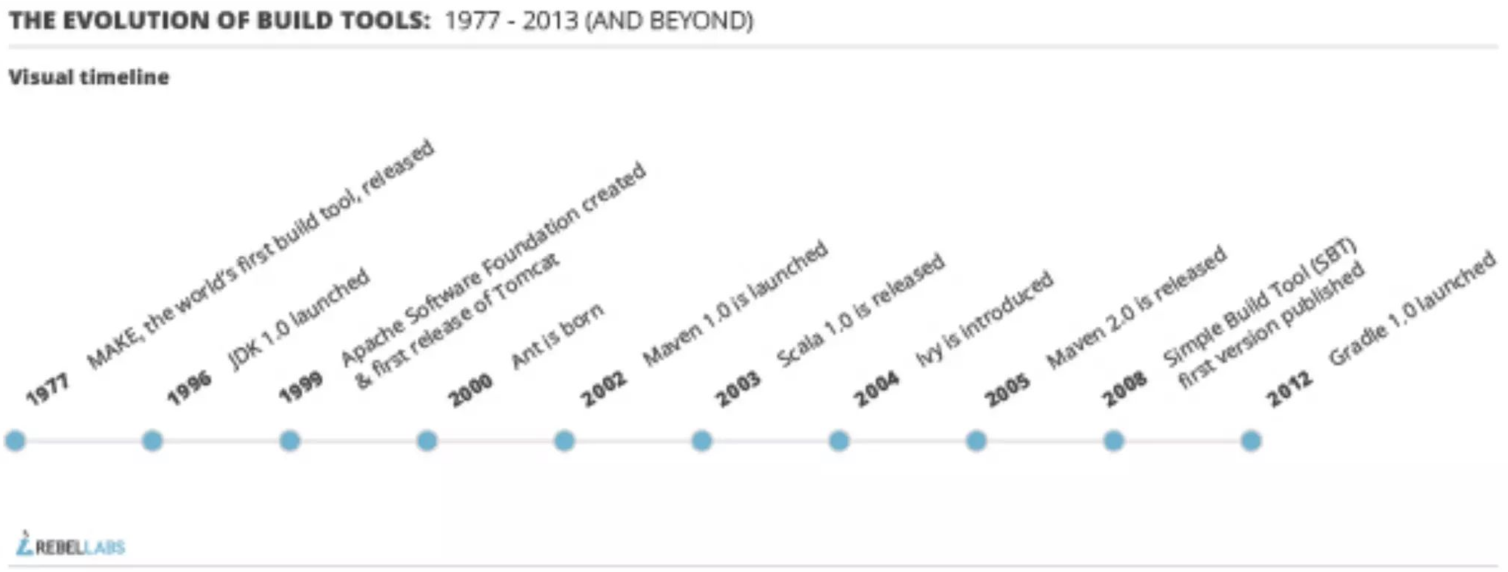
Maven Fundamentals

Rakesh Vardan
Lead SDET

Course Content

- Introduction to Maven
- Maven Installation and Setup
- Understanding Maven POM file
- Maven Dependencies
- Maven Life Cycle and Phases
- Exploring Maven Repositories
- Working with Maven Plugins
- Maven Properties and Configuration
- Maven and Test Automation
- Maven Best Practices & Tips

Course Content


THE EVOLUTION OF BUILD TOOLS: 1977 - 2013 (AND BEYOND)
Visual timeline
1977 MAKE, the world's first build tool, released
1996 JDK 1.0 launched
1999 Apache Software Foundation created & first release of Tomcat
2000 Ant is born
2002 Maven 1.0 is launched
2003 Scala 1.0 is released
2004 Ivy is introduced
2005 Maven 2.0 is released
2008 Simple Build Tool (SBT) first version published
2012 Gradle 1.0 launched
REBELLABS

What is Maven?

- Apache Maven is a **build automation** and **project management** tool that simplifies the software development process.

- It helps manage the **build lifecycle** of a project by defining a standard **project structure**, automating the **build process**, and **managing dependencies**.

Key Features of Maven:

1. Dependency Management: Maven handles project dependencies by automatically downloading and managing libraries from remote repositories.

2. Build Automation: It automates the build process, which includes compiling source code, running tests, packaging, and deploying the project.

3. Project Lifecycle Management: Maven defines a series of build phases and goals, making it easy to understand and execute a project's lifecycle.

4. Convention over Configuration: Maven follows conventions that allow projects to be built with minimal configuration. For example, the source code should be in the `src/main/java` directory.

5. Extensible: Maven is extensible, allowing users to create custom plugins and configurations to meet specific project needs.

Maven Terminology:

1. Project: A software project that is managed by Maven. Each project has a unique identifier consisting of a group ID and an artifact ID.

2. POM (Project Object Model): The POM is an XML file (pom.xml) that contains project configuration and metadata, including dependencies, plugins, and build settings.

3. Dependency: A library or module required for the project to build and run successfully. Dependencies are defined in the POM file.

4. Repository: A location, either local or remote, where Maven stores project dependencies. The local repository is typically located in the `.m2/repository` directory.

5. Plugin: A set of goals that can be executed during the build process. Plugins provide additional functionality, such as compiling source code or creating JAR files.

Installation:

- Download maven and extract it (*https://maven.apache.org/download.cgi*)

- Add **JAVA_HOME** and **MAVEN_HOME** in environment variable

- Add maven path in environment variable

- Verify Maven
    - *mvn -version*

Creating a Maven Project:

- From Command Line Interface (CLI)
  - With Interactive Mode
  - Without Interactive Mode (*easiest option!*)

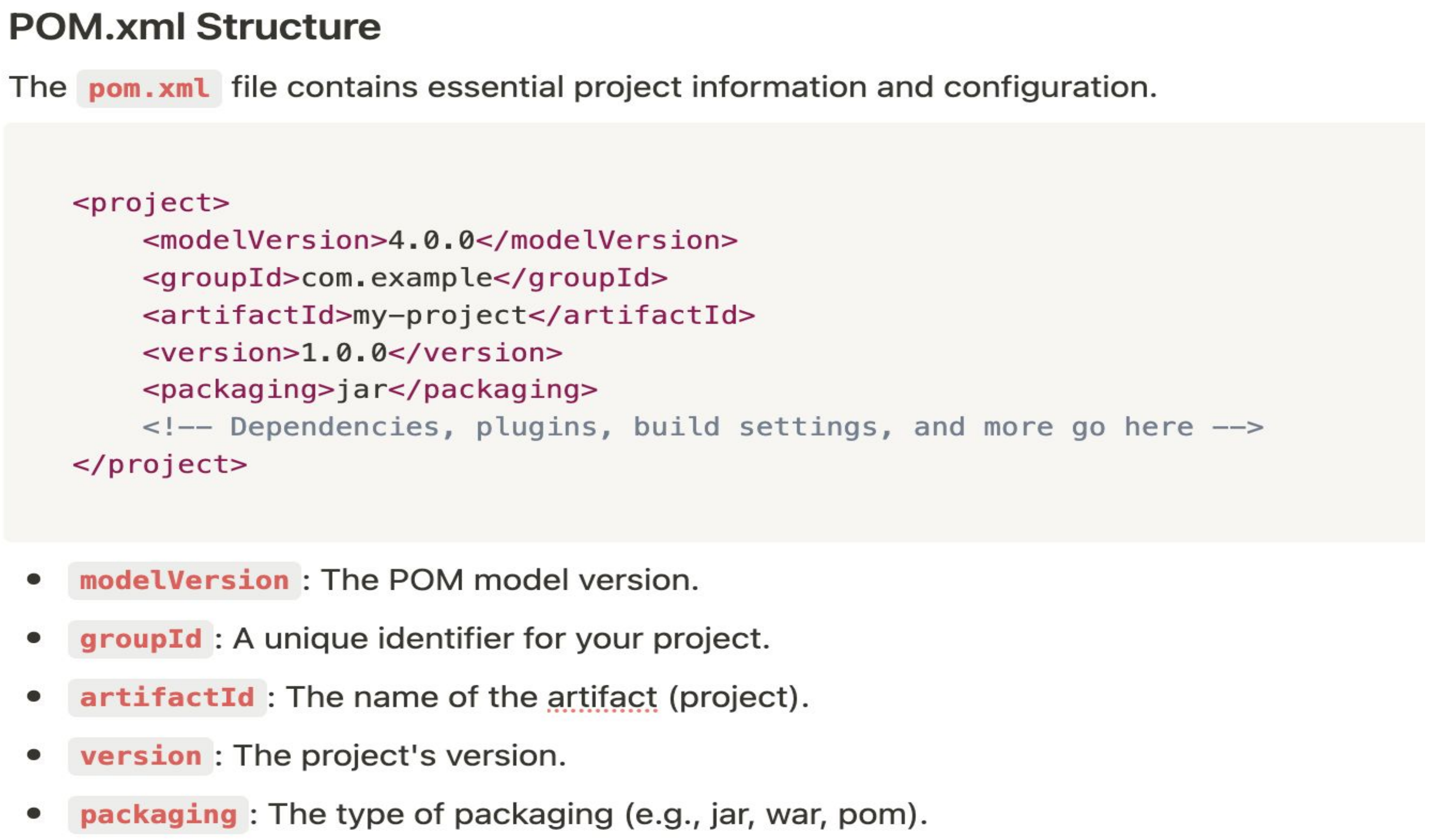
- From IDE
  - Jetbrains IntelliJ IDEA
  - Eclipse

Maven Project Structure:

```
my-project/
  ├── src/
  │    ├── main/
  │    │    ├── java/
  │    │    ├── resources/
  │    ├── test/
  │    │    ├── java/
  │    │    ├── resources/
  ├── target/
  ├── pom.xml
```

- `src/main` : Contains the main application code.
- `src/test` : Contains test code.
- `target` : The build output directory where compiled classes and packaged artifacts are placed.
- `pom.xml` : The Project Object Model (POM) file that defines project configuration.

POM (Project Object Model):



POM.xml Structure

The `pom.xml` file contains essential project information and configuration.

```xml
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>my-project</artifactId>
    <version>1.0.0</version>
    <packaging>jar</packaging>
    <!-- Dependencies, plugins, build settings, and more go here -->
</project>
```

- `modelVersion` : The POM model version.
- `groupId` : A unique identifier for your project.
- `artifactId` : The name of the artifact (project).
- `version` : The project's version.
- `packaging` : The type of packaging (e.g., jar, war, pom).

Understanding Maven Dependencies:

What are Dependencies?

- Dependencies are external libraries or frameworks your project needs to function.
- Maven manages these libraries and their versions automatically.

How Dependencies Work in Maven:

- **Declaration:** Add dependencies to the pom.xml file under the <dependencies> section.
- **Resolution:** Maven downloads these libraries from repositories and includes them in the project.
- **Scope:** Define how dependencies are used in different phases, e.g., compile, test, runtime.

Understanding Maven Dependencies:

Example dependency Declaration:

```xml
<dependency>
   <groupId>org.apache.commons</groupId>
   <artifactId>commons-lang3</artifactId>
   <version>3.15.0</version>
</dependency>
```

Key Points:

- **Transitive Dependencies:** Maven also handles dependencies of your dependencies.
- **Centralized Management:** Version and library management are handled in one place (pom.xml).

Benefits:

- Simplifies project setup and build processes.
- Ensures consistent versions across environments.

Maven Dependency Scopes:

Scope	**Description**	**Available Classpaths**	**Use Case**	**Example Library**
Compile	Default scope	Compile, runtime, test	Core libraries needed for compilation	`org.apache.commons:commons-lang3`
Provided	Required for compilation, provided at runtime	Compile, test	Servlet APIs, container-provided libs	`javax.servlet:javax.servlet-api`
Runtime	Required during execution	Runtime, test	JDBC drivers	`mysql:mysql-connector-java`
Test	Required only for testing	Test	Testing frameworks	`junit:junit`
System	Explicitly provided, not in a repository	Compile, test	Local JARs not available in repositories	`com.example:example-library` (local JAR)
Import	Used in `<dependencyManagement>` for BOMs	N/A	Importing dependency versions	`org.springframework.boot:spring-boot-dependencies`

Repositories:

1. Local Repository

a. The local repository is located in the **~/.m2/repository** (on Unix-based systems) or **C:\Users\<your-username>\.m2\repository** (on Windows).

b. It stores downloaded project dependencies and plugins

2. Central Repository

a. Maven central repository is located on the web. It has been created by the apache maven community itself

b. The path of central repository is: **https://repo.maven.apache.org/maven2/**

c. The central repository contains a lot of common libraries that can be viewed by this url **https://mvnrepository.com/**

3. Remote Repository

a. Apart from the central repository, you may have needed artifacts deployed in other remote locations. For example, in your corporate office, there may be projects or modules specific to the organization only. This remote repository will be accessible only inside the organization.

Maven Build Life-Cycle:

Maven’s **Build lifecycle** defines the sequence of phases that are executed to build and deploy a project. Each phase represents a specific stage in the build process, and the phases are executed in a predefined order.

Maven includes three built-in build life cycles:

1. **Clean** - This lifecycle handles the cleaning of the build environment.
2. **Site** - This lifecycle is used to generate project documentation and site reports.
3. **Default** - This is the main lifecycle and is responsible for the build and deployment process.

1. **Clean Lifecycle:**

   pre-clean -> clean -> post-clean

2. **Site Lifecycle:**

   pre-site -> site -> post-site -> post-site-deploy

3. **Default Lifecycle:**

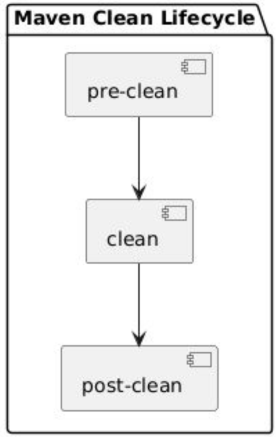
   validate -> compile -> test -> package -> verify -> install -> deploy

Clean Lifecycle

The Clean Life cycle is used to clean up the build environment. It includes the following phases:

1. **pre-clean**: Executes tasks before the cleaning process begins.
2. **clean**: Deletes the output of the previous build (e.g., target directory).
3. **post-clean**: Executes tasks after the cleaning process is complete.

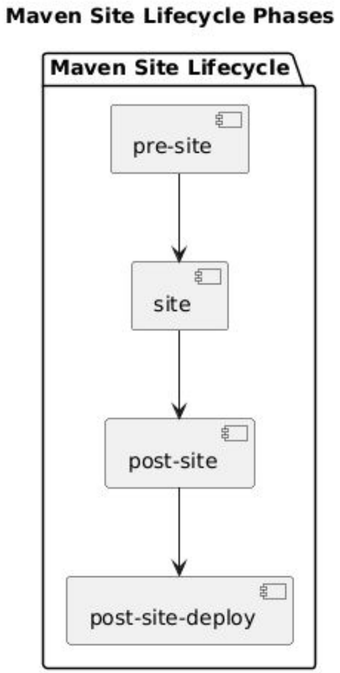
Maven Clean Lifecycle Phases



Site Lifecycle

The Site Lifecycle is used for generating project documentation and reports. It includes the following phases:

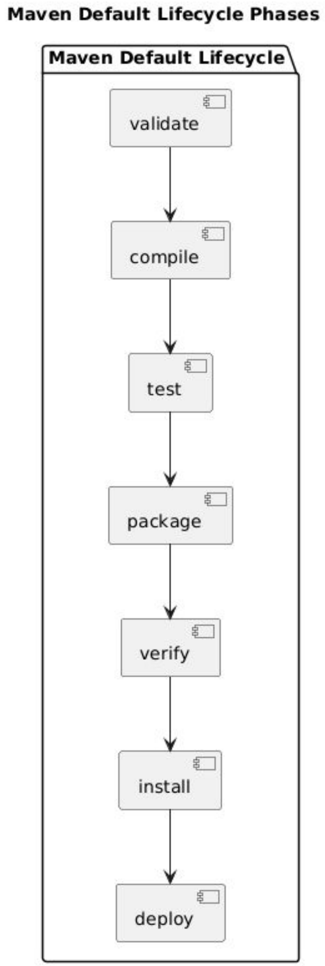
1. **pre-site**: Executes tasks before site generation.
2. **site**: Generates the project's site documentation.
3. **post-site**: Executes tasks after site generation.
4. **post-site-deploy**: Executes tasks after the site has been deployed.



Default Lifecycle

The Default Lifecycle handles the main build process. It includes the following phases:

1. **validate**: Validates the project's structure and configuration. Ensures the project is correctly set up.
2. **compile**: Compiles the source code of the project.
3. **test**: Runs tests against the compiled source code using a testing framework like JUnit/TestNG.
4. **package**: Packages the compiled code into a distributable format (e.g., JAR, WAR).
5. **verify**: Runs additional checks to verify the quality and integrity of the package.
6. **install**: Installs the package into the local Maven repository, making it available for other projects on the local machine.
7. **deploy**: Deploys the package to a remote repository so it can be shared with other developers or projects.



Mostly Used Maven Commands:

Command	**Lifecycle**	**Description**
mvn clean	Clean	Deletes the output of the previous build (e.g., target directory).
mvn compile	Default	Compiles the source code of the project.
mvn test	Default	Runs tests against the compiled source code.
mvn package	Default	Packages the compiled code into a distributable format, such as JAR or WAR.
mvn install	Default	Installs the package into the local Maven repository, making it available for other local projects.
mvn deploy	Default	Deploys the package to a remote repository, making it available to other developers or projects.
mvn site	Site	Generates the project's site documentation.

Maven Plugins:

Maven plugins are tools that extend the capabilities of Maven.

1. Plugins perform specific tasks during the build process, such as *compiling* code, *running* tests, *creating* documentation, or *packaging* applications.
2. Plugins are designed to be easily configurable and can be used to customize the build lifecycle to fit your project's needs.

Maven Properties:

- Maven properties are placeholders that you can define and use throughout your pom.xml file.
- They allow you to centralize values that may be reused in different places, making your configuration easier to manage and modify.

Example:

```
<properties>
    <java.version>11</java.version>
    <project.version>1.0.0</project.version>
    <encoding>UTF-8</encoding>
</properties>
```

Maven Profiles:

- Profiles in Maven allow you to customize the build process based on different environments or conditions.
- They enable you to specify different *configurations*, *dependencies*, or *plugins* that can be activated under certain conditions, making it easier to manage multiple build scenarios.

Maven profiles are useful for:

- **Handling Multiple Environments:** Different environments (like development, testing, staging, and production) often require different configurations or dependencies.
- **Customizing Builds:** You might need different settings for different types of builds or different projects.
- **Conditional Execution:** Running specific tasks or using specific configurations only when certain conditions are met.

Maven Best Practices & Tips:

1. Manage Dependencies Wisely

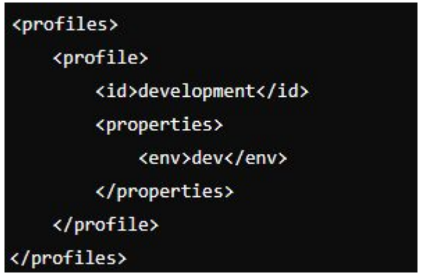
- **Use Proper Scopes:**
  - Use <scope>test</scope> for test-specific dependencies to keep them separate from production code.
- **Avoid Dependency Conflicts:**
  - Define versions in the <dependencyManagement> section to avoid version conflicts across modules.

2. Utilize Standard Project Structure

- **Adhere to Maven's Directory Layout:**
  - Follow Maven's standard project structure (src/main/java, src/test/java, src/main/resources, etc.) to ensure consistency and easier navigation.

3. Utilize Maven Profiles

- **Create Profiles for Different Environments:**
  - Use Maven profiles to handle different build configurations or environments (e.g., development, testing, production).

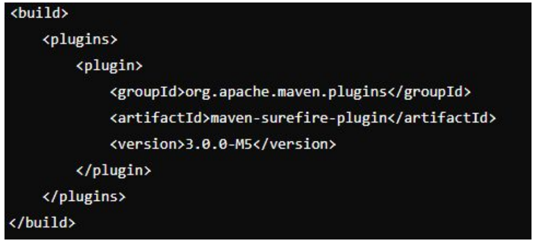


```
<profiles>
    <profile>
        <id>development</id>
        <properties>
            <env>dev</env>
        </properties>
    </profile>
</profiles>
```

Maven Best Practices & Tips:

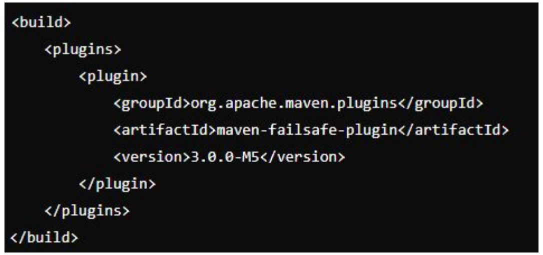
4. Configure Test Plugins Effectively

- **Use Surefire for Unit Tests:**
  - Configure the Maven Surefire Plugin to manage unit tests and generate detailed reports.



```xml
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.0.0-M5</version>
        </plugin>
    </plugins>
</build>
```

- **Use Failsafe for Integration Tests:**
  - Configure the Maven Failsafe Plugin for integration tests that run after the main build.



```xml
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-failsafe-plugin</artifactId>
            <version>3.0.0-M5</version>
        </plugin>
    </plugins>
</build>
```

Maven Best Practices & Tips:

5. Leverage Maven Wrapper

- **Ensure Consistent Maven Versions:**
  - Use the Maven Wrapper (mvnw) to maintain consistent Maven versions across different environments and systems.
  - ./mvnw clean install

6. Optimize Build Performance

- **Enable Parallel Builds:**
  - Use the -T option to run parallel builds and speed up the build process.
  - mvn clean install -T 2C

7. Regularly Update Dependencies

- **Keep Dependencies Current:**
  - Regularly update your project’s dependencies to benefit from the latest features, improvements, and security fixes.
  - Use tools like OWASP Dependency-Check or the Maven Versions Plugin.

Maven Best Practices & Tips:

8. Document and Simplify Build Process

- **Maintain Clear Documentation:**
  - Document build processes and profiles to help team members understand and maintain the Maven configuration.
- **Avoid Overly Complex Configurations:**
  - Keep your Maven configurations simple and easy to understand to avoid confusion and potential errors.

9. Validate POM File Regularly

- **Check for Errors and Consistency:**
  - Use mvn validate to check the correctness of your pom.xml and ensure that the project configuration is as expected.

Maven in Test Automation:

1. **Structured Project Layout:**
   - Maven enforces a standardized project structure (e.g., **src/main/java, src/test/java, src/test/resources**), which organizes test code and resources systematically and improves project maintainability.
2. **Centralized Dependency Management:**
   - Maven manages test dependencies (e.g., JUnit, TestNG, Selenium, REST Assured) and ensures consistent versions across projects, simplifying dependency updates and management.
   - Define and manage test libraries and frameworks in the **pom.xml** file.
3. **Consistent Build Environment:**
   - Maven provides a standardized build process, ensuring that tests run in a consistent environment across different systems and setups.
   - Use Maven phases to trigger test execution as part of the build lifecycle (e.g., **mvn test** for unit tests).
4. **Automated Test Execution:**
   - Maven integrates with plugins like Surefire and Failsafe to automate the execution of unit and integration tests, streamlining the testing process.
5. **Reporting and Analysis:**
   - Maven generates detailed test reports, helping teams analyze test results, detect failures, and maintain high code quality.
6. **Consistent Build with Maven Wrapper:**
   - Use Maven Wrapper to ensure consistent Maven versions and build processes across different environments.