



# Apache Maven

## „Automated build management“

Basic Trainings

@ CGS IT – 2018/2024 – Version 1.1.14



# Inhalt

- Chapter 00: Setup
- Chapter 01
  - What is Apache Maven and concepts
- Chapter 02
  - Maven POM, Project Description
  - Maven Directory Structure
  - Maven Web Application
- Chapter 03
  - Maven Lifecycles, Maven Multi Module Project and BOM
- Chapter 04
  - How it all Works!

# Trainier -



- Mag. Christian Schäfer
  - Software Architekt – CGS IT Solutions
  - TU-Wien/Uni Wien
- Gebiete
  - Software Architektur & Cloud Computing
  - JEE / Java Software Entwicklung
- Kontaktdaten für Rückfragen zum Kurs
- [christian.schafer@cgs.at](mailto:christian.schafer@cgs.at) “Maven Basics Schulung 2024.06”
- Tel: 0650 944 6990

# Chapter 00 : Setup

# Training git repositories

- Basic Examples

- [https://github.com/CGS-IT/train\\_maven\\_24\\_06\\_basic](https://github.com/CGS-IT/train_maven_24_06_basic)

- Further Examples

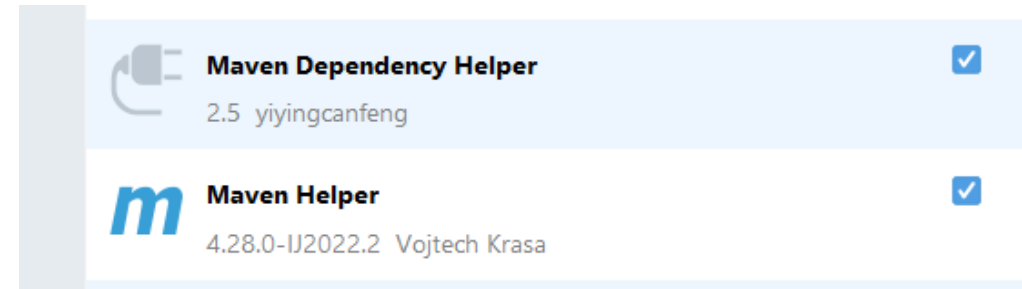
- [https://github.com/chris-cgsit/train\\_maven\\_24\\_06\\_part2](https://github.com/chris-cgsit/train_maven_24_06_part2)

# Download, Extract, Path

- Download latest maven binary  
<https://maven.apache.org/download.cgi>
- Extract to path:
  - Z.b. D:\training\apache-maven-3.9.5
- Set MAVEN\_HOME, M2\_HOME, JAVA\_HOME for command line usage
- Check if java and maven are found in command terminal
  - java -version
  - mvn -version
- Existing IntelliJ or Eclipse assumed

# IntelliJ

- Import maven projects or as module from existing source



# Chapter 01: Maven Introction

Basic Introduction, Concepts and Architecture



# What is „Apache Maven“

- A powerful tool that helps to manage and build software projects
- It simplifies:
  - Software project documentation including project structures (multiple projects)
  - Software dependency and version management
  - compiling code
  - running tests
  - packaging applications
- Organize and “easily” handle project dependencies
  - (the external libraries your project needs)
  - ensure that everything works well together.
- It uses a simple configuration file called pom.xml
  - to define the project's setup and dependencies
- Automates many repetitive tasks in software development
- Making it easier for developers to focus on writing code

# Maven – Further basic concepts

- **Convention over Configuration**

- Convention over configuration is a simple concept. Systems, libraries, and frameworks. should assume reasonable defaults without requiring that unnecessary configuration systems should “just work.”

- **A Common Interface**

- To enable build and compile tasks regardless of the project

- **Universal Reuse Through Maven Plugins**

# Maven - Conceptual Model of a “Project”

Maven maintains a model of a project:

- you are not just compiling source code into bytecode,
- you are developing a description of a software project and assigning a unique set of coordinates to a project.

A Project will include

1. Project base descriptions
2. Dependencies and version information
3. Build, Test and Packaging Configuration, Setup and Tools
4. Documentation

# Maven – Building Blocks - Architecture

- Maven-Project is a
  - Is a maven managed „project“ as described above
- Maven Modul
  - Is the pom.xml description of the project or a module only if the project is splittet into multiple maven modules.
  - Each maven module has its own pom.xml with a parent.pom
- Maven Core System
- Maven Plugins and Goals
- Maven Lifecycle
- Maven Repositories
- Maven Artefacts / Groups / Versions
  - Is the output of a project e.g. JAR-File. Each Project/Module has a group, an artefact-id-name and a version (see POM)

# Chapter 02: Maven - Elements

Maven POM Structure, LifeCycle and Goals

# Maven – POM – Project Object Model

- Describes the Project Metadata as xml file
- **Project Information**
  - groupId
  - artifactId
  - version
  - packaging
- **Packaging** - defines the output project type
- **Dependencies** - Direct dependencies used in our app.
- **Build** - Plugins, Directory Structure etc.
- **Repositories** - Holds all the dependencies and artifacts.
- Reference Model  
<https://maven.apache.org/ref/3.9.8/maven-model/maven.html>

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                              http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>at.cgsit.training01</groupId>
  <artifactId>HelloWorldTest</artifactId>

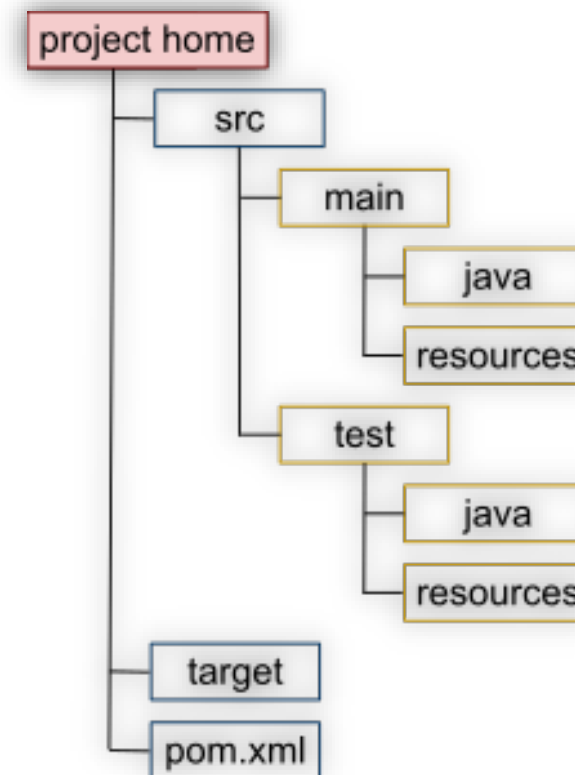
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>HelloWorldTest</name>
  <url>http://maven.apache.org</url>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

</project>
```

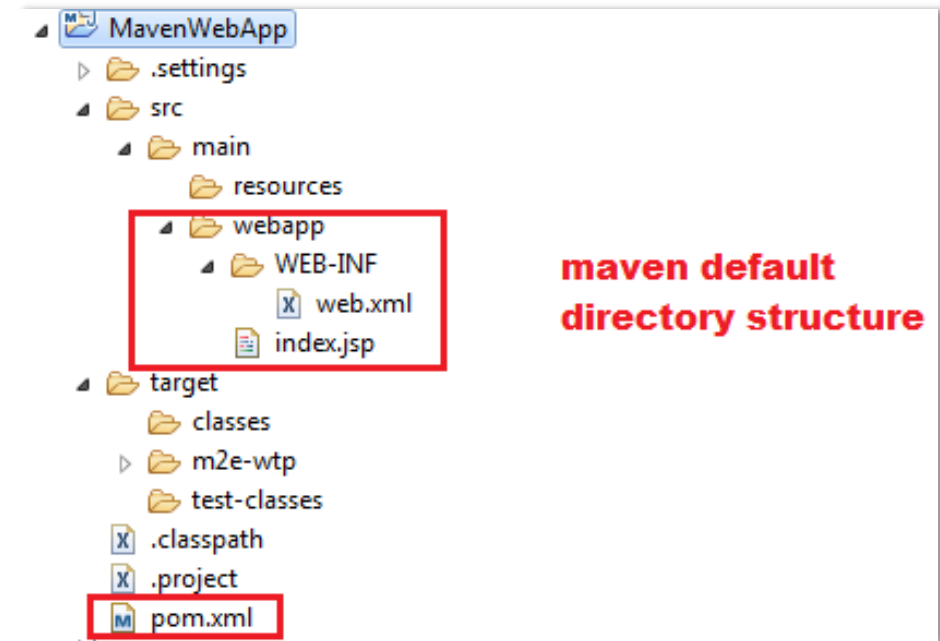
# Maven Project - Directory Structure

- The pom.xml file is located in the root director for each maven module.
- A Maven project has a standard default directory layout
  - „convention over configuration“
- src/main and src/test are the default directories where project sources are kept for and seperated test sources
- Resources: other text/xml files are kept in a resource folder, and are still packed to output or test
- Target: Directory is used for all temporary files during build, test and packaging phases.



# Maven – Web Application Layout

- For extended projects additional directories can be defined for the project.
- Defines default webapp additional directory for add files
- Check Plugin Docu for available paramters for each plugin:
  - <https://maven.apache.org/plugins/maven-war-plugin/war-mojo.html>
- Maybe check source code for the „final truth“:
  - <https://github.com/apache/maven-war-plugin/blob/master/src/main/java/org/apache/maven/plugins/war/AbstractWarMojo.java>





# Maven properties

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  <maven.compiler.target>11</maven.compiler.target>
  <maven.compiler.source>11</maven.compiler.source>
</properties>

<dependencies> ...
```

- Properties enable configuration options for the maven project and internal plugins and tools.
- In this example we set the build source file Encoding to UTF8
- And we configure the maven.compiler plugin to generate and assume java 11 source code
- List details of parameters for plugin:
  - `mvn help:describe -Dplugin=org.apache.maven.plugins:maven-compiler-plugin -Dgoal=compile -Ddetail`

# Maven: Property Resolution Order

## Properties resolution ordering:

1. -D property via command line
  1. `mvn package -DdatenbankURL=jdbc:/host11:1521`
2. `<properties>` in `<profile>` in `settings.xml`
3. `<properties>` in `<profile>` in the child pom
4. `<properties>` directly in child pom
5. `<properties>` in `<profile>` in the parent pom
6. `<properties>` directly in parent pom

## Summary:

1. Commandline first
2. Settings before child before parent
3. profile before directly defined properties

Maven „in action“  
generated maven modules

# Maven artefact archetypes

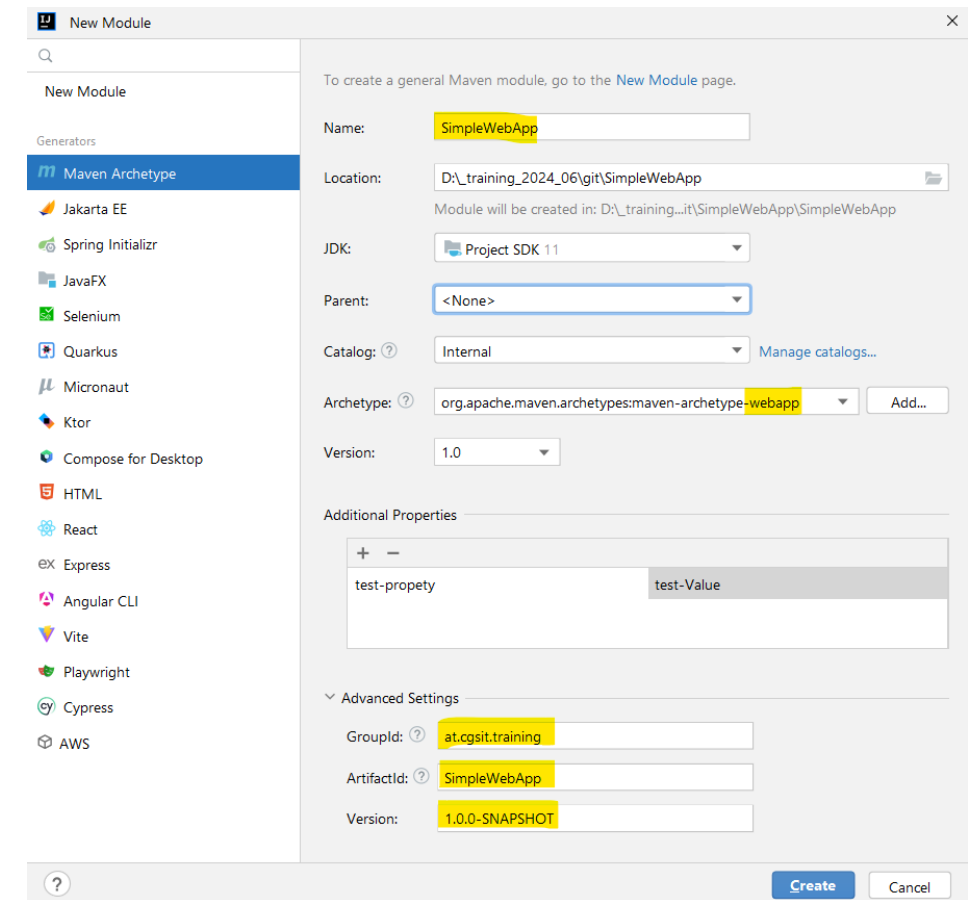
- ***mvn archetype:generate***
  - Creates an application in interactive mode; will prompt for
    - archetype - default 414 (maven-archetype-quickstart)
    - archetype version - default 1.1
    - groupId - com.hashedin.hu
    - artifactId - HelloWorldMaven
    - version - default 1.0-SNAPSHOT
    - package - default *groupId*

## ***Example:***

***mvn archetype:generate -DgroupId=at.cgsit.training01 -DartifactId=HelloWorldTest -DinteractiveMode=false -DarchetypeArtifactId=maven-archetype-quickstart***

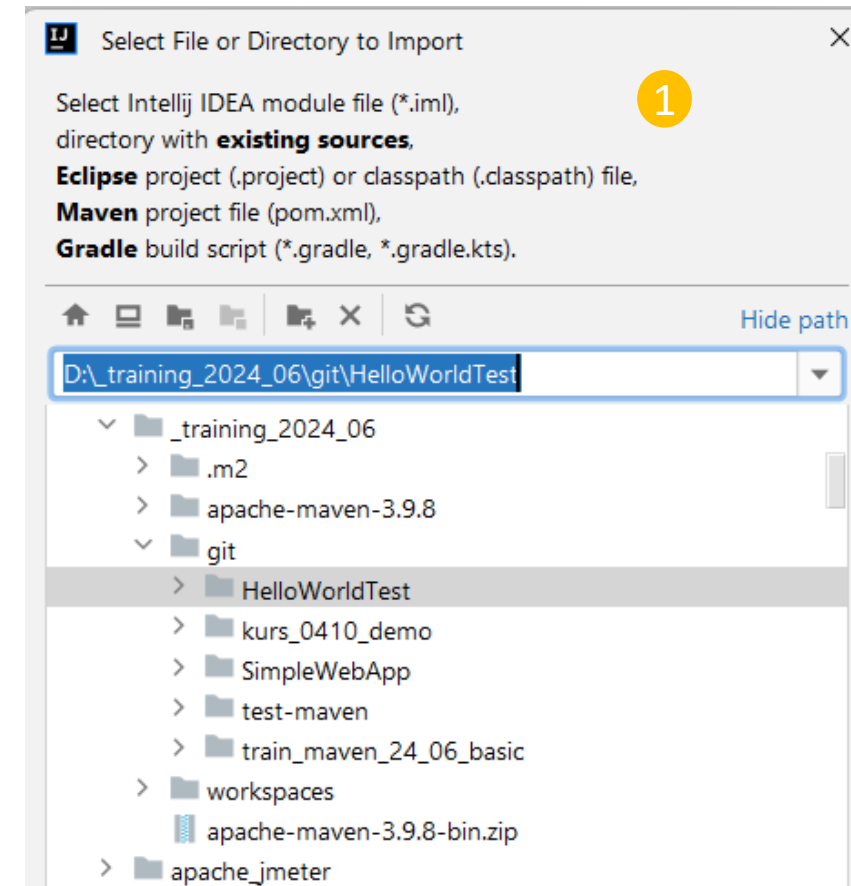
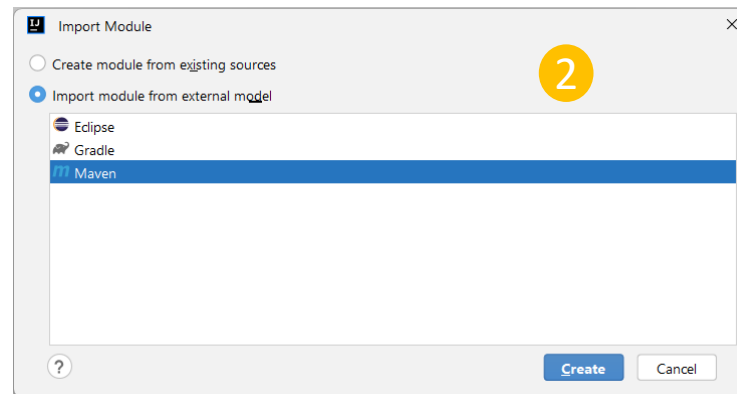
# Maven intelli-j project creation

- IntelliJ module wizzard calls mvn command like configured
- Uses settings as configured.
- Check advanced box for group and artefact names



# Import Maven Module into IntelliJ Project

- IntelliJ provides import as project or additional module into an IntelliJ project (different from maven project).
- Import wizard recognizes maven and can import from external maven model (2)



# Use maven commands/api

Use basic maven commands to get to know maven

- [mvn help:help -Ddetail=true](#)
- [maven-help-plugin](#) provides basic help
  - mvn help:describe -Dplugin=org.apache.maven.plugins:maven-war-plugin
  - mvn help:describe -Dcmd=install
- Show effective pom file: (includes also default super pom)
  - Shows resolved final variables, locations and plugins etc.
- mvn:evaluate
  - can resolve expressions like `${project.groupId}`

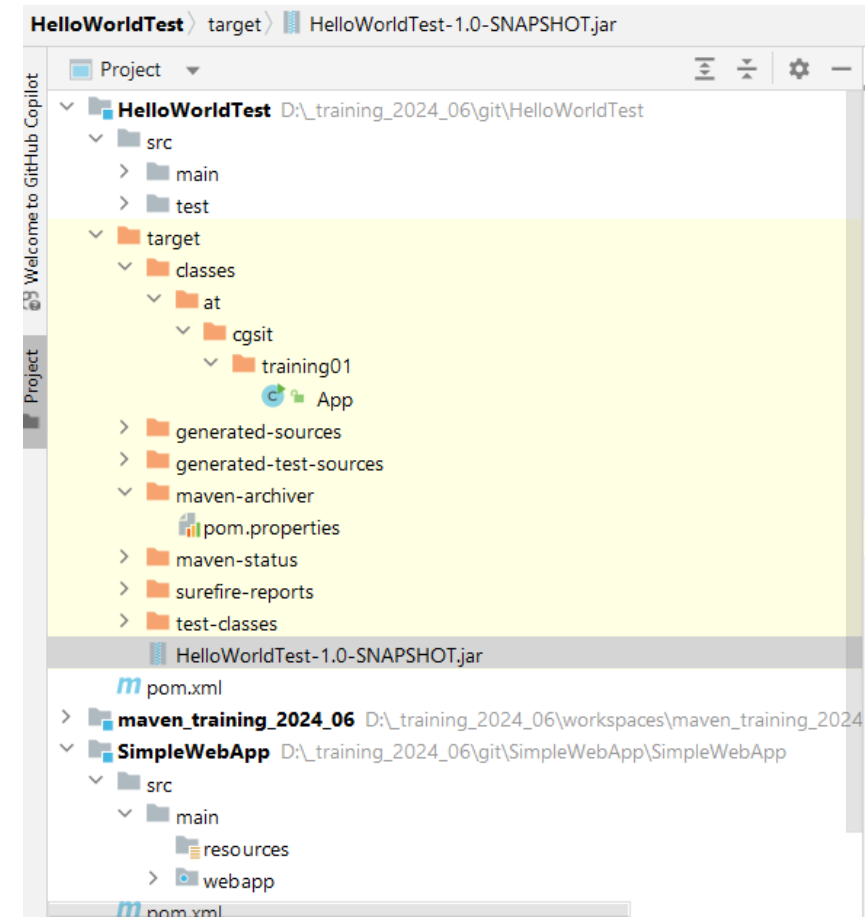
# Maven

## Build and testing



# Maven : simple compile, test and package

- Use maven command:
  - to compile, test and build the project jar file:
  - mvn package
- It compiles all java files, runs the generated test and creates the final output JAR file,
  - Including the version in the file name
- The JAR file itself also contains the pom file and a property file with the artefact attributes.
  - See Appendix: „Maven JAR file Metadata“
- The files in the target folder are temporary and MUST be ignored by .gitignore file
  - Will be deleted by mvn clean



# Maven : Jar Plugin

( [Apache Maven JAR Plugin](#) )

<https://maven.apache.org/shared/maven-archiver/index.html>

# Maven Dependency Management

# View current maven dependencies

1. `mvn dependency:tree -Dverbose`
2. `mvn dependency:resolve-plugins`
3. In IntelliJ view tab: show graphical dependency tree

# Maven Dependencies and Scope

- Dependencies to required libraries and frameworks are configured via dependencies
- Each required library is available via a central nexus artefact repository
- And has a group, an id and a VERSION
- Additionally it is imported for specific scopes

```
<dependencies>  
  <dependency>  
    <groupId>junit</groupId>  
    <artifactId>junit</artifactId>  
    <version>3.8.1</version>  
    <scope>test</scope>  
  </dependency>  
</dependencies>
```

# Maven dependency scopes

## 1. Compile

- **Description:** The default scope. Dependencies with this scope are available in all classpaths (compile, test, runtime) and included in the final artifact.

## 2. Provided

- **Description:** Dependencies are required for compiling and running the project but should not be included in the final artifact, as they are provided by the runtime environment.

## 3. Runtime

- **Description:** Dependencies are not required for compiling the project but are necessary for running it.

## 4. Test

- **Description:** Dependencies are only available during the test phase and are not included in the final artifact.

## 5. System

- **Description:** Dependencies that are provided explicitly by the user. The system path must be specified, and these dependencies are not looked up in remote repositories.

## 6. Import (for dependency management only)

- **Description:** This scope is used to import dependencies from BOM (Bill Of Materials) files. It allows managing versions of dependencies in multi-module projects.

# maven dependency tree example

- The tree view shows all dependencies for the current maven module
- The hierarchy indicates „ **Direct**“ and „ **Transitive**“ dependencies.
- „ **Transitive**“ refers to dependencies required by direct or other transitive dependencies.
- You can jump to the POM file of the Direct or transitive dependency to see and review it.

```
cgssc@DESKTOP-MAP12L0 MINGW64 /d/_training_2024_06/git/train_maven_24_06_basic_trainer (main)
$ mvn dependency:tree
[INFO] Scanning for projects...
[INFO] -----< at.cgsit.training01:train_maven_24_06 >-----
[INFO] Building train_maven_24_06 1.0-SNAPSHOT
[INFO] from pom.xml
[INFO] -----[ jar ]-----
[INFO] --- dependency:3.7.0:tree (default-cli) @ train_maven_24_06 ---
[INFO] at.cgsit.training01:train_maven_24_06:jar:1.0-SNAPSHOT
[INFO] +- org.apache.logging.log4j:log4j-api:jar:2.1:compile
[INFO] +- org.apache.logging.log4j:log4j-core:jar:2.23.1:compile
[INFO] +- org.junit.jupiter:junit-jupiter:jar:5.10.2:test
[INFO] | +- org.junit.jupiter:junit-jupiter-api:jar:5.10.2:test
[INFO] | | +- org.opentest4j:opentest4j:jar:1.3.0:test
[INFO] | | +- org.junit.platform:junit-platform-commons:jar:1.10.2:test
[INFO] | | \- org.apiguardian:apiguardian-api:jar:1.1.2:test
[INFO] | +- org.junit.jupiter:junit-jupiter-params:jar:5.10.2:test
[INFO] | \- org.junit.jupiter:junit-jupiter-engine:jar:5.10.2:test
[INFO] | \- org.junit.platform:junit-platform-engine:jar:1.10.2:test
[INFO] \- junit:junit:jar:3.8.1:test
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.315 s
[INFO] Finished at: 2024-06-23T16:47:19+02:00
[INFO] -----
```

```
cgssc@DESKTOP-MAP12L0 MINGW64 /d/_training_2024_06/git/train_maven_24_06_basic_trainer (main)
```

# Maven conflicting and duplicate dependency

- Using an old API package together with the newer log4j-core implementation
- leads to conflicts in the dependency tree because core includes newer api version as transitive dependency

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.1</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.23.1</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>5.10.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```



# Maven conflicting and duplicate dependency

- Mvn dependency:tree -Dverbose also lists the omitted libraries due to duplication (classpath only needs it once)
- Here we have explicitly specified the older log4j-api version 2.1 but using the core 2.23.1 which would depend on the API 2.23.1.
- It is omitted due to conflict, and the direct dependency is used because it is nearer.
- This of course can lead to compile or runtime issues, depending on the specific differences.
- We also have a possible issue here with JUNIT 5 and 4 .. But it is basically not visible because the classes and packages are exclusive.
  - But still this may lead to problems during runtime and or compile time.

```
cgssc@DESKTOP-MAP12LO MINGW64 /d/_training_2024_06/git/train_maven_24_06_basic_trainer (main)
$ mvn dependency:tree -Dverbose=true
[INFO] Scanning for projects...
[INFO]
[INFO] -----< at.cgsit.training01:train_maven_24_06 >-----
[INFO] Building train_maven_24_06 1.0-SNAPSHOT
[INFO] from pom.xml
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- dependency:3.7.0:tree (default-cli) @ train_maven_24_06 ---
[INFO] at.cgsit.training01:train_maven_24_06:jar:1.0-SNAPSHOT
[INFO] +- org.apache.logging.log4j:log4j-api:jar:2.1:compile
[INFO] +- org.apache.logging.log4j:log4j-core:jar:2.23.1:compile
[INFO] | \- (org.apache.logging.log4j:log4j-api:jar:2.23.1:compile - omitted for conflict with 2.1)
[INFO] +- org.junit.jupiter:junit-jupiter:jar:5.10.2:test
[INFO] | +- org.junit.jupiter:junit-jupiter-api:jar:5.10.2:test
[INFO] | | +- org.opentest4j:opentest4j:jar:1.3.0:test
[INFO] | | +- org.junit.platform:junit-platform-commons:jar:1.10.2:test
[INFO] | | \- (org.apiguardian:apiguardian-api:jar:1.1.2:test - omitted for duplicate)
[INFO] | \- org.apiguardian:apiguardian-api:jar:1.1.2:test
[INFO] +- org.junit.jupiter:junit-jupiter-params:jar:5.10.2:test
[INFO] | +- (org.junit.jupiter:junit-jupiter-api:jar:5.10.2:test - omitted for duplicate)
[INFO] | \- (org.apiguardian:apiguardian-api:jar:1.1.2:test - omitted for duplicate)
[INFO] \- org.junit.jupiter:junit-jupiter-engine:jar:5.10.2:test
[INFO] | +- org.junit.platform:junit-platform-engine:jar:1.10.2:test
[INFO] | | +- (org.opentest4j:opentest4j:jar:1.3.0:test - omitted for duplicate)
[INFO] | | +- (org.junit.platform:junit-platform-commons:jar:1.10.2:test - omitted for duplicate)
[INFO] | | \- (org.apiguardian:apiguardian-api:jar:1.1.2:test - omitted for duplicate)
[INFO] | +- (org.junit.jupiter:junit-jupiter-api:jar:5.10.2:test - omitted for duplicate)
[INFO] | \- (org.apiguardian:apiguardian-api:jar:1.1.2:test - omitted for duplicate)
[INFO] \- junit:junit:jar:3.8.1:test
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 1.205 s
```

# Dependency management solution via excludes for transitive dependencies

- A dependency configuration can be configured to exclude transitive dependencies by using exclude configurations.
- This solves the maven tree conflict ... “basically”.

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.1</version>
    <scope>compile</scope>
  </dependency>
  <!-- https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-core -->
  <!-- is the default scope. It tells Maven to package the dependency within the -->
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>2.23.1</version>
    <scope>compile</scope>
    <exclusions>
      <exclusion>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-api</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
</dependencies>
```

You, Moments ago • Uncommitted changes

# Manage dependencies via excludes

- Even if no conflict now is visible anymore, it does not mean that excluding transitive dependencies always solves the problem.
- In this case:
  - Dependency version conflict led to runtime misconfiguration. Log4J switched to SimpleLogger.
  - So error is partly invisible “worst case”

```
[INFO] -----  
[INFO] T E S T S  
[INFO] -----  
[INFO] Running at.cgsit.training01.HelloWorldTest  
ERROR StatusLogger Log4j2 could not find a logging implementation. Please add log4j-core to the classpath. Using SimpleLogger to log to the console...  
Hello World! test  
Hello World! test  
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.079 s -- in at.cgsit.training01.HelloWorldTest
```

```
[INFO] --- surefire:3.2.5:test (default-test) @ train_maven_24_06 ---  
[INFO] Using auto detected provider org.apache.maven.surefire.junitplatform.JUnitPlatformProvider  
[INFO] -----  
[INFO] T E S T S  
[INFO] -----  
[INFO] Running at.cgsit.training01.HelloWorldTest  
17:19:02.606 [main] INFO at.cgsit.training01.HelloWorld - Hello World! test  
Hello World! test  
17:19:02.622 [main] INFO at.cgsit.training01.HelloWorld - Hello World! test  
Hello World! test  
17:19:02.624 [main] WARN at.cgsit.training01.HelloWorld - No input string provided  
17:19:02.627 [main] WARN at.cgsit.training01.HelloWorld - No input string provided  
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.542 s -- in at.cgsit.training01.HelloWorldTest  
[INFO] Results:  
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
```

# Solution with extracted version variable

- The variable for the version can be extracted to a pom.property
- This can be used for all direct dependencies to be sure it is the same.
- Exclude might still make sence, since we are developing against the APIs and use the core for runtime only.
  - KISS Principle
- OK for simple pom.xml projects

```
<maven.compiler.source>11</maven.compiler.source>
<version.log4j>2.23.1</version.log4j>
</properties>

<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>${version.log4j}</version>
    <scope>compile</scope>
  </dependency>
  <!-- https://mvnrepository.com/artifact/org.apache.logging.log
  <!-- is the default scope. It tells Maven to package the deper
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-core</artifactId>
    <version>${version.log4j}</version>
    <scope>compile</scope>
    <exclusions>
      <exclusion>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-api</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
```

# Maven dependencyManagement

The `<dependencyManagement>` section

is used to specify the version of dependencies that are inherited by all child modules in a multi-module project.

It is not intended to include actual dependencies but rather to manage their versions. This helps ensure consistency across all modules.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.apache.logging.log4j</groupId>
      <artifactId>log4j-api</artifactId>
      <version>${version.log4j}</version>
    </dependency>
    <dependency>
      <groupId>org.apache.logging.log4j</groupId>
      <artifactId>log4j-core</artifactId>
      <version>${version.log4j}</version>
      <exclusions>
        <exclusion>
          <groupId>org.apache.logging.log4j</groupId>
          <artifactId>log4j-api</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
    <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter</artifactId>
      <version>5.10.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
  </dependency>
  <dependency>
```

# Maven dependency management enforced

**Dependency Convergence** is a concept in Maven that ensures all versions of a dependency used across a multi-module project converge to a single version.

Per default it is executed in the validate Phase

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-enforcer-plugin</artifactId>
  <version>3.0.0</version>
  <executions>
    <execution>
      <id>enforce-versions</id>
      <goals>
        <goal>enforce</goal>
      </goals>
      <configuration>
        <rules>
          <dependencyConvergence/>
          <!-- This rule checks that all dependencies are released -->
          <requireReleaseDeps />
        </rules>
        <fail>true</fail>
      </configuration>
      <!-- This specifies the phase -->
      <!-- validate is the default -->
      <phase>validate</phase>
    </execution>
  </executions>
</plugin>
</plugins>
```

You, 3 minutes ago • Uncommitted changes

# Aufgabe : 5 bis 10 min

- Neue apache common library dependency einfügen
- Und im source code verwenden
- Die version im dependency Management hinterlegen
- In der dependency ohne version inkludieren

<https://mvnrepository.com/artifact/org.apache.commons/commons-lang3>

```
<dependencyManagement>
  <dependencies>
    <!-- https://mvnrepository.com/artifact/org.apache.commons/commons-
lang3 -->
    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-lang3</artifactId>
      <version>3.14.0</version>
    </dependency>
```

# Chapter 03:Maven lifecycles

Maven lifecycles, plugins and goals and Maven Architecture

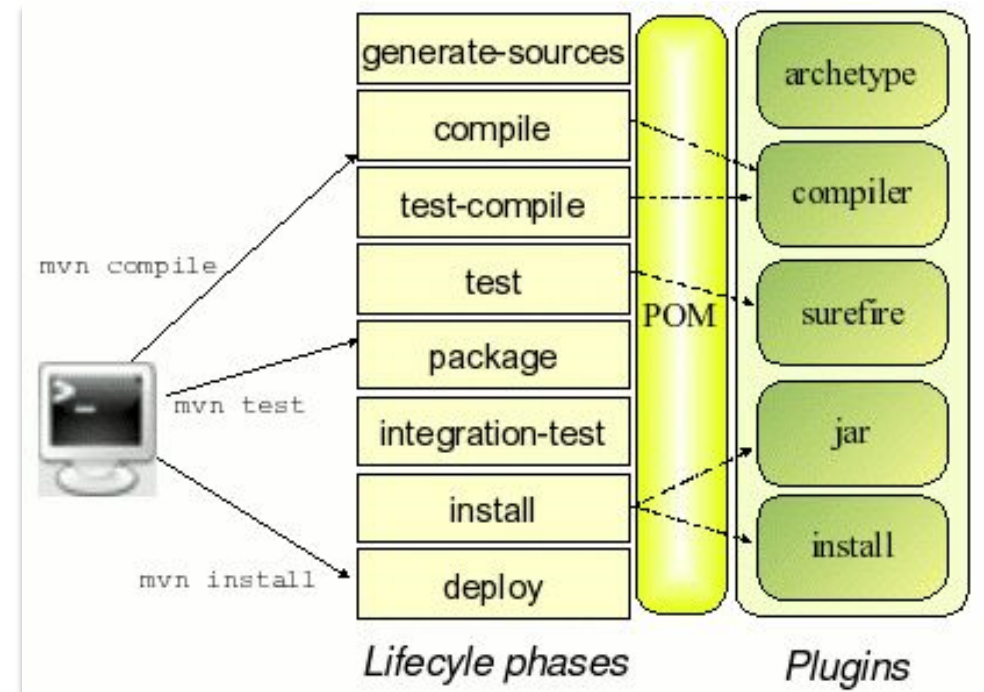


# Maven - Lifecycles

- Maven defines 3 preconfigured lifecycles
  - Clean
    - Is used to cleanup artefacts and the project. Mostly deleting files in the target directory
  - Default
    - Is the workhorse cycle doing the compile, test, package and deploy steps
  - Site
    - Is the lifecycle for performing and creating documentation and reports for the project
- Each Lifecycle is a sequence of phases performing a step in the lifecycle
- A Phase is a Step which is implemented by a maven plugin “goal”
- So goals are bound to phases to perform the actual work

# Maven – Lifecycle – Phases – Plugins and Goals

- Maven core defines the lifecycle phases
- A phase is a step in a lifecycle
- Configured or default plugins can extend maven to perform certain tasks for a phase
- Each plugin provides goals to be mapped to phases
- So the compiler plugin goal compile implements the lifecycle phase step compile.



# Detailed Default phases

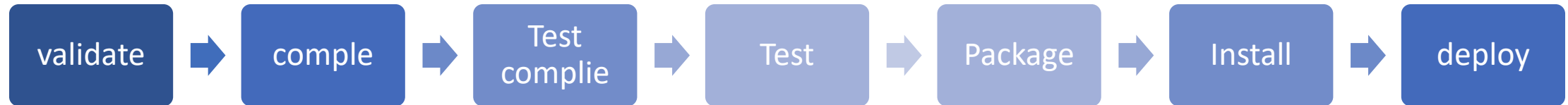
- This is the detailed list of all phases for the default lifecycle.

## ◀ Default Lifecycle¶

The default lifecycle handles project deployment and consists of the following phases:¶

1. → **validate**:¶
  - → validate: Validate the project structure and configuration.¶
2. → **initialize**:¶
  - → initialize: Initialize build state and set properties.¶
3. → **generate-sources**:¶
  - → generate-sources: Generate any source code for inclusion in compilation.¶
  - → process-sources: Process the source code, such as filtering properties.¶
  - → generate-resources: Generate resources, such as copying files into target directory.¶
  - → process-resources: Copy and process resources to the output directory.¶
4. → **compile**:¶
  - → compile: Compile the project's source code.¶
5. → **process-classes**:¶
  - → process-classes: Post-process compiled classes, such as bytecode enhancement.¶
6. → **generate-test-sources**:¶
  - → generate-test-sources: Generate test source code.¶
  - → process-test-sources: Process the test source code.¶
  - → generate-test-resources: Create resources needed for testing.¶
  - → process-test-resources: Copy and process test resources into the test output directory.¶
7. → **test-compile**:¶
  - → test-compile: Compile the test source code.¶
8. → **test**:¶
  - → test: Run the tests using a suitable unit testing framework.¶
9. → **package**:¶
  - → package: Package the compiled code into its distributable format, such as a JAR, WAR, or other.¶
10. → **pre-integration-test**:¶
  - → pre-integration-test: Perform actions needed before integration tests, such as setting up the environment.¶
11. → **integration-test**:¶
  - → integration-test: Process and deploy the package to a test environment and run integration tests.¶
12. → **post-integration-test**:¶
  - → post-integration-test: Perform actions needed after integration tests, such as cleaning up the environment.¶
13. → **verify**:¶
  - → verify: Run checks to verify the package is valid and meets quality criteria.¶
14. → **install**:¶
  - → install: Install the package into the local repository for use as a dependency in other projects locally.¶
15. → **deploy**:¶
  - → deploy: Copy the final package to the remote repository for sharing with other developers and projects.¶

# Maven commands and lifecycle



- You can invoke a specific phase on the lifecycle
  - ***mvn test***
- This will execute all phases until the specified one
- If there is an error and you fix it:
  - Second mvn test Run:Maven will check the timestamps of the files.
  - Only the modified files (in this case, the test class) will be recompiled.
  - Maven will then execute the tests again

# Maven skipping the tests

If the tests should not be done and just the source code should be compiled and the JAR/WAR file be created, we have several possibilities

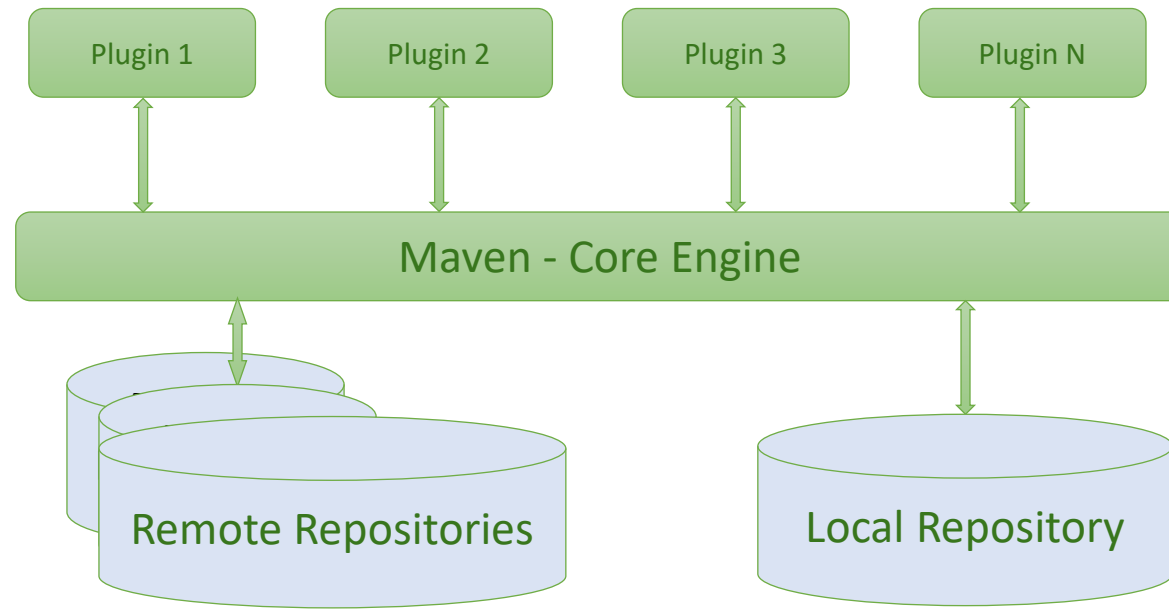
1. `mvn package -DskipTests`
  - or: `mvn -Dmaven.test.skip.exec=true`
2. `mvn package -Dmaven.test.skip=true`

## Why Use Each Option?

- `DskipTests`:  
Use this option when you want to ensure that your test classes are compiled and available (for example, to catch any compilation errors in test code) but you do not want to run the tests during this build.
- `-Dmaven.test.skip=true`:  
Use this option when you want to completely skip any test-related processes, which can save time when you are confident that the tests are not needed for this particular build.

# Maven Architecture: Plugins and Repositories

- Maven build artefacts are



# Maven – Plugin System

- Maven Available Plugins  
<https://maven.apache.org/plugins/index.html>

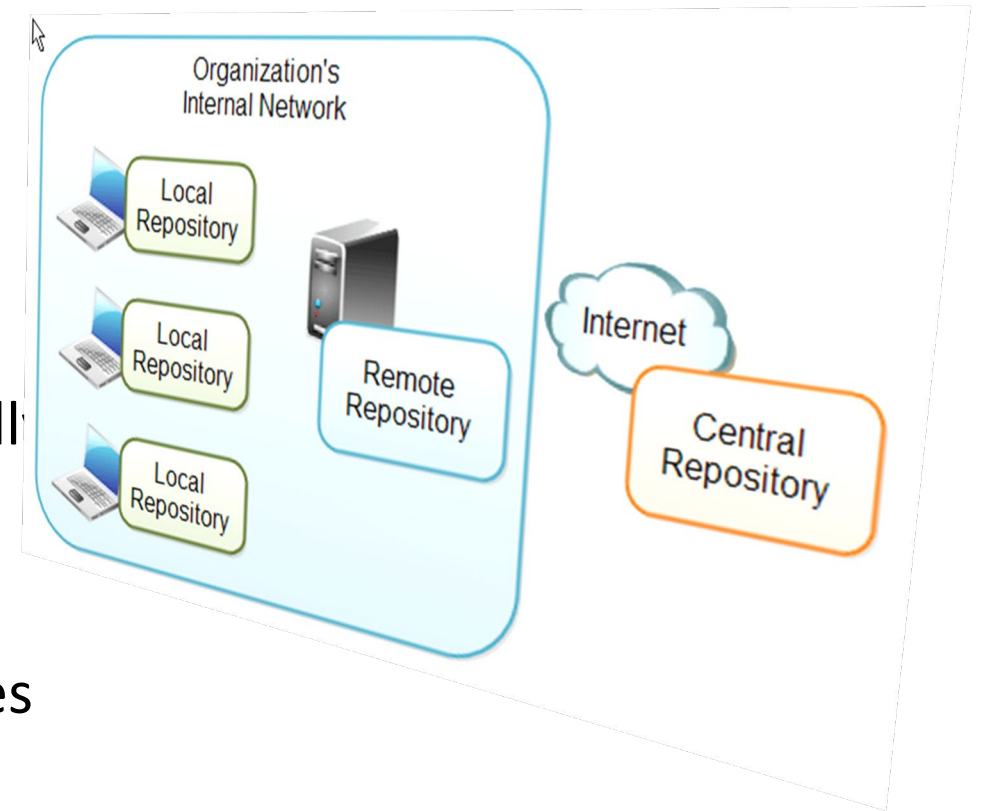
# Maven Plugins

- Provides the core operations to build your project.
  - E.g. to create a jar the maven jar plugin will do the job
  - So maven will delegate the work to plugins
- Plug-ins provides one or more “Goals”
- A Goal perform some operation on the project.
  - Ex: compile, create a Jar, deploy to Jboss, etc.
- Goals can be bound to build lifecycle phases



# Maven - Repositories

- **Maven Repositories:**  
**Essential for Java Project Management**
- **Local and Remote Repositories:**  
Enhance collaboration and consistency.
- Each user has a local repository used to store locally built artefacts and cache artefacts from remote repositories
- Remote repositories can proxy central repositories and provide the artefacts from there to the developer or Jenkins system



# Installing artefacts into repositories

- mvn install
  - Installs the built project into your local Maven repository.
- mvn deploy
  - Deploys the built project to a specified remote Maven repository.
  - Mostly done from Jenkins or CI/CD server itself

# SNAPSHOT and Release Versions

## Snapshot Repositories:

- Used for ongoing development.
- Versions are marked with -SNAPSHOT (e.g., 1.0-SNAPSHOT).
- Frequently updated with the latest changes.
- Ideal for development, testing, and integration purposes.

## •Release Repositories:

- Used for stable, final versions.
- Versions are without -SNAPSHOT (e.g., 1.0).
- Immutable and not frequently updated.
- Suitable for production and deployment.

## Key Points

- **Snapshots:** Use for iterative development and testing.
- **Releases:** Use for stable, production-ready versions

# Maven release process

A maven standard release is done in 2 steps:

## 1. mvn release:prepare

- # Follow the prompts to enter the release version and new development version
- Prompts for the release version (e.g., 1.0.0).
- Prompts for the new development version (e.g., 1.1.0-SNAPSHOT).
- Updates the pom.xml versions.
- Commits the changes and creates a tag in the version control system.

## 2. mvn release:perform

- # This will checkout the tag and deploy the release
- Checks out the tagged version.
- Builds the project from the tag.
- Deploys the built artifacts to the remote RELEASE repository

# SemVer: Semantic Versioning

Versioning system for software that uses a three-part number format:  
MAJOR.MINOR.PATCH.

Each part of the version number indicates a specific level of change:

1. **MAJOR** (version): Increased when there are incompatible changes
2. **MINOR**: Increased when functionality is added in a backwards-compatible manner
3. **PATCH**: Increased when backwards-compatible bug fixes are made

Example-Version: 2.1.3 indicates:

- MAJOR version 2, which might include breaking changes from version 1.x.x.
- MINOR version 1, which includes new features added since version 2.0.x.
- PATCH version 3, which includes bug fixes applied since version 2.1.0.

# Maven dependency - tree

- `Mvn dependency:tree -Dverbose -Dincludes=com.google.guava:guava`

Keeps the context where it came from in the tree

- Java class loader will search for a class and will load the FIRST found class in the classpath, by taking one jar dependency after the other
- Maven: takes nearest dependency used in the tree to select it,
  - The first and nearest will win
  - Maven does not pick the newest one
  - Maven does not understand compatibility issues within APIs ..
  - Maven does not understand Semver

# Maven Profiles

- Profiles are configured in xml also
- They can be active or not. And have a default.
- They can be activated via profile maven command switch  
-P<profilename>
- They can include custom properties
- But also can specifically configure plugins and

```
<!-- profiles enable different configurations and settings  
<profiles>  
  <profile>  
    <id>no-tests</id>  
    <properties>  
      <maven.test.skip>true</maven.test.skip>  
    </properties>  
    <activation>  
      <activeByDefault>false</activeByDefault>  
    </activation>  
  </profile>  
</profiles>
```

# Profiles to specifically configure plugins

- Profiles can specifically configure plugins where the plugin configuration is included in the profile configuration
- A standard example therefore are “development” and “production” profiles to provide specific configurations

Usage:

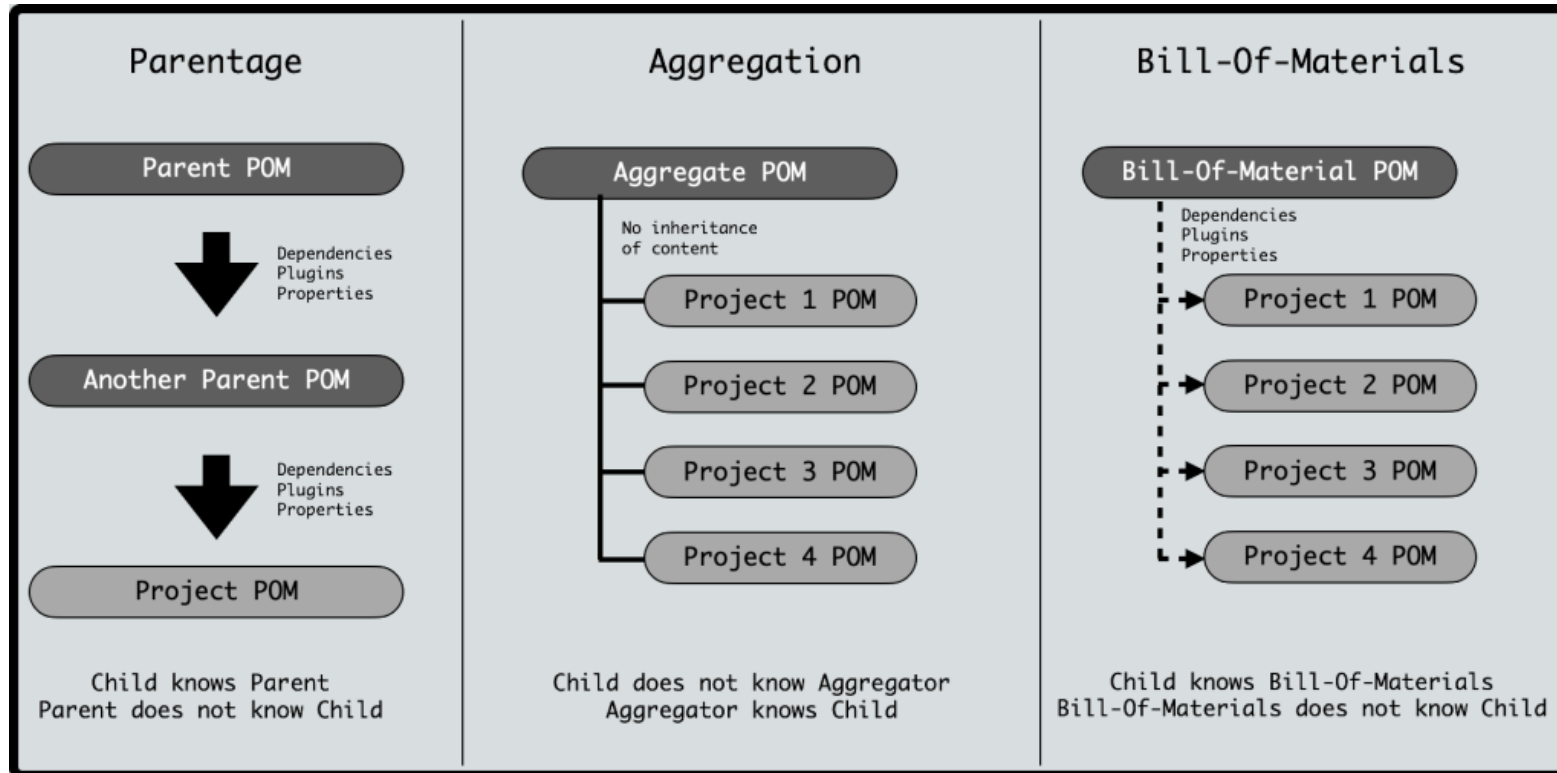
```
mvn clean package -Pno-tests
```



# Maven

## Multi module projects

# Maven Multi Modul Varianten



# Maven: Bill of Materials

- Used to collect a defined list of dependencies to be included in other poms

# Maven Parent Pom

- A project can be split into multiple modules with a common parent pom.xml
- The parent pom can again have a company parent pom
- mvn clean install -N
- With the -N or --non-recursive option only the parent can be built

# Maven: bidirectional dependency error

- Creating new project and move all common classes.  
This is the first preferred option.

```
[WARNING]
[WARNING] Some problems were encountered while building the effective model for at.cgsit.training:maven-training-dtos:jar:1.0-SNAPSHOT
[WARNING] 'dependencies.dependency.(groupId:artifactId:type:classifier)' must be unique: io.quarkus:quarkus-resteasy-reactive-jackson:jar -> duplicate declaration
version (?) @ line 26, column 21
[WARNING]
[WARNING] It is highly recommended to fix these problems because they threaten the stability of your build.
[WARNING]
[WARNING] For this reason, future Maven versions might no longer support building such malformed projects.
[WARNING]
[ERROR] [ERROR] The projects in the reactor contain a cyclic reference: Edge between 'Vertex{label='at.cgsit.training:maven-training-restapi:1.0-SNAPSHOT'}' and
'Vertex{label='at.cgsit.training:maven-training-dtos:1.0-SNAPSHOT'}' introduces to cycle in the graph at.cgsit.training:maven-training-dtos:1.0-SNAPSHOT -->
at.cgsit.training:maven-training-restapi:1.0-SNAPSHOT --> at.cgsit.training:maven-training-dtos:1.0-SNAPSHOT @
[ERROR] The projects in the reactor contain a cyclic reference: Edge between 'Vertex{label='at.cgsit.training:maven-training-restapi:1.0-SNAPSHOT'}' and
'Vertex{label='at.cgsit.training:maven-training-dtos:1.0-SNAPSHOT'}' introduces to cycle in the graph at.cgsit.training:maven-training-dtos:1.0-SNAPSHOT -->
at.cgsit.training:maven-training-restapi:1.0-SNAPSHOT --> at.cgsit.training:maven-training-dtos:1.0-SNAPSHOT -> [Help 1]
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the -e switch.
[ERROR] Re-run Maven using the -X switch to enable full debug logging.
[ERROR]
```

# Blick über den Tellerrand



(Bild: Shutterstock)

# Maven - References

1. [Apache Maven Guides](#)
2. [Top-5-apache-maven-free-ebooks-for-java](#)
3. [Apache Maven tutorialspoint](#)
4. [Maven by example](#)
5. [SemVer - Semantic Versioning](#)
6. <https://maven.apache.org/articles.html>
7. <https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle> Reference
8. <https://medium.com/@TimvanBaarsen/maven-cheat-sheet-45942d8c0b86>

# Appendix

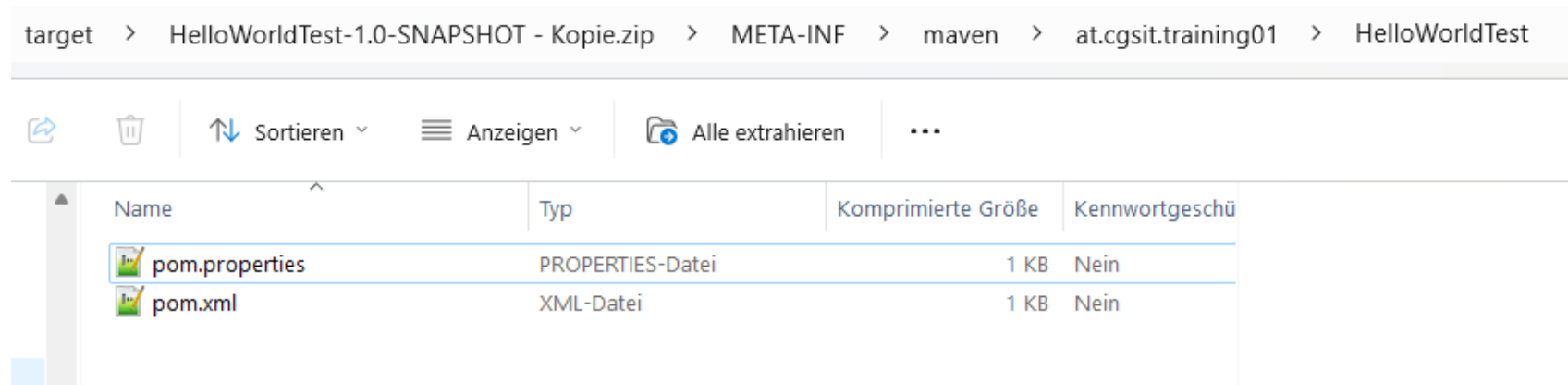









# Dependency Management Issues

- Scope → take care .. Test if p
- Maven enforcer plugin
  - dependencyConvergence
  - requiresUpperBound

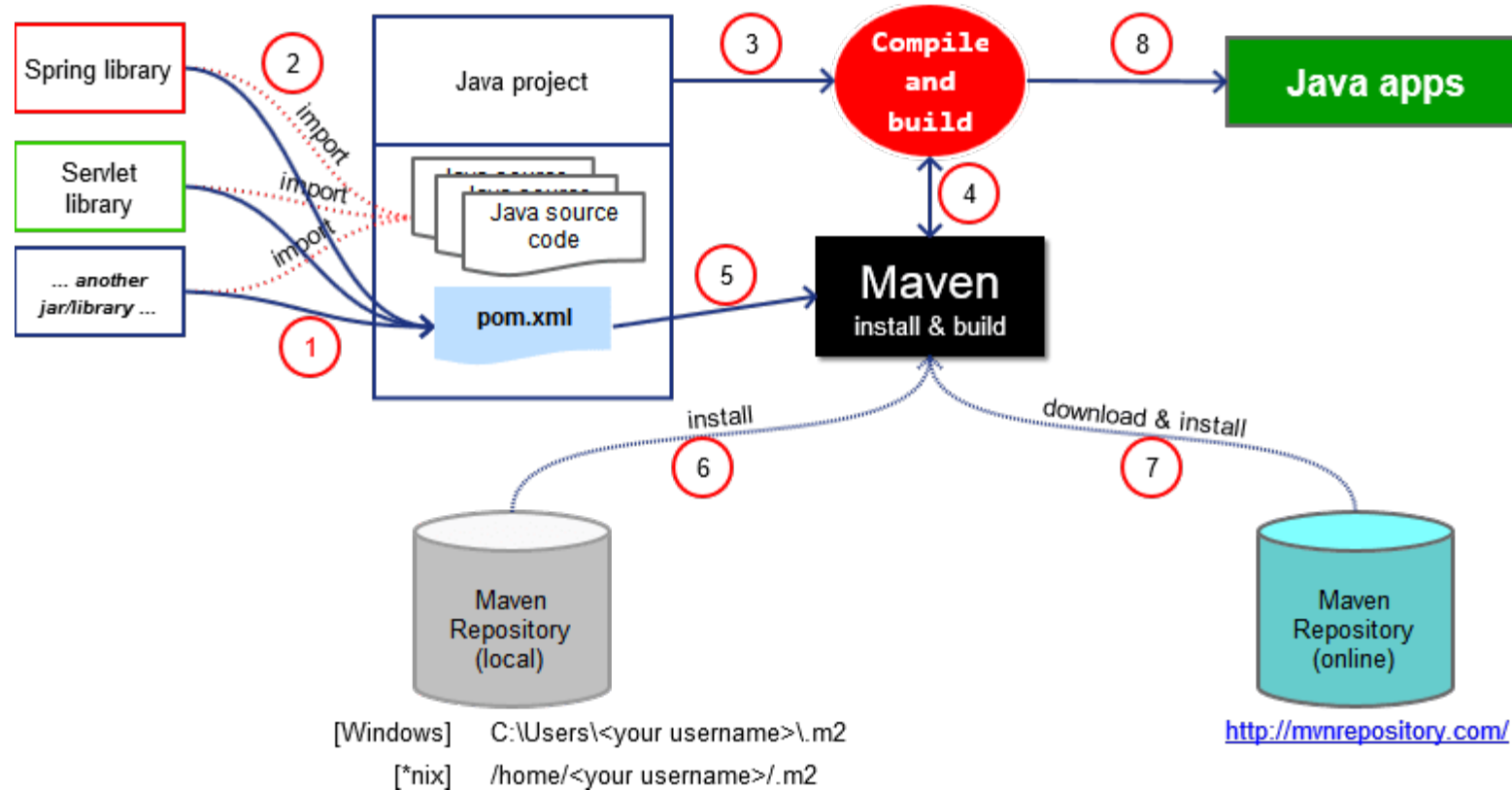
# Maven JAR file Metadata

- Placed in JAR META-INF/maven Folder
- can be used for information purposes



target > HelloWorldTest-1.0-SNAPSHOT - Kopie.zip > META-INF > maven > at.cgsit.training01 > HelloWorldTest				
   Sortieren ▾  Anzeigen ▾  Alle extrahieren ...				
Name	Typ	Komprimierte Größe	Kennwortgeschü	
 pom.properties	PROPERTIES-Datei	1 KB	Nein	
 pom.xml	XML-Datei	1 KB	Nein	

# Maven how it works



Danke für Ihre Aufmerksamkeit