may provide features for code generation. For example, tools like Apache Maven and Gradle can be configured to generate source code during the build process.

### Generate Documentation from Source Code:

Build tools often include plugins or features for generating documentation from source code comments. Tools like Javadoc in Java or Doxygen in C++ are commonly integrated into build processes to create documentation based on code comments.

### Compile the Source:

One of the primary tasks of a build tool is to compile source code into executable code. This includes compiling source files written in programming languages like Java, C++, Python, etc. The build tool ensures that dependencies are resolved and the code is compiled correctly.

### Compile Code Packages into JAR or ZIP Files:

After compiling the source code, build tools package the compiled code and resources into distributable formats like JAR (Java Archive), ZIP, or other formats depending on the language and platform. This step may also involve creating executable JAR files or other types of archives.

### Install the Packages in Repositories:

Build tools often support the installation of artifacts (compiled code, libraries, etc.) into repositories. These repositories can be local, remote, or central repositories. Local installations are typically for development purposes, while remote or central repositories are used for sharing and distributing artifacts across teams or the broader community.

In the Java ecosystem, tools like Apache Maven and Gradle can install artifacts into Maven Central or other repositories. For example, Maven's install goal copies the project's artifacts (JAR, WAR, etc.) into the local Maven repository.

Build tools like Apache Maven, Gradle, and Ant are widely used to automate these tasks in a standardized and reproducible way. They provide a declarative way to define the build process, manage dependencies, and ensure consistent builds across different environments.

## Maven Build Lifecycle

Apache Maven defines a standard build lifecycle to manage the process of building and distributing projects. The Maven build lifecycle is composed of phases, and each phase represents a specific step in the project's life cycle. These phases are organized into three main build lifecycles: default, clean, and site.

### Default Lifecycle:

* The default lifecycle is the most important and is automatically executed during the build process. It includes the following phases:
* **validate**: Validates the project structure.
* **compile**: Compiles the source code of the project.
* **test**: Runs tests against the compiled source code.
* **package**: Takes the compiled code and packages it in a distributable format, such as JAR or WAR.
* **verify**: Runs checks on the results of integration tests to ensure quality criteria are met.
* **install**: Installs the packaged artifact into the local Maven repository.
* **deploy**: Copies the final package to the remote repository for sharing with other developers and projects.

**Usage:**

**mvn compile test package**

This will execute the compile, test, and package phases of the default lifecycle in sequence.

### Clean Lifecycle:

* The clean lifecycle is used to clean the project by removing any files or directories created during the build process.
* The Maven clean lifecycle is a set of predefined phases and goals that are focused on cleaning and preparing the project for a fresh build. The primary purpose of the clean lifecycle is to remove any files or directories that were generated during previous builds, ensuring a clean and consistent starting point for the next build.
* It includes the following phases:
* **pre-clean**: Executes tasks before the cleaning process.
  1. The pre-clean phase executes any tasks or plugins that need to run before the actual cleaning process.
  2. This phase is rarely used but provides a hook for custom actions before cleaning.
* **clean**: Removes the files and directories created during the build.
  1. Maven's clean lifecycle has a single phase called clean.
  2. When you run mvn clean, Maven executes the clean phase.
  3. The clean phase removes the target directory and other build artifacts from the project.
* **post-clean**: Executes tasks after the cleaning process.
  1. The post-clean phase executes any tasks or plugins that need to run after the cleaning process.
  2. Similar to pre-clean, this phase is rarely used but provides a hook for custom actions after cleaning.

**Usage:**

**mvn clean**

This command will trigger the clean phase and remove the target directory. After running this command, you have a clean project ready for a new build.

### Why Clean?

Cleaning the project before each build is important for several reasons:

* Ensures a consistent and reproducible build environment.
* Prevents potential issues caused by old or cached artifacts.
* Helps in identifying build-related problems as you start from a clean state.
* Essential when working with continuous integration and automated build systems.

### Site Lifecycle:

* The site lifecycle is used for generating project documentation and reports.
* It helps in creating a comprehensive set of documents that describe various aspects of the project, such as project information, source code metrics, test results, and more. This documentation is often generated in HTML format and can be deployed to a web server for easy access by team members or stakeholders.
* It includes the following phases:
* **pre-site**: Executes tasks before generating the site documentation.
  1. The pre-site phase executes tasks or plugins before generating the site documentation.
  2. It provides an opportunity for custom actions before the documentation generation process starts.
* **site**: Generates project documentation.
  1. The site phase is the main phase responsible for generating project documentation.
  2. When you run mvn site, Maven executes this phase.
  3. Documentation includes project information, reports (such as code analysis, testing results), and other details specified in the POM file.
* **post-site**: Executes tasks after generating the site documentation.
  1. The post-site phase executes tasks or plugins after generating the site documentation.
  2. It provides an opportunity for custom actions after the documentation generation process.
* **site-deploy**: Deploys the generated site documentation to a specified server.
  1. The site-deploy phase deploys the generated site documentation to a specified server or location.
  2. This phase is optional and is used when you want to publish the documentation to a web server or another location.

**Usage:**

**mvn site**

* This command will trigger the entire site lifecycle, generating project documentation and reports. The resulting HTML files and other artifacts will be available in the target/site directory.

### Why Site Lifecycle?

* **Documentation**: Generates comprehensive project documentation in a standardized format.
* **Reports**: Provides various reports, including code analysis, test results, and project information.
* **Sharing Information**: Enables teams to share project details and progress easily.
* **Web Deployment**: Allows deployment of the generated documentation to a web server for broader access.

## Goals

A "goal" is a **specific task or operation** that can be executed during a build phase. Goals are associated with plugins, and plugins are configured in the **Maven Project Object Model (POM) file.**

Each Maven plugin provides one or more goals that you can execute. For example, the **compiler** plugin provides the **compile** goal, the **surefire** plugin provides the **test** goal, and so on.

Goals are bound to specific phases in the build lifecycle. When you run a specific Maven phase, the associated goals for that phase are executed. For example, when you run the **compile** phase, Maven executes the goals associated with that phase, which might include the **compiler:compile** goal.

**mvn clean dependency:copy-dependencies -DoutputDirectory=target/my-dependencies package**

This command will execute the following tasks:

1. **clean**: Cleans the project by removing the target directory and other build artifacts.
2. **dependency:copy-dependencies**: Copies project dependencies to the specified location.
3. **package**: Compiles the source code, runs tests, and packages the project into a distributable format (e.g., JAR, WAR).

## Maven Repositories

Maven repositories play a crucial role in managing dependencies and artifacts in a Maven-based project.

A Maven repository is a centralized storage location for Java libraries, also known as artifacts. It stores compiled JARs, source code, and metadata needed for project builds.

### Local Repository:

* The local repository is a directory on your development machine where Maven stores project-specific dependencies and artifacts.
* When you build a Maven project for the first time, dependencies are downloaded from remote repositories (like Central) and stored in your local repository.
* Subsequent builds can use the locally cached dependencies without re-downloading them from remote repositories, providing faster build times.

### Central Repository:

* The Central Repository is the default public repository for Maven artifacts.
* It is a remote repository hosted by Sonatype and contains a vast collection of open-source Java libraries and artifacts.
* Maven automatically checks the Central Repository for dependencies unless explicitly configured otherwise.

### Remote Repositories:

* Remote repositories are external repositories, other than the local repository, that can be specified in a Maven project.
* They can be public repositories, private repositories within an organization, or any other remote location hosting Maven artifacts.
* Maven checks remote repositories to download dependencies that are not available in the local repository.

### When They Are Used:

#### Local Repository:

* Used to store project-specific dependencies on your local machine.
* Avoids the need to download dependencies from remote repositories every time you build the project.
* Helps in offline development, as Maven can use the locally cached dependencies without an internet connection.

#### Central Repository:

* Used by default for fetching dependencies.
* Primary source for commonly used open-source Java libraries.
* Maven automatically checks the Central Repository for dependencies unless configured otherwise.

#### Remote Repositories:

* Used when you need to fetch dependencies from repositories other than the Central Repository.
* Useful for accessing private repositories within an organization.
* Enables fetching dependencies from custom repositories that host specific artifacts not available in the Central Repository.