Application Delivery Fundamentals 2.0 B: Java

Introduction to Maven

High performance. Delivered.
consulting | technology | outsourcing

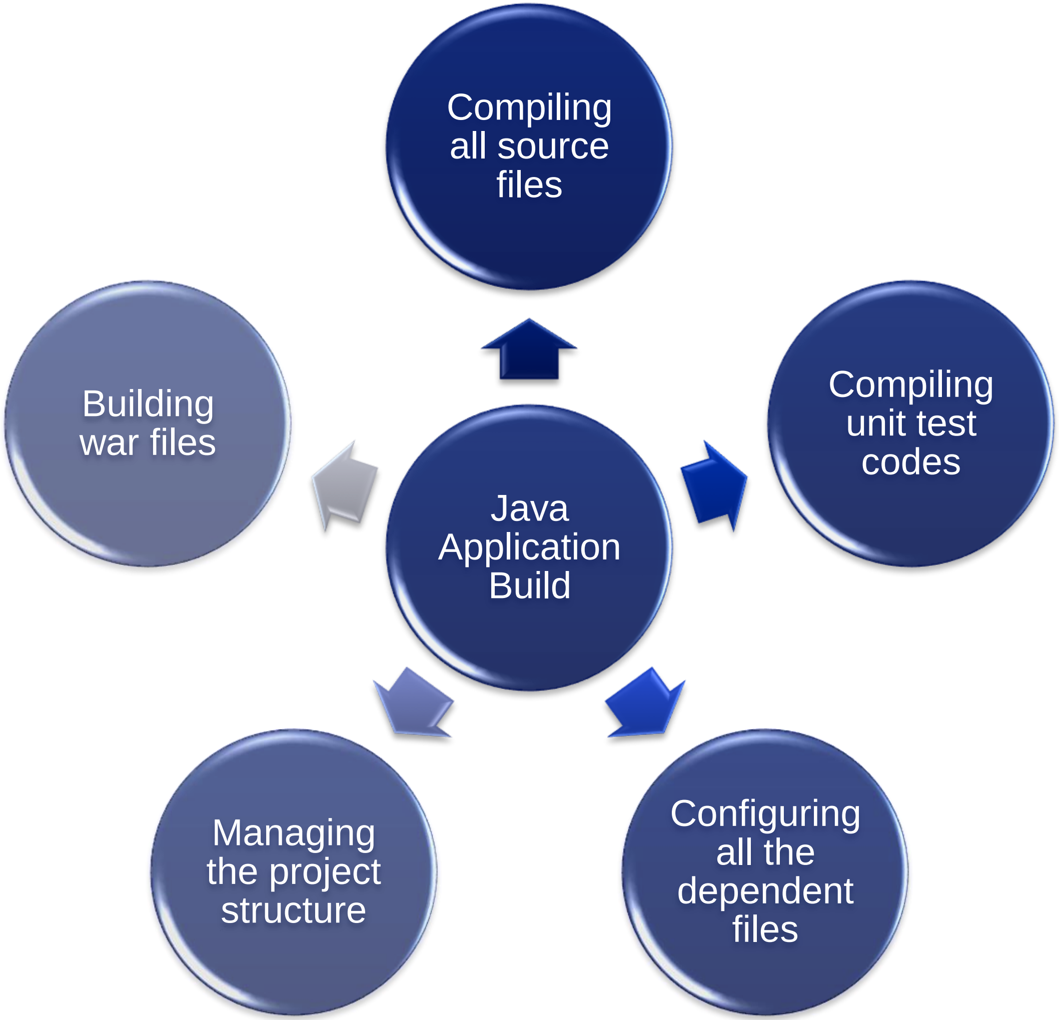
Course Goals / Objectives

- At the end of this module, participants will be able to understand:
  - The Physical Overview of Maven3
  - The Maven repositories
  - Maven Archetypes
  - Maven 3 life cycle phases
  - How to set the environment for Maven 3

Course Goals / Objectives

- At the end of this module, participants will be able to understand:
  - The pom.xml file
  - How to customize the pom.xml file
  - How to create Eclipse related files using Maven
  - How to create a WAR file

Java Application Build Steps



Java Application Build

Compiling all source files

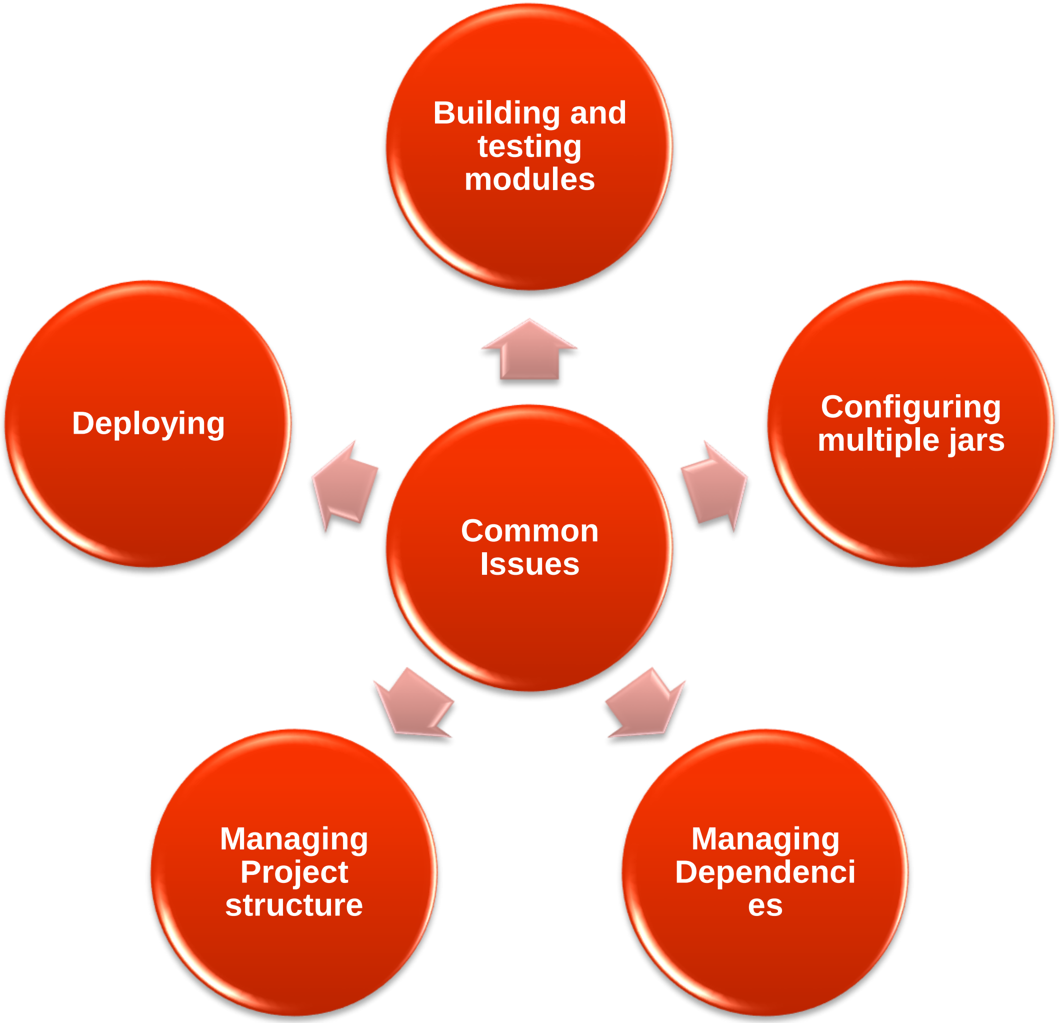
Compiling unit test codes

Configuring all the dependent files

Managing the project structure

Building war files

Issues



Building and testing modules

Deploying

Configuring multiple jars

Common Issues

Managing Project structure

Managing Dependenci es

What is Maven ?

- Build tool
- A software project management and comprehension tool.
- It can manage the build of project, documentation and reporting from a central piece of information
- Maven is hosted by Apache Software Foundation

What is Maven ?

- Maven is a project management tool which encompasses a project object model.
- a set of standards, a project lifecycle, a dependency management system, and logic for executing plugin goals at defined phases in a lifecycle.

What is Maven ?

- Maven uses POM (Project Object Model) in order to describe the software project which is being built, it’s dependencies on other external modules and build order.

What is Maven ?

- Maven comes with predefined targets for performing certain tasks like compilation of code and it’s packaging.
- Maven is network-ready. The core engine can dynamically download plug-ins from a repository.

Apache Ant vs Maven

- Apache Ant
- Ant doesn’t have formal conventions like a common project directory structure or default behaviour.
- You have to tell Ant exactly where to find the source and where to put the output.
- Informal conventions have emerged over time, but they haven’t been codified into the product.
- Ant is procedural. You have to tell Ant exactly what to do and when to do it.
- You have to tell it to compile, then copy, then compress.
- Ant doesn’t have a lifecycle. You have to define goals and goal dependencies.
- You have to attach a sequence of tasks to each goal manually.

Apache Ant vs Maven

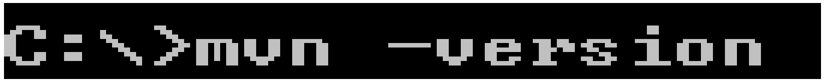
- Apache Maven
- Maven has conventions. It knows where your source code is because you followed the convention.
- Maven’s Compiler plugin put the bytecode in target/classes, and it produces a JAR file in target.
- Maven is declarative. All you had to do was create a pom.xml file and put your source in the default directory. Maven took care of the rest.
- Maven has a lifecycle which was invoked when you executed mvn install.
- This command tells Maven to execute a series of sequential lifecycle phases until it reached the install lifecycle phase.
- As a side effect of this journey through the lifecycle, Maven executed a number of default plugin goals which did things like compile and create a JAR.

Setting the environment for Maven 3.6.3

- Download Maven3 from **https://mirrors.estointernet.in/apache/**
- Unzip the installation archive to E:/
- Set the M2_HOME and Path environment variables in the following way:
  - M2_HOME=E:\software\A08\file\apache-maven-3.6.1-bin\apache-maven-3.6.1
  - Path=%M2_HOME%\bin

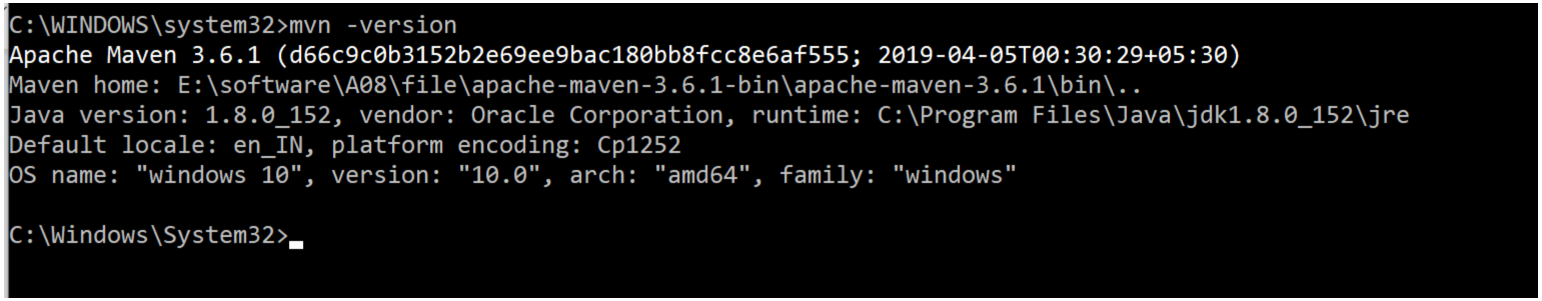
Setting the environment for Maven 2

- If installed correctly, you should be able to test it by opening a command prompt and typing :



```
C:\>mvn -version
```

- The output should be like this:



```
C:\WINDOWS\system32>mvn -version
Apache Maven 3.6.1 (d66c9c0b3152b2e69ee9bac180bb8fcc8e6af555; 2019-04-05T00:30:29+05:30)
Maven home: E:\software\A08\file\apache-maven-3.6.1-bin\apache-maven-3.6.1\bin\..
Java version: 1.8.0_152, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk1.8.0_152\jre
Default locale: en_IN, platform encoding: Cp1252
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"

C:\Windows\System32>_
```

User-Specific Configuration and Repository

- ~/.m2/settings.xml
- A file containing user-specific configuration for authentication, repositories, and other information to customize the behaviour of Maven.
- ~/.m2/repository/
- This directory contains your local Maven repository. When you download a dependency from a remote Maven repository, Maven stores a copy of the dependency in your local repository.

Maven Settings File

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
                        http://maven.apache.org/xsd/settings-1.0.0.xsd">
    <localRepository/>
    <interactiveMode/>
    <usePluginRegistry/>
    <offline/>
    <pluginGroups/>
    <servers/>
    <mirrors/>
    <proxies/>
    <profiles/>
    <activeProfiles/>
</settings>
```

Maven Settings File

ELEMENT NAME	DESCRIPTION
localRepository	Maven stores copies of plug-ins and dependencies locally in the C:\Users\<your_user_name>\.m2\repository folder. This element can be used to change the path of the local repository.
interactiveMode	As the name suggests, when this value is set to true, the default value, Maven interacts with the user for input.
usePluginRegistry	It decide that if Maven should use the ${user.home}/.m2/plugin-registry.xml file to manage plugin versions. Its default value is false.
offline	When set to true, this configuration instructs Maven to operate in an offline mode. The default is false.

Maven Settings File

pluginGroups	It contains a list of pluginGroup elements, each contains a groupId. The list is searched when a plugin is used and the groupId is not provided in the command line. This list automatically contains org.apache.maven.plugins and org.codehaus.mojo.
servers	Maven can interact with a variety of servers, such as Apache Subversion (SVN) servers, build servers, and remote repository servers. This element allows you to specify security credentials, such as the username and password, which you need to connect to those servers.
mirrors	As the name suggests, mirrors allow you to specify alternate locations for your repositories.
proxies	proxies contains the HTTP proxy information needed to connect to the Internet.
profiles	profiles allow you to group certain configuration elements, such as repositories and pluginRepositories.
activeProfile	The activeProfile allows you to specify a default profile to be active for Maven to use.

Settings.xml

- Indicates if maven should interact with the user for input. Is a true/false field. Defaults to true.
- This option can be useful when we want to create an empty and default java project.
- mvn archetype:generate -DgroupId=com.example -DartifactId=DemoJavaCodeGeeks -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false

Settings.xml

- usePluginRegistry
- There is a file called plugin-registry.xml in ${user.home}/.m2 folder. This field indicates if maven should use that file to manage plugins versions. Defaults to false.
- The Maven 2 plugin registry (~/.m2/plugin-registry.xml) is a mechanism to help the user exert some control over their build environment.
- Rather than simply fetching the latest version of every plugin used in a given build, this registry allows the user to get a plugin to a particular version, and only update to newer versions under certain restricted circumstances.
- There are various ways to configure or bypass this feature, and the feature itself can be managed on either a per-user or global level.

Settings.xml

- offline
- Indicates if maven should work in offline mode, this is, maven can not connect to remote servers. Defaults to false.

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <localRepository>${user.home}/.m2/repository</localRepository>
  <interactiveMode>true</interactiveMode>
  <usePluginRegistry>false</usePluginRegistry>
  <offline>false</offline>
</settings>
```

Settings.xml

- PluginGroups
- The puglingGroup accepts multiples values, when a plugin is invoked, maven will search along this element in order to find the groupId for the plugin.
- It makes more easy the maven execution.
- You can define several plugins groupId, by default, it contains the following ones:
- org.apache.maven.plugins
- org.codehaus.mojo

Settings.xml

- <pluginGroups>
- <pluginGroup>org.mortbay.jetty</pluginGroup>
- <pluginGroup>your.own.plugin.groupId</pluginGroup>
- </pluginGroups>

Settings.xml

Servers

```
 <servers>
    <server>
      <id>server_repo_java_code_geeks</id>
      <username>john</username>
      <password>doeIsMyPass</password>
      <privateKey>${user.home}/.ssh/dsa_key</privateKey>
      <passphrase>my_passphrase</passphrase>
      <filePermissions>774</filePermissions>
      <directoryPermissions>775</directoryPermissions>
      <configuration></configuration>
    </server>
```

Settings.xml

```
<server>
    <id>server_repo_java_code_geeks_2</id>
    <username>steve</username>
    <password>steve_password</password>
    <privateKey>${user.home}/.ssh/id_dsa</privateKey>
    <passphrase>steve_passphrase</passphrase>
    <filePermissions>664</filePermissions>
    <directoryPermissions>775</directoryPermissions>
    <configuration></configuration>
  </server>
</servers>
```

Settings.xml

Mirrors

```
<mirrors>
  <mirror>
   <id>centralmirror</id>
   <name>Apache maven central mirror Spain</name>
   <url>http://downloads.centralmirror.com/public/maven</url>
   <mirrorOf>maven_central</mirrorOf>
  </mirror>
  <mirror>
   <id>jcg_mirror</id>
   <name>Java Code Gueeks Mirror Spain</name>
   <url>http://downloads.jcgmirror.com/public/jcg</url>
   <mirrorOf>javacodegeeks_repo</mirrorOf>
  </mirror>
 </mirrors>
```

Settings.xml

Proxies

```
<proxies>
  <proxy>
   <id>jcg_proxy</id>
   <active>true</active>
   <protocol>http</protocol>
   <host>proxy.javacodegueeks.com</host>
   <port>9000</port>
   <username>proxy_user</username>
   <password>user_password</password>
   <nonProxyHosts>*.google.com|javacodegueeks.com</nonProxyHosts>
  </proxy>
 </proxies>
```

Settings.xml

Profiles

```
 <profiles>
   <profile>
    <id>test</id>
    <activation>
     <activeByDefault>false</activeByDefault>
     <jdk>1.6</jdk>
     <os>
      <name>Windows XP</name>
      <family>Windows</family>
      <arch>x86</arch>
      <version>5.1.3200</version>
     </os>
```

Settings.xml

Profiles

```
 <property>
     <name>mavenVersion</name>
     <value>3.0.3</value>
    </property>
    <file>
     <exists>${basedir}/windows.properties</exists>
     <missing>${basedir}/windows_endpoints.properties</missing>
    </file>
   </activation>

   <properties>
    <user.project>${user.home}/your-project</user.project>
    <system.jks>${user.home}/your_jks_store</system.jks>
   </properties>
```

Settings.xml

Profiles

```
<repositories>
        <repository>
          <id>codehausSnapshots</id>
          <name>Codehaus Snapshots</name>
          <releases>
            <enabled>false</enabled>
            <updatePolicy>always</updatePolicy>
            <checksumPolicy>warn</checksumPolicy>
          </releases>
          <snapshots>
            <enabled>true</enabled>
            <updatePolicy>never</updatePolicy>
            <checksumPolicy>fail</checksumPolicy>
          </snapshots>
```

Settings.xml

Profiles

```
 <url>http://snapshots.maven.codehaus.org/maven2</url>
      <layout>default</layout>
     </repository>
    </repositories>

    <pluginRepositories>
     <pluginGroup>your.own.plugin.groupId</pluginGroup>
    </pluginRepositories>

   </profile>
  </profiles>
```

Settings.xml

Active Profile

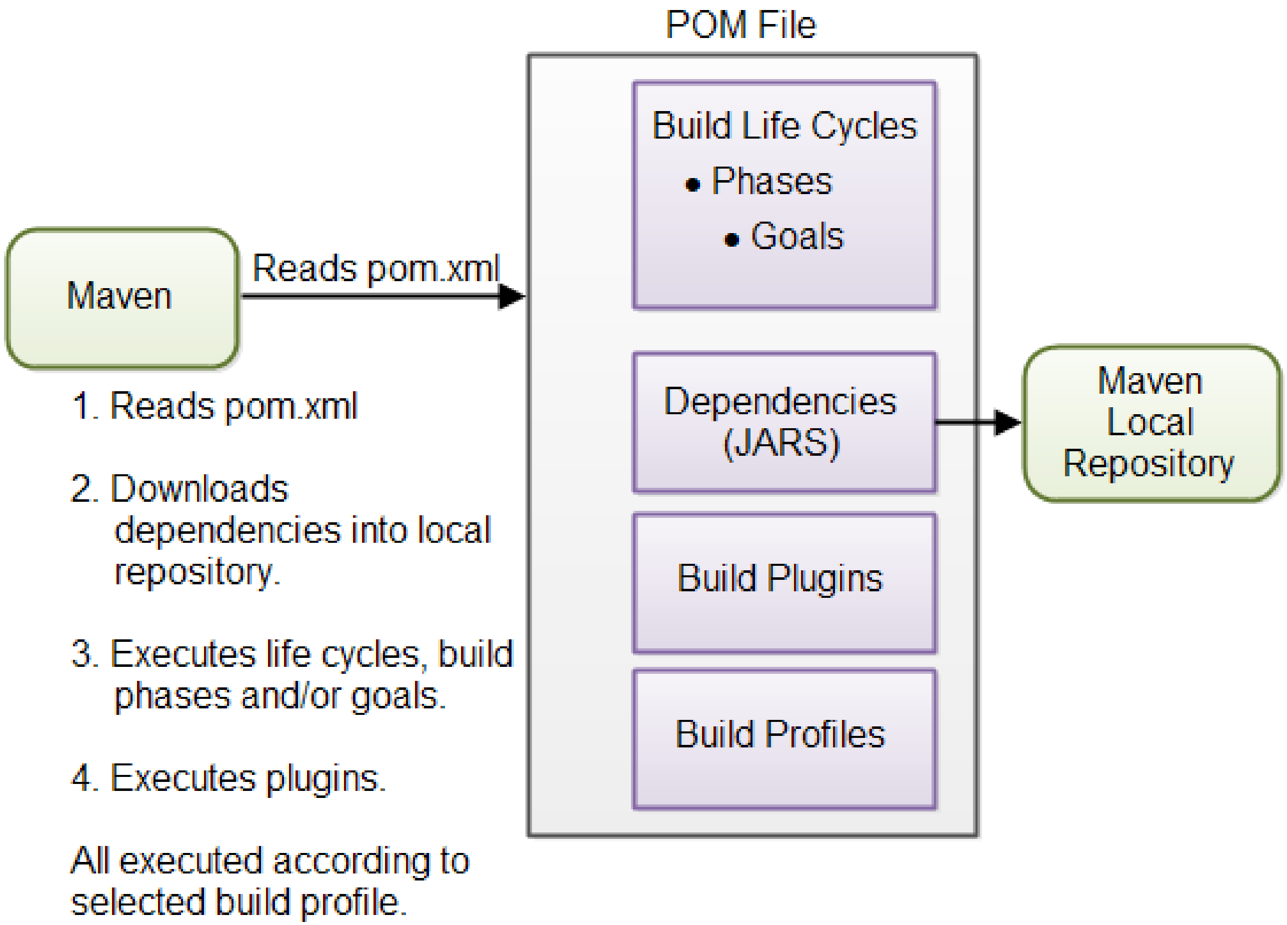
```
 <activeProfiles>
    <activeProfile>test</activeProfile>
  </activeProfiles>
```

```
 $mvn package -P prod
$ mvn package -D env=prod
```

Maven Key terms

- Archetype - is a template of a project which is combined with some user input to produce a working Maven project that has been tailored to the user's requirements
- POM - The pom.xml file is the core of a project's configuration in Maven. It is a single configuration file that contains the majority of information required to build a project.
- Dependencies - Maven allows projects to declare what dependencies they have, and will automatically materialize those dependencies

Physical Overview of Maven



1. Reads pom.xml

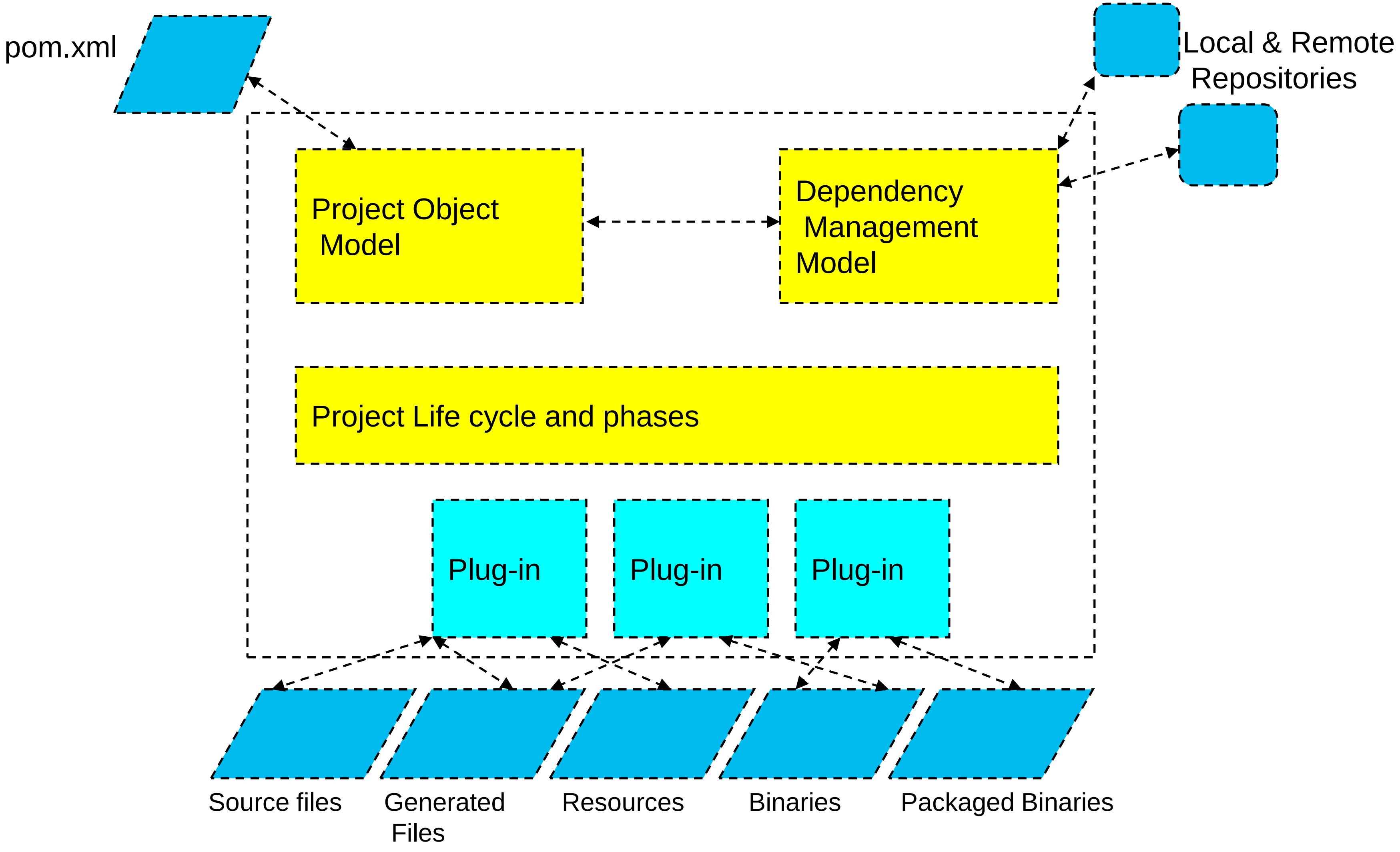
2. Downloads dependencies into local repository.

3. Executes life cycles, build phases and/or goals.

4. Executes plugins.

All executed according to selected build profile.

Physical Overview of Maven



What is Maven Build Lifecycle?

- The sequence of steps which is defined in order to execute the tasks and goals of any maven project is known as build life cycle in maven.
- Maven 2.0 version is basically a build life cycle oriented and clearly says that these steps are well defined to get the desired output after the successful execution of the build life cycle.

Maven Life Cycle

- Maven comes with 3 built-in build life cycles as shown below :
- **Clean** - this phase involves cleaning of the project (for a fresh build & deployment)
- **Default** - this phase handles the complete deployment of the project
- **Site** - this phase handles the generating the java documentation of the project.

Clean Life Cycle

- When we execute *mvn post-clean* command, Maven invokes the clean lifecycle consisting of the following phases.
- pre-clean
- clean
- post-clean

- The **clean: clean** goal performs the deletion of the target directory which is the actual build directory.

Maven: Build Life Cycle of clean phase

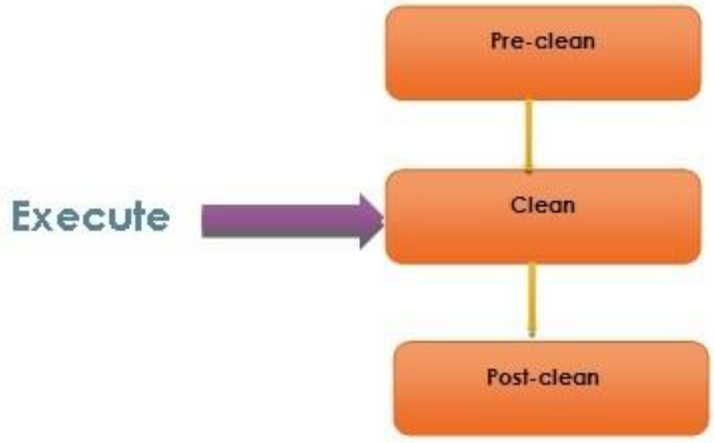
- Clean phase is used to clean up the project and make it ready for the fresh compile and deployment.
- The command used for the same is mvn post-clean.
- When mvn post-clean command is invoked, maven executes the below tasks via executing the below commands internally :
- mvn pre-clean
- mvn clean
- mvn post-clean
- This maven's clean is a goal and on executing it cleans up the output directory (target folder) by deleting all the compiled files.

Maven Life Cycle

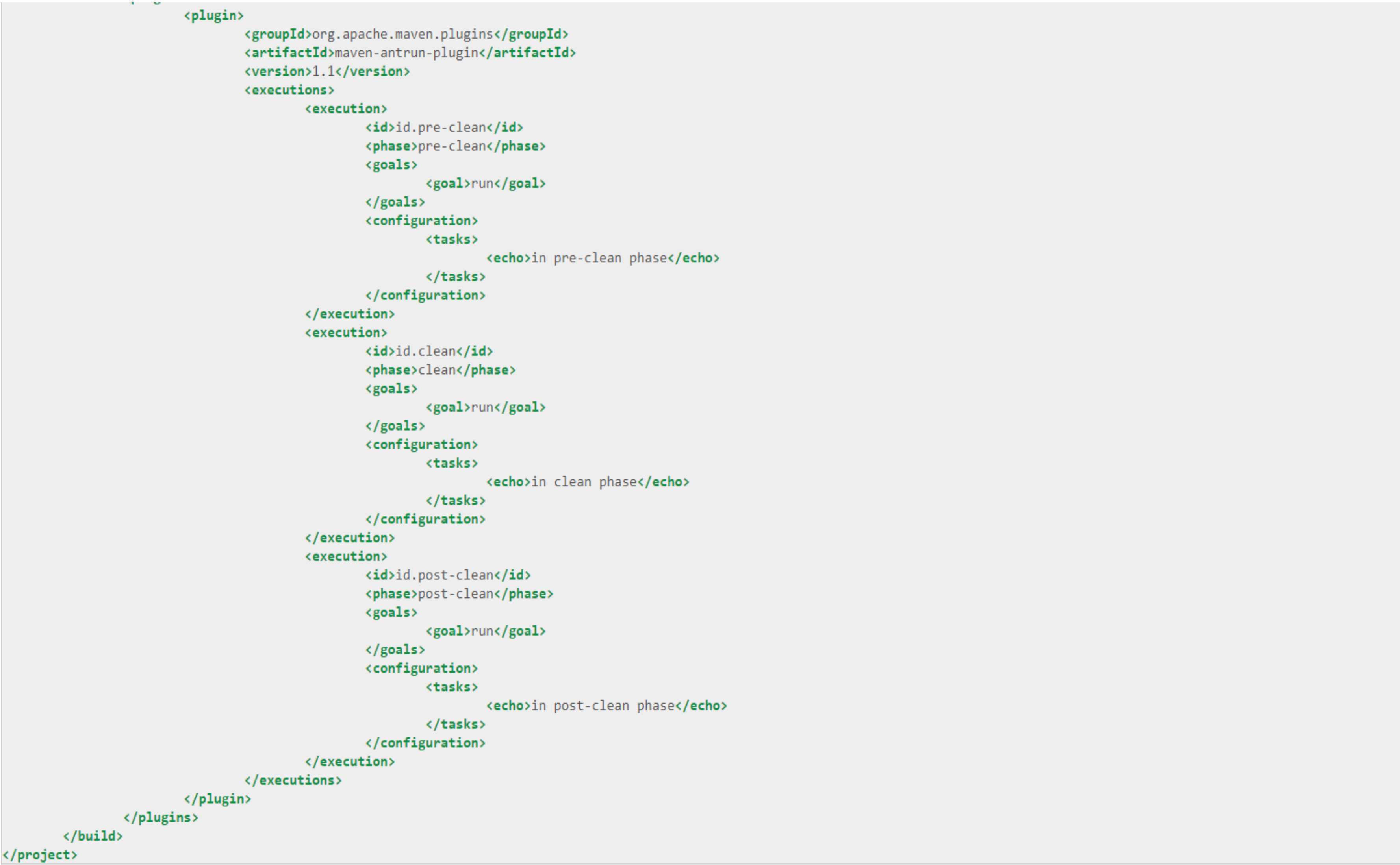
- Calling one phase of the clean lifecycle results in the execution of all phases up to an including that phase.
- So, if we perform a "mvn clean", it will execute the pre-clean and the clean phases.
- If we perform a "mvn post-clean", it will execute the pre-clean, clean, and post-clean phases.
- The pre-clean phase can be used for any tasks required prior to cleanup.
- The post-clean phase can be used for tasks following the cleanup.

Maven: Build Life Cycle of clean phase

- Whenever a maven command for any life cycle is invoked, maven executes the phases till and up to the invoked phase.
- E.g. when 'mvn clean' is invoked, maven will execute only the phase clean.
- But, no compile/deployment/site phase is invoked.

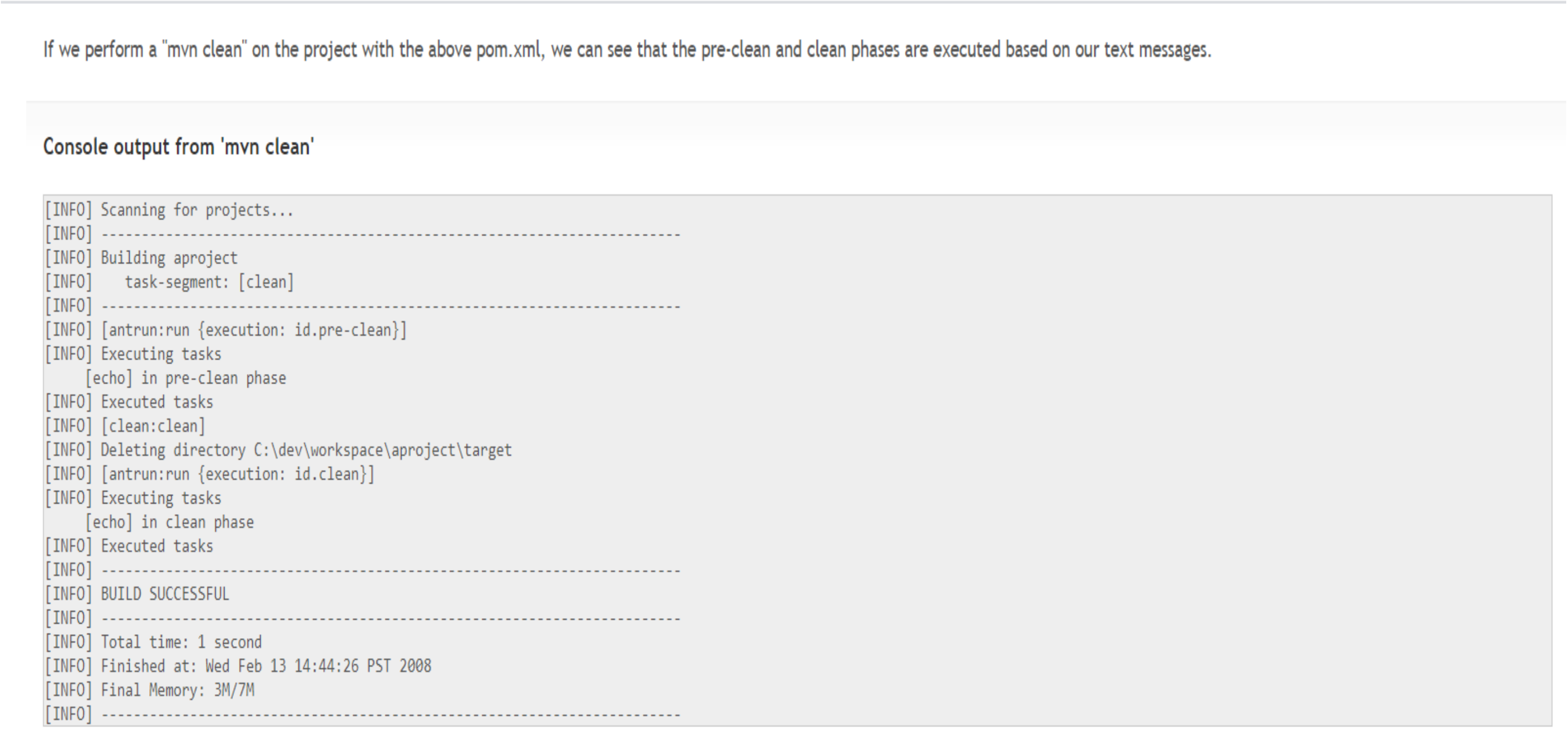


Maven: Build Life Cycle of clean phase



```
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-antrun-plugin</artifactId>
                <version>1.1</version>
                <executions>
                    <execution>
                        <id>id.pre-clean</id>
                        <phase>pre-clean</phase>
                        <goals>
                            <goal>run</goal>
                        </goals>
                        <configuration>
                            <tasks>
                                <echo>in pre-clean phase</echo>
                            </tasks>
                        </configuration>
                    </execution>
                    <execution>
                        <id>id.clean</id>
                        <phase>clean</phase>
                        <goals>
                            <goal>run</goal>
                        </goals>
                        <configuration>
                            <tasks>
                                <echo>in clean phase</echo>
                            </tasks>
                        </configuration>
                    </execution>
                    <execution>
                        <id>id.post-clean</id>
                        <phase>post-clean</phase>
                        <goals>
                            <goal>run</goal>
                        </goals>
                        <configuration>
                            <tasks>
                                <echo>in post-clean phase</echo>
                            </tasks>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

Maven: Build Life Cycle of clean phase

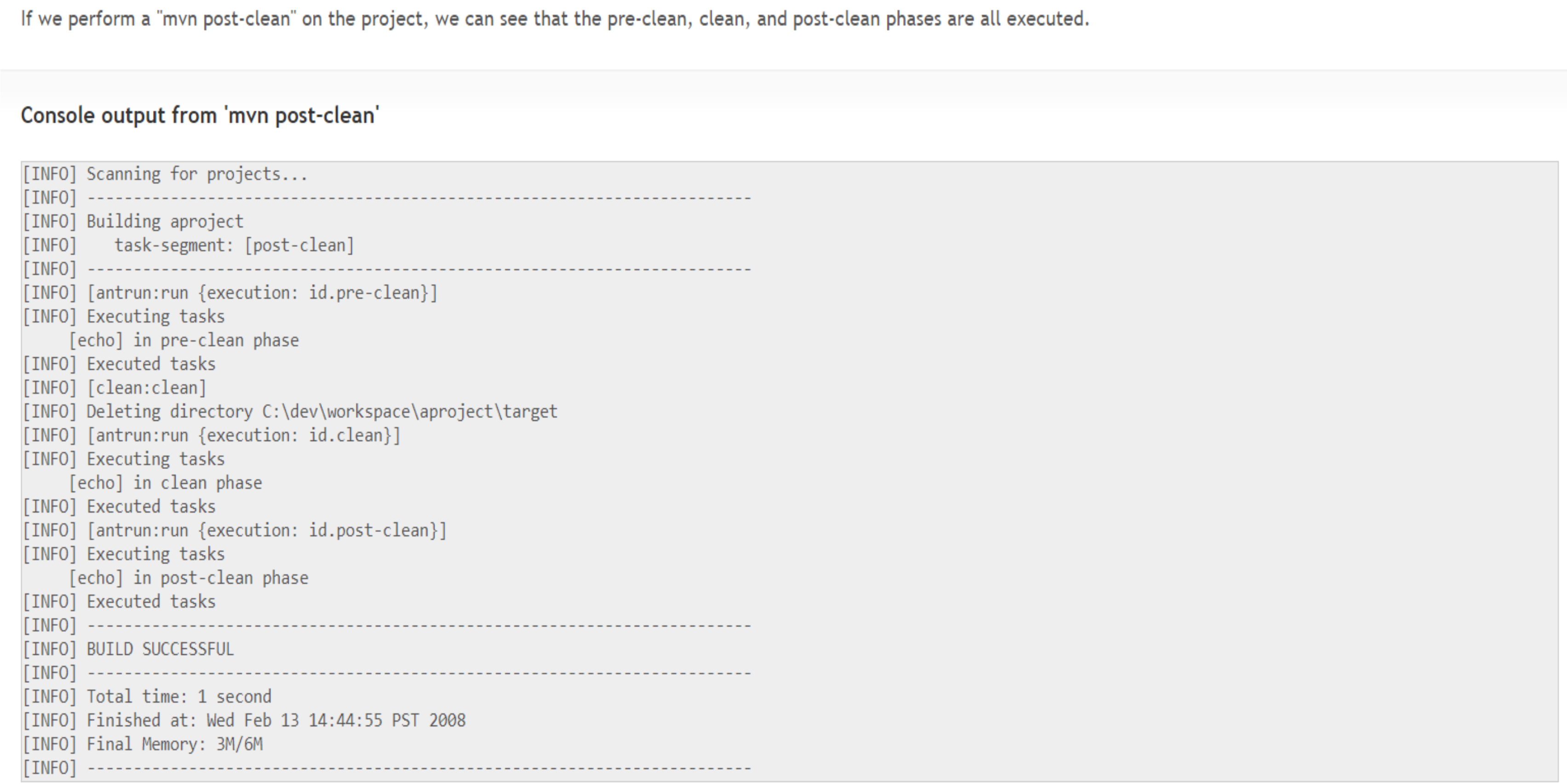


If we perform a "mvn clean" on the project with the above pom.xml, we can see that the pre-clean and clean phases are executed based on our text messages.

Console output from 'mvn clean'

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building aproject
[INFO]    task-segment: [clean]
[INFO] ------------------------------------------------------------------------
[INFO] [antrun:run {execution: id.pre-clean}]
[INFO] Executing tasks
     [echo] in pre-clean phase
[INFO] Executed tasks
[INFO] [clean:clean]
[INFO] Deleting directory C:\dev\workspace\aproject\target
[INFO] [antrun:run {execution: id.clean}]
[INFO] Executing tasks
     [echo] in clean phase
[INFO] Executed tasks
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1 second
[INFO] Finished at: Wed Feb 13 14:44:26 PST 2008
[INFO] Final Memory: 3M/7M
[INFO] ------------------------------------------------------------------------
```

Maven: Build Life Cycle of clean phase



If we perform a "mvn post-clean" on the project, we can see that the pre-clean, clean, and post-clean phases are all executed.

Console output from 'mvn post-clean'

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building aproject
[INFO]    task-segment: [post-clean]
[INFO] ------------------------------------------------------------------------
[INFO] [antrun:run {execution: id.pre-clean}]
[INFO] Executing tasks
     [echo] in pre-clean phase
[INFO] Executed tasks
[INFO] [clean:clean]
[INFO] Deleting directory C:\dev\workspace\aproject\target
[INFO] [antrun:run {execution: id.clean}]
[INFO] Executing tasks
     [echo] in clean phase
[INFO] Executed tasks
[INFO] [antrun:run {execution: id.post-clean}]
[INFO] Executing tasks
     [echo] in post-clean phase
[INFO] Executed tasks
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1 second
[INFO] Finished at: Wed Feb 13 14:44:55 PST 2008
[INFO] Final Memory: 3M/6M
[INFO] ------------------------------------------------------------------------
```

Site Lifecycle

- Maven Site plugin is generally used to create fresh documentation to create reports, deploy site, etc. It has the following phases −
- pre-site
- site
- post-site
- site-deploy

Maven Default Lifecycle

- validate: This phase validates the project POM file and makes sure all the necessary information related to carry out the build is available.
- initialize: This phase initializes the build by setting up the right directory structure and initializing properties.
- generate-sources: This phase generates any required source code.
- process-sources: This phase processes the generated source code. For example, there can be a plugin running in this phase to filter the source code based on some defined criteria.
- generate-resources: This phase generates any resources that need to be packaged with the final artifact.

Maven Default Lifecycle

- process-resources: This phase processes the generated resources. It copies the resources to their destination directories and makes them ready for packaging.
- compile: This phase compiles the source code.
- process-classes: This phase can be used to carry out any bytecode enhancements after the compile phase.
- generate-test-sources: This phase generates the required source code for tests.
- process-test-sources: This phase processes the generated test source code. For example, there can be a plugin running in this phase to filter the source code based on some defined criteria.

Maven Default Lifecycle

- generate-test-resources: This phase generates all the resources required to run tests.
- process-test-resources: This phase processes the generated test resources. It copies the resources to their destination directories and makes them ready for testing.
- test-compile: This phase compiles the source code for tests.
- process-test-classes: This phase can be used to carry out any bytecode enhancements after the test-compile phase.

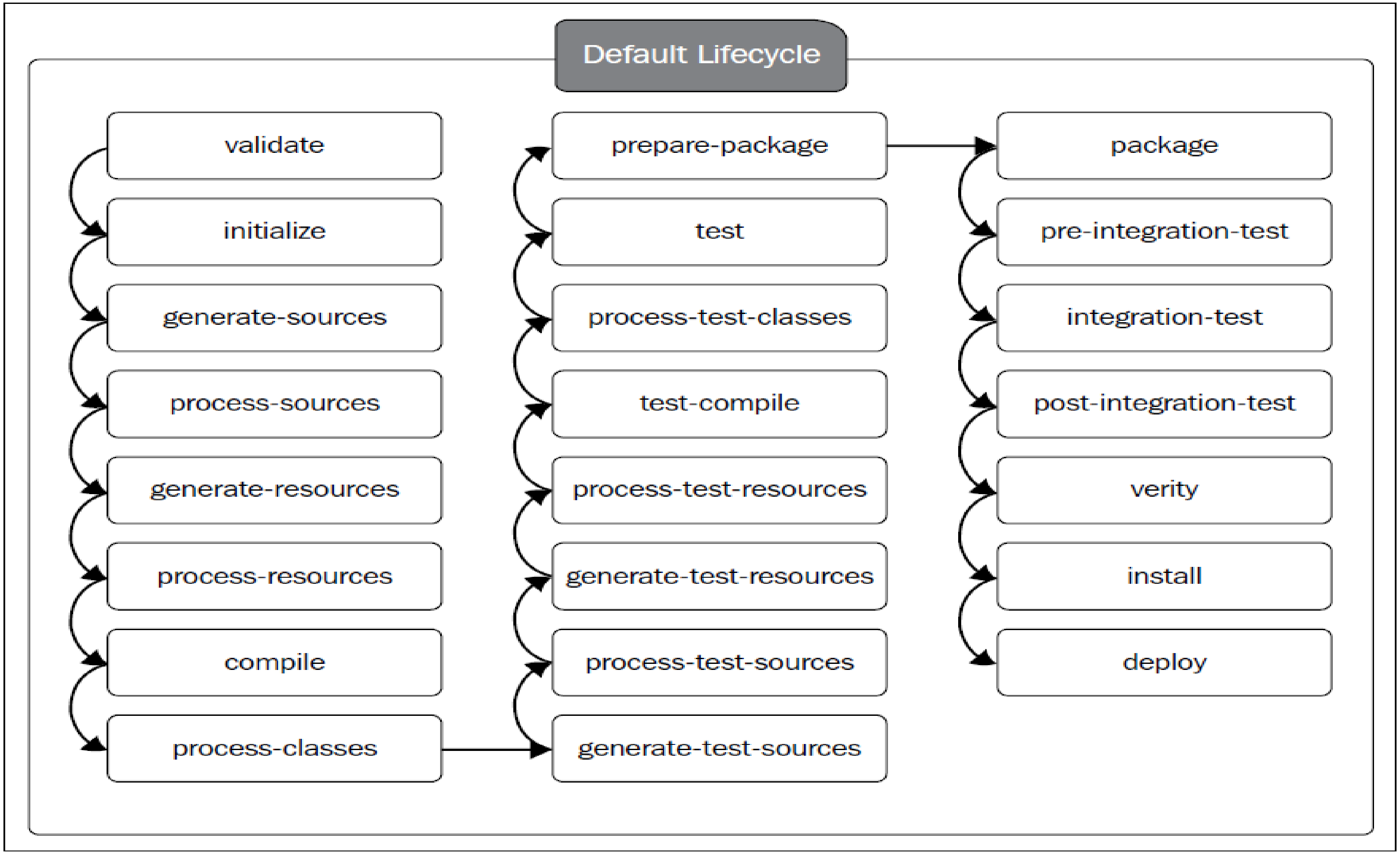
Maven Default Lifecycle

- test: This phase executes tests using the appropriate unit test framework.
- prepare-package: This phase is useful in organizing the artifacts to be packaged.
- package: This phase packs the artifacts into a distributable format, for example, JAR or WAR.
- pre-integration-test: This phase performs the actions required (if any) before running integration tests. This may be used to start any external application servers and deploy the artifacts into different test environments.
- integration-test: This phase runs integration tests.

Maven Default Lifecycle

- post-integration-test: This phase can be used to perform any cleanup tasks after running the integration tests.
- verify: This phase verifies the validity of the package. The criteria to check the validity needs to be defined by the respective plugins.
- install: This phase installs the final artifact in the local repository.
- deploy: This phase deploys the final artifact to a remote repository.

Maven Default Lifecycle



Default Lifecycle

validate

initialize

generate-sources

process-sources

generate-resources

process-resources

compile

process-classes

generate-test-sources

process-test-sources

generate-test-resources

process-test-resources

test-compile

process-test-classes

test

prepare-package

package

pre-integration-test

integration-test

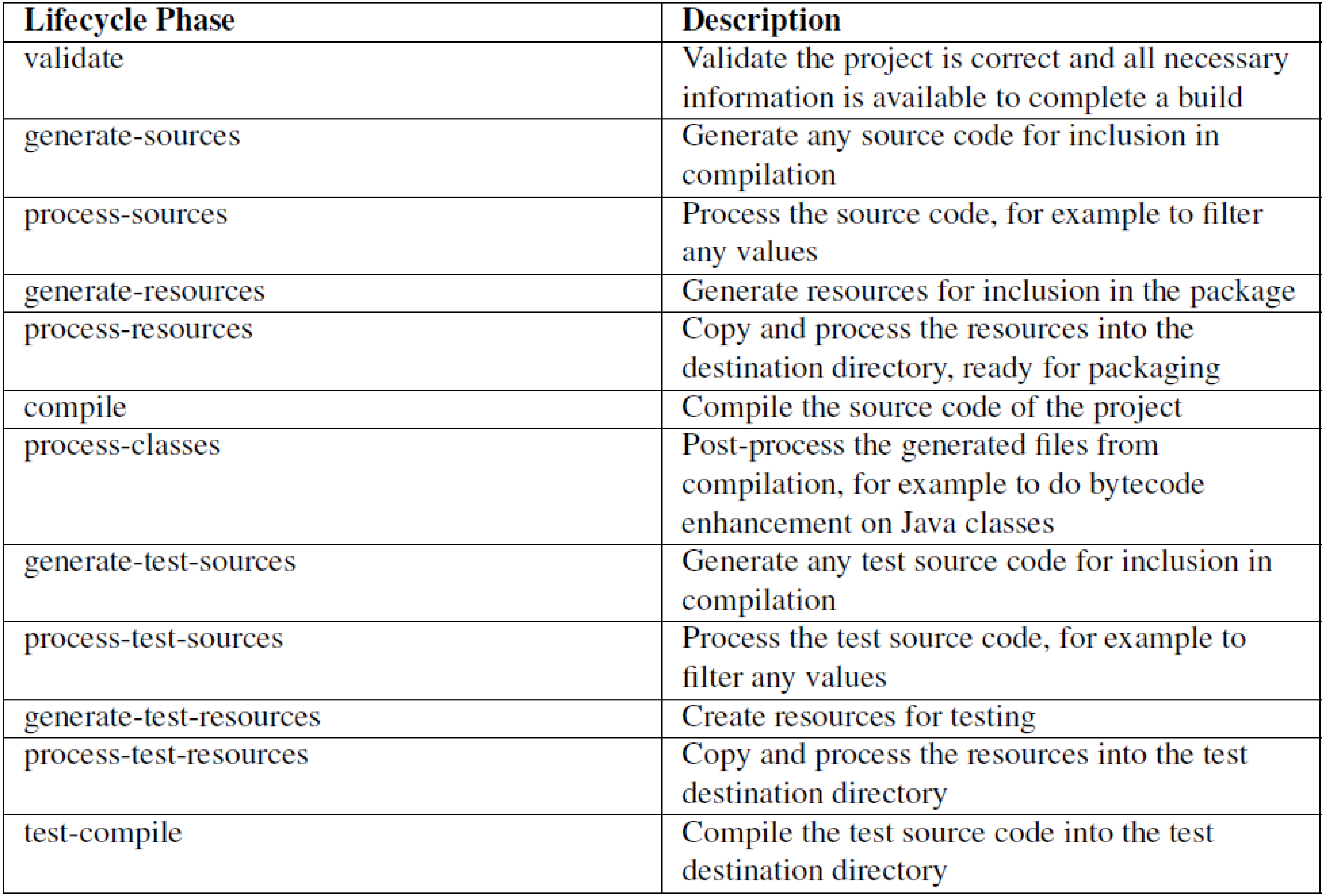
post-integration-test

verity

install

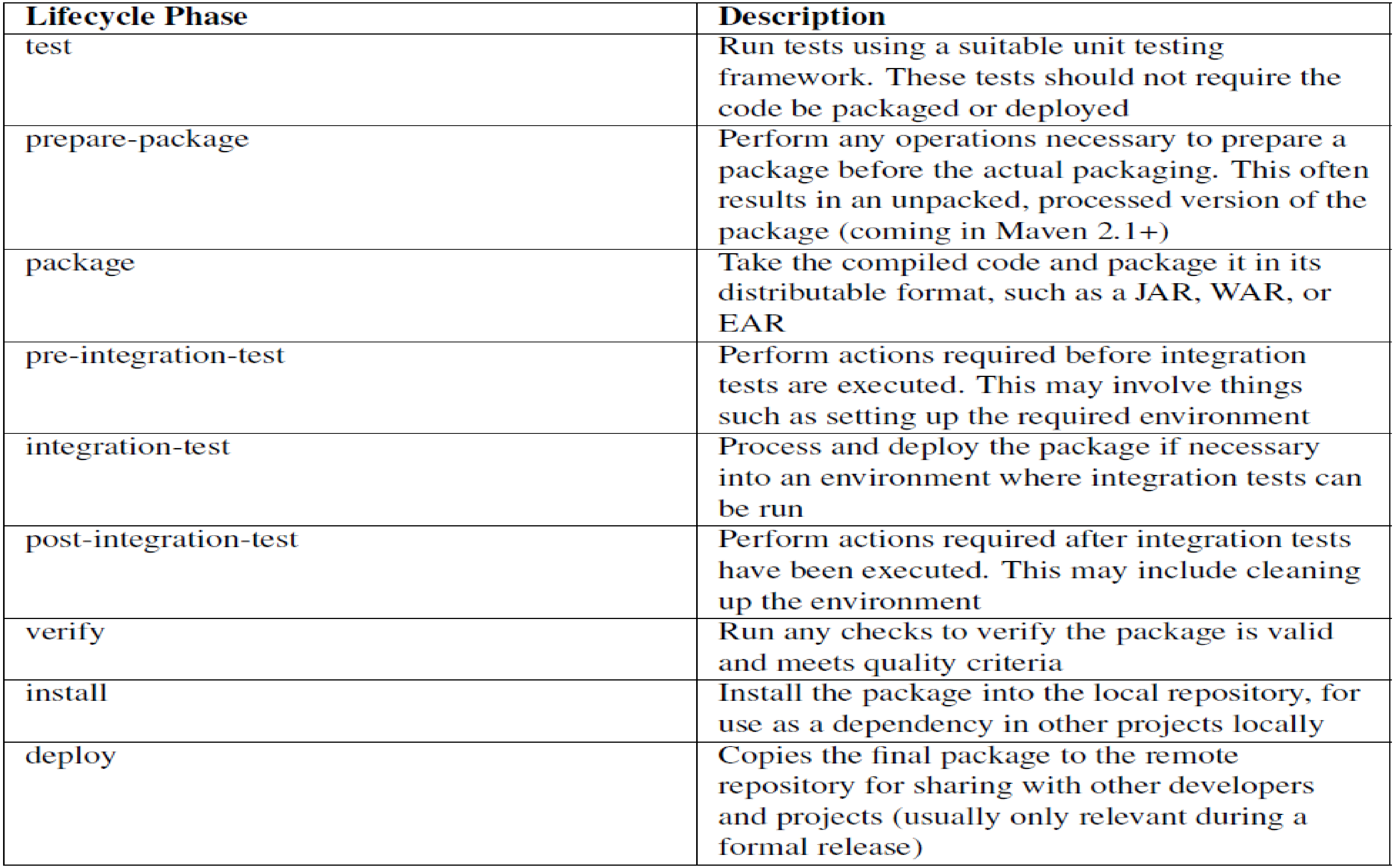
deploy

Maven Life Cycle (Default)



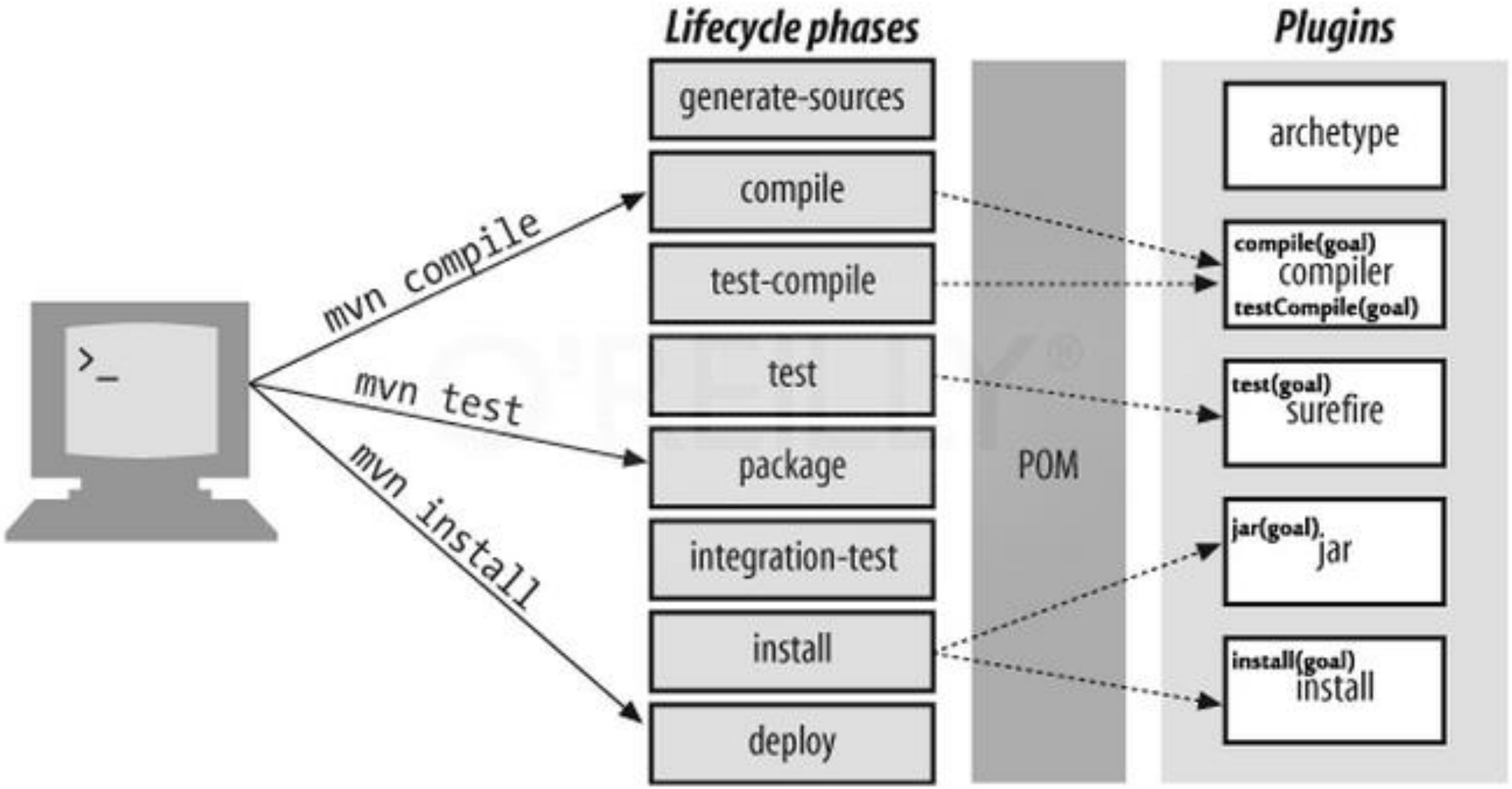
Lifecycle Phase	Description
validate	Validate the project is correct and all necessary information is available to complete a build
generate-sources	Generate any source code for inclusion in compilation
process-sources	Process the source code, for example to filter any values
generate-resources	Generate resources for inclusion in the package
process-resources	Copy and process the resources into the destination directory, ready for packaging
compile	Compile the source code of the project
process-classes	Post-process the generated files from compilation, for example to do bytecode enhancement on Java classes
generate-test-sources	Generate any test source code for inclusion in compilation
process-test-sources	Process the test source code, for example to filter any values
generate-test-resources	Create resources for testing
process-test-resources	Copy and process the resources into the test destination directory
test-compile	Compile the test source code into the test destination directory

Maven Life Cycle (Default)



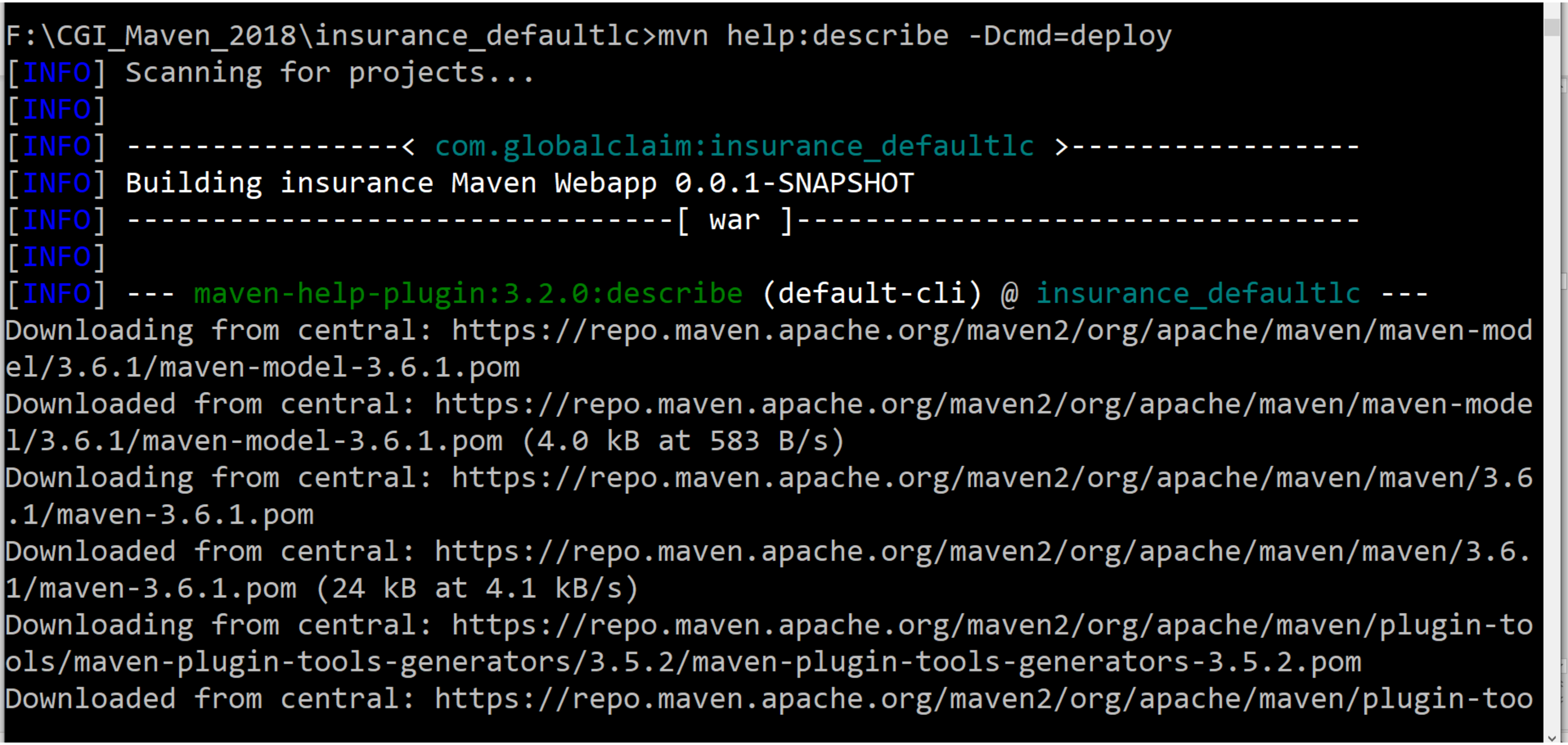
Lifecycle Phase	Description
test	Run tests using a suitable unit testing framework. These tests should not require the code be packaged or deployed
prepare-package	Perform any operations necessary to prepare a package before the actual packaging. This often results in an unpacked, processed version of the package (coming in Maven 2.1+)
package	Take the compiled code and package it in its distributable format, such as a JAR, WAR, or EAR
pre-integration-test	Perform actions required before integration tests are executed. This may involve things such as setting up the required environment
integration-test	Process and deploy the package if necessary into an environment where integration tests can be run
post-integration-test	Perform actions required after integration tests have been executed. This may include cleaning up the environment
verify	Run any checks to verify the package is valid and meets quality criteria
install	Install the package into the local repository, for use as a dependency in other projects locally
deploy	Copies the final package to the remote repository for sharing with other developers and projects (usually only relevant during a formal release)

Maven Life Cycle (Default)



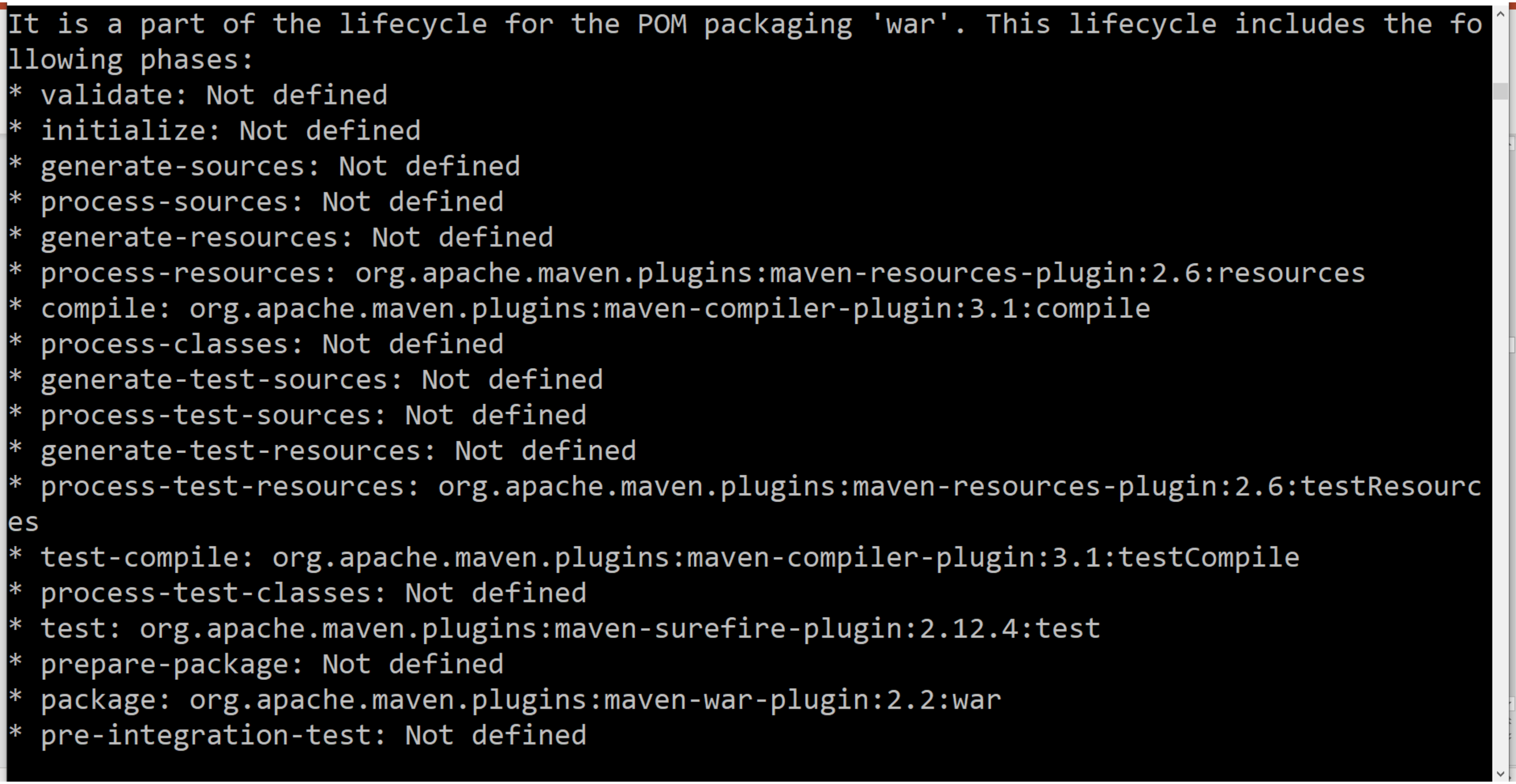
Maven Life Cycle (Default)

Mvn help:describe –Dcmd=clean



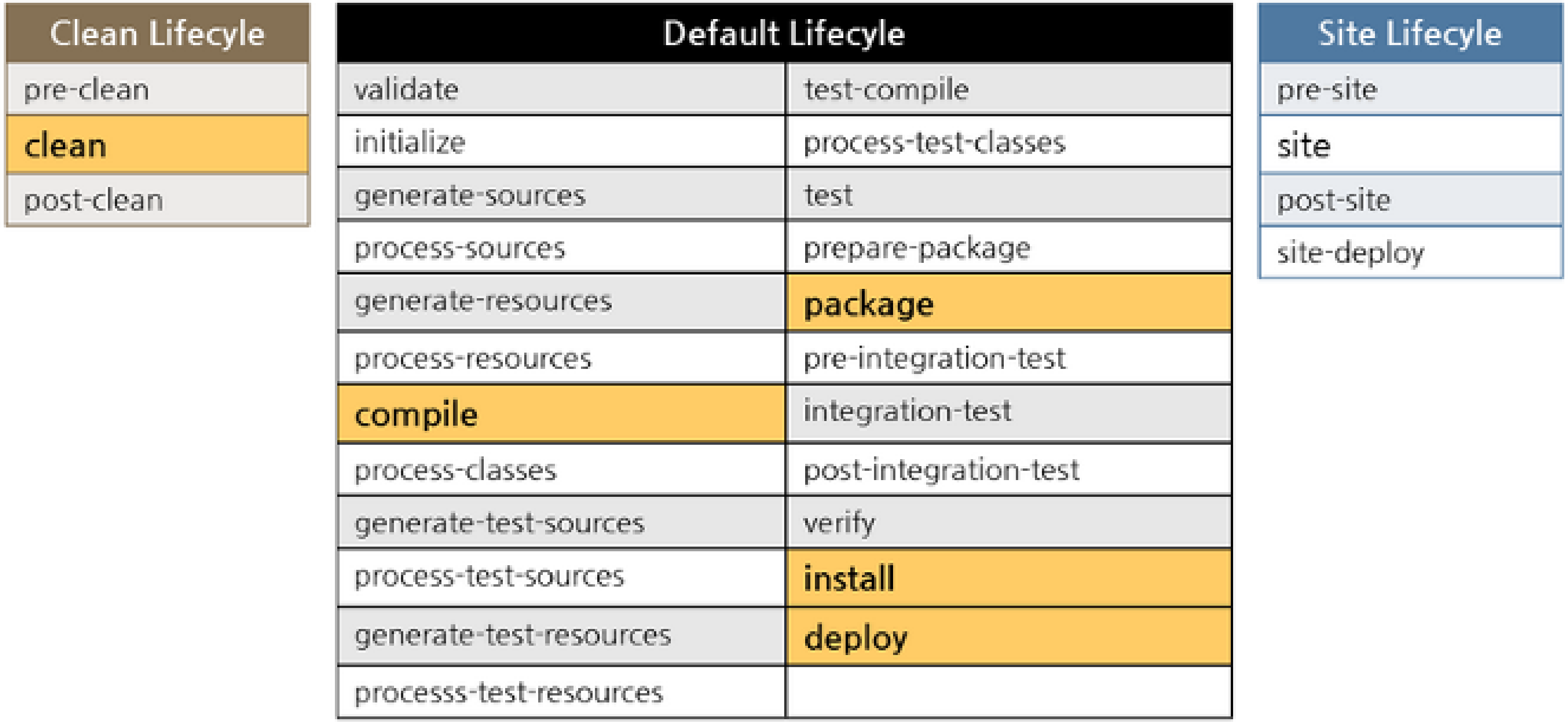
```
F:\CGI_Maven_2018\insurance_defaultlc>mvn help:describe -Dcmd=deploy
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------< com.globalclaim:insurance_defaultlc >-----------------
[INFO] Building insurance Maven Webapp 0.0.1-SNAPSHOT
[INFO] --------------------------------[ war ]---------------------------------
[INFO]
[INFO] --- maven-help-plugin:3.2.0:describe (default-cli) @ insurance_defaultlc ---
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven-mod
el/3.6.1/maven-model-3.6.1.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven-mode
l/3.6.1/maven-model-3.6.1.pom (4.0 kB at 583 B/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven/3.6
.1/maven-3.6.1.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven/3.6.
1/maven-3.6.1.pom (24 kB at 4.1 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugin-to
ols/maven-plugin-tools-generators/3.5.2/maven-plugin-tools-generators-3.5.2.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugin-too
```

Maven Life Cycle (Default)



```
It is a part of the lifecycle for the POM packaging 'war'. This lifecycle includes the fo
llowing phases:
* validate: Not defined
* initialize: Not defined
* generate-sources: Not defined
* process-sources: Not defined
* generate-resources: Not defined
* process-resources: org.apache.maven.plugins:maven-resources-plugin:2.6:resources
* compile: org.apache.maven.plugins:maven-compiler-plugin:3.1:compile
* process-classes: Not defined
* generate-test-sources: Not defined
* process-test-sources: Not defined
* generate-test-resources: Not defined
* process-test-resources: org.apache.maven.plugins:maven-resources-plugin:2.6:testResourc
es
* test-compile: org.apache.maven.plugins:maven-compiler-plugin:3.1:testCompile
* process-test-classes: Not defined
* test: org.apache.maven.plugins:maven-surefire-plugin:2.12.4:test
* prepare-package: Not defined
* package: org.apache.maven.plugins:maven-war-plugin:2.2:war
* pre-integration-test: Not defined
```

Maven Life Cycle (Default)



Clean Lifecyle
pre-clean
clean
post-clean

Default Lifecyle	
validate	test-compile
initialize	process-test-classes
generate-sources	test
process-sources	prepare-package
generate-resources	**package**
process-resources	pre-integration-test
compile	integration-test
process-classes	post-integration-test
generate-test-sources	verify
process-test-sources	**install**
generate-test-resources	**deploy**
processs-test-resources	

Site Lifecyle
pre-site
site
post-site
site-deploy

Maven Life Cycle (Default)

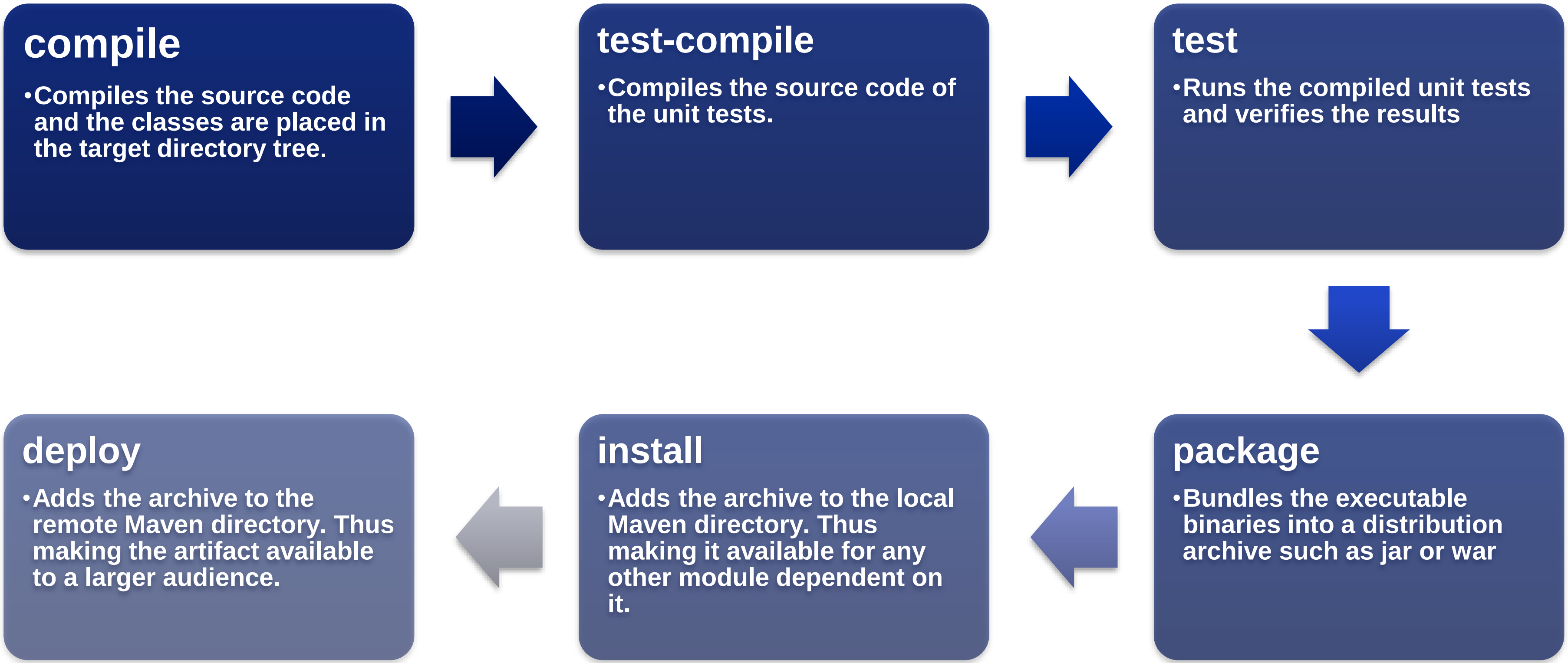
- if we call a particular phase via a maven command, such as "mvn compile", all phases up to and including that phase will be executed.
- So, in the case of "mvn compile", we would actually go through the validate, generate-sources, process-sources, generate-resources, process-resources, and compile phases.
- The second main concept to be aware of in regards to lifecycles is that, based on the packaging of a project (jar, warW, earW, etc), different maven goals will be bound to different phases of the maven lifecycle.

Maven Life cycle phases

- Plug-ins are software modules which are written to fit into the plug-in framework of Maven.
- Each task within a plugin is called a *mojo*.
- A mojo is executed when the Maven engine executes the corresponding phase on the build life cycle.
- The association between the phase of a life cycle and a mojo is called as a *binding*.

Maven Life cycle phases (Continued…)

Few phases of the build life cycle are described below:



compile

- Compiles the source code and the classes are placed in the target directory tree.

test-compile

- Compiles the source code of the unit tests.

test

- Runs the compiled unit tests and verifies the results

package

- Bundles the executable binaries into a distribution archive such as jar or war

install

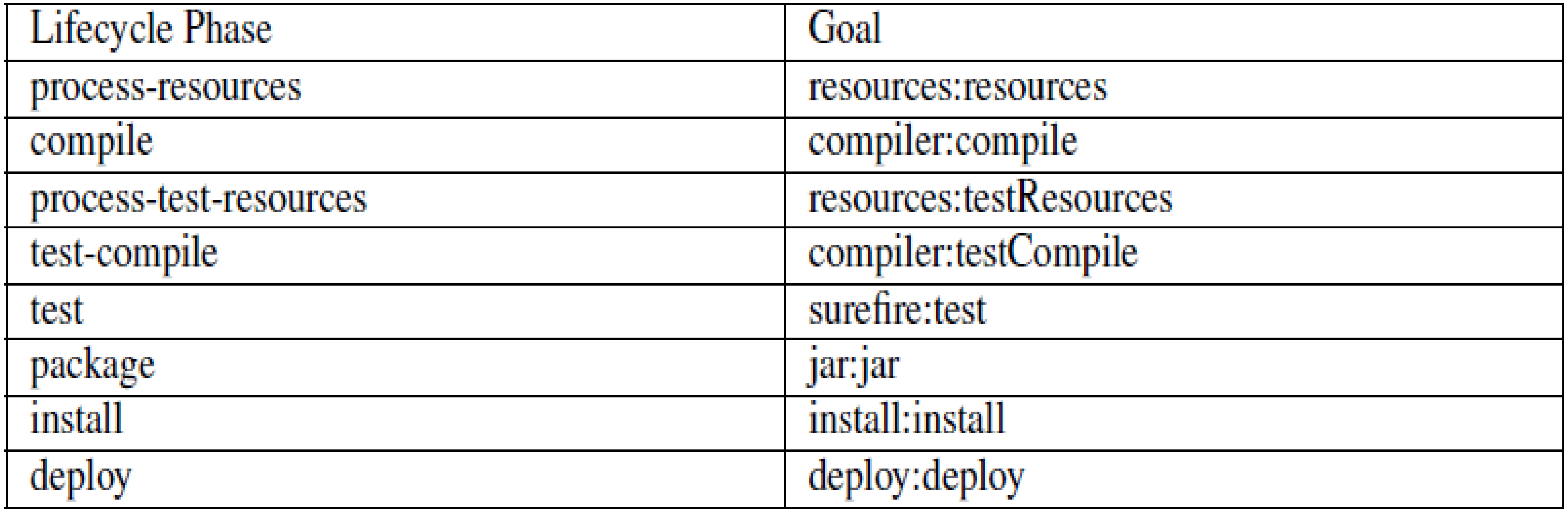
- Adds the archive to the local Maven directory. Thus making it available for any other module dependent on it.

deploy

- Adds the archive to the remote Maven directory. Thus making the artifact available to a larger audience.

Package-specific Lifecycles

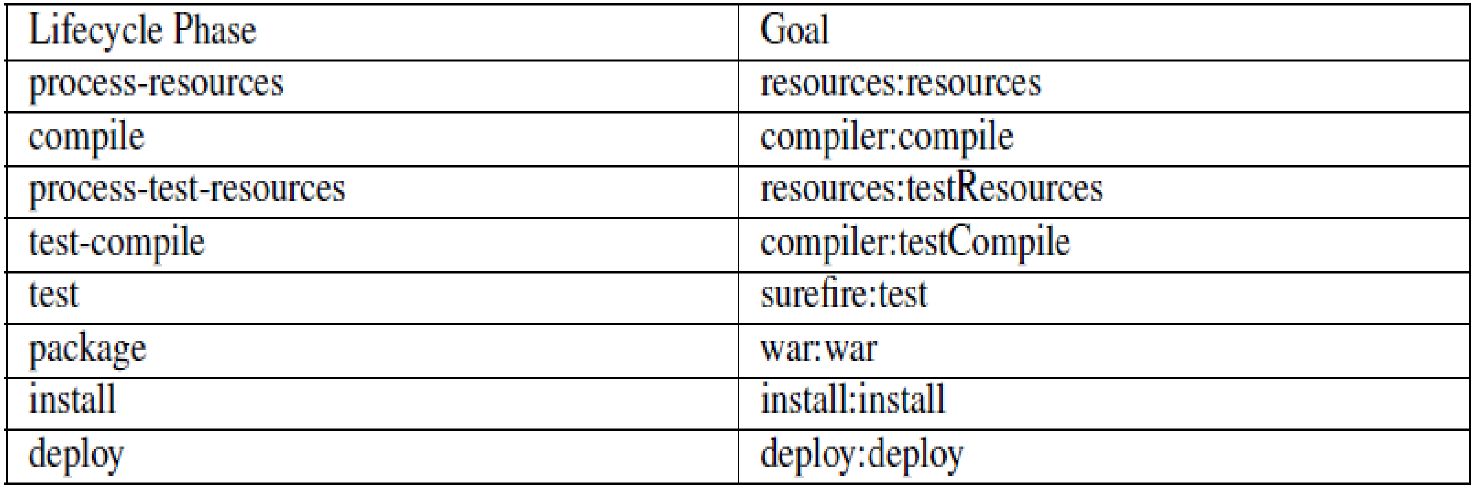
- JAR



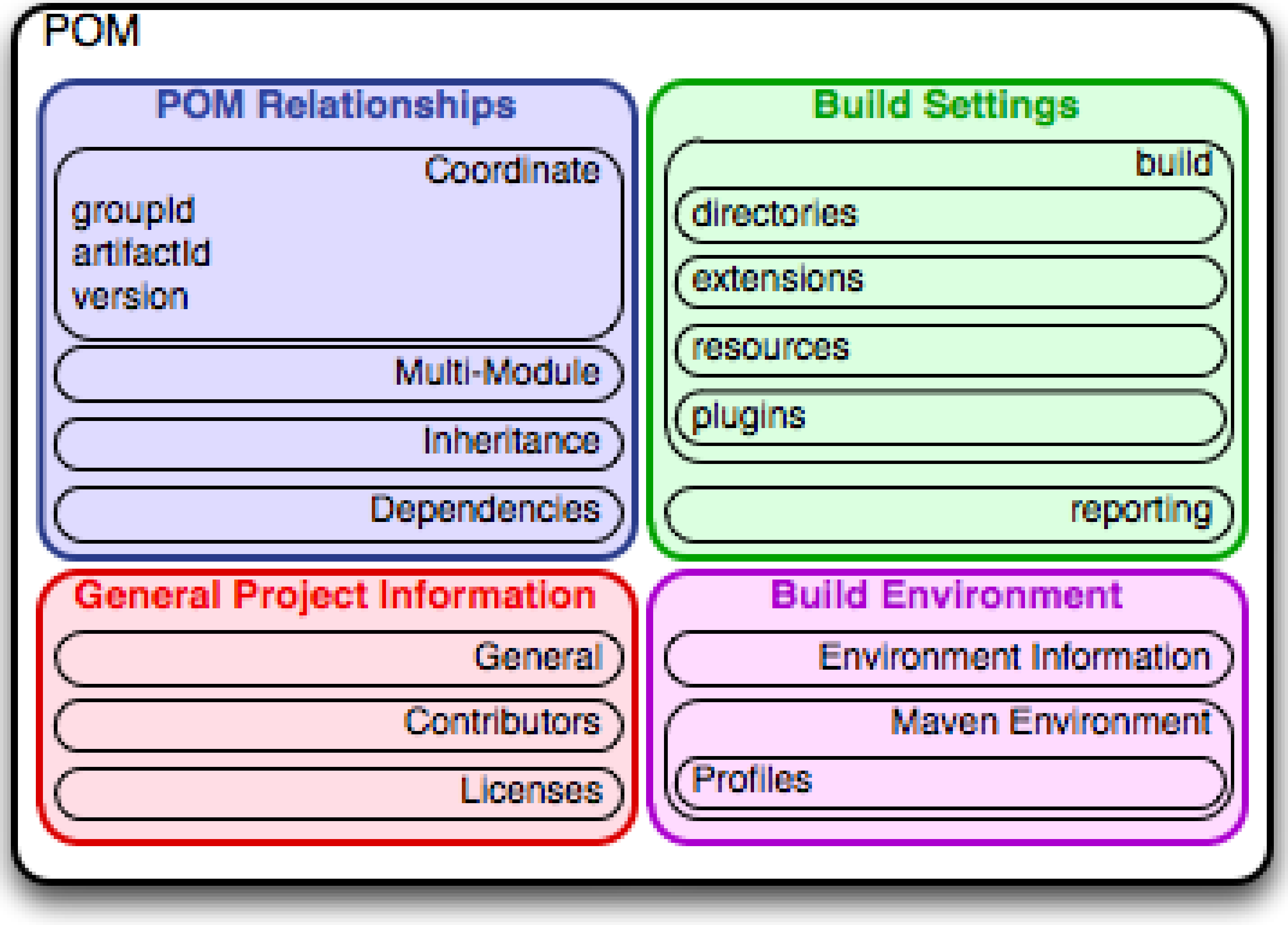
Lifecycle Phase	Goal
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar
install	install:install
deploy	deploy:deploy

Package-specific Lifecycles

- WAR



Lifecycle Phase	Goal
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	war:war
install	install:install
deploy	deploy:deploy



POM

POM Relationships

Coordinate

groupId
artifactId
version

Multi-Module

Inheritance

Dependencies

Build Settings

build

directories

extensions

resources

plugins

reporting

General Project Information

General

Contributors

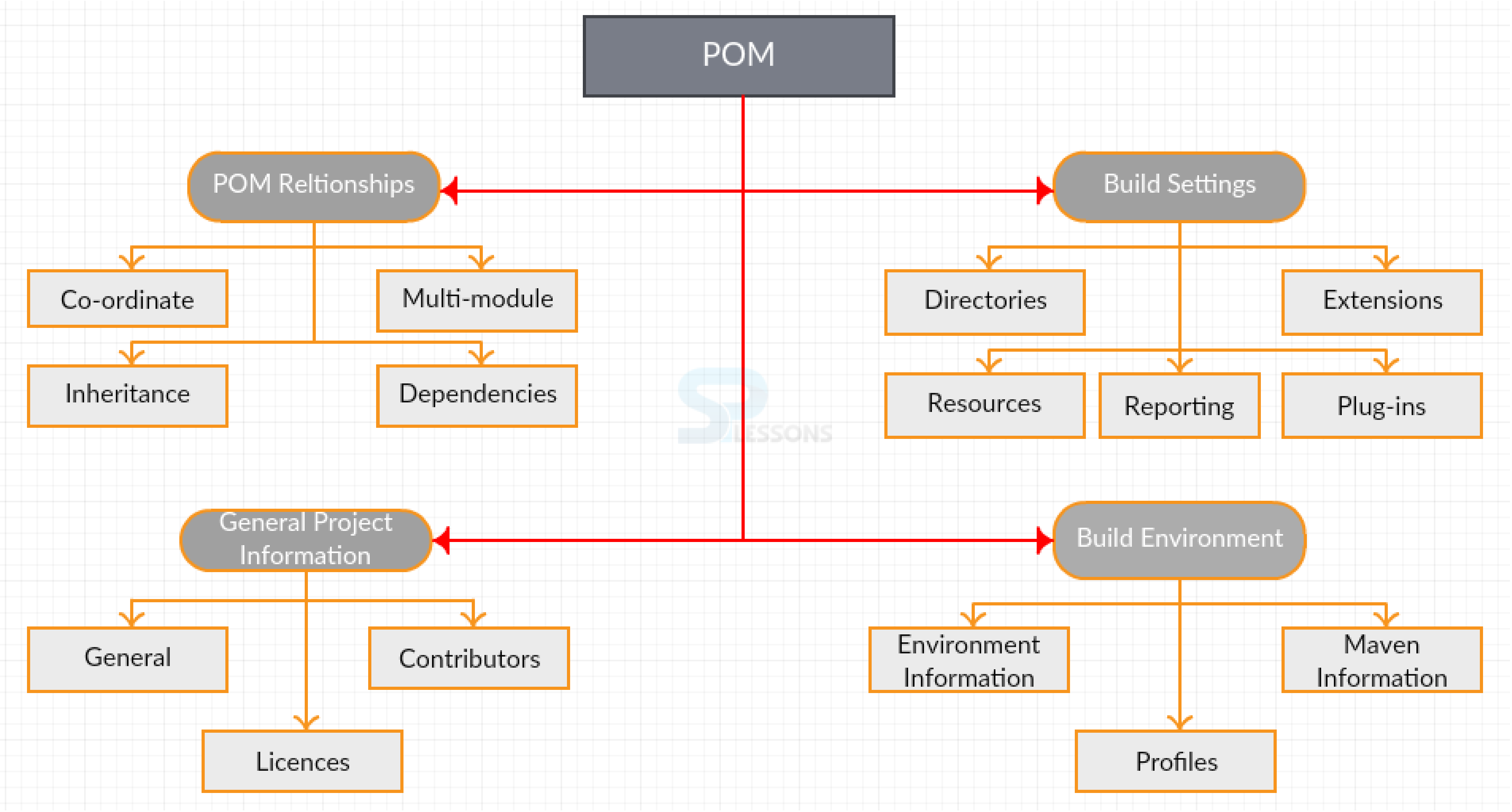
Licenses

Build Environment

Environment Information

Maven Environment

Profiles


POM
POM Reltionships
Build Settings
Co-ordinate
Multi-module
Inheritance
Dependencies
Directories
Extensions
Resources
Reporting
Plug-ins
General Project Information
Build Environment
General
Contributors
Licences
Environment Information
Maven Information
Profiles

POM Major Elements

- project –> root element
- modelVersion –> mandatory element with value 4.0.0 which is currently supported version for both Maven 2 & 3
- groupId –> this is the top-most element while considering maven repository structure and its generally kept unique amongst organization
- artifactId –> this is similar to name of the project and this lies under <groupId> element
- version –> this is final element in Maven co-ordinate system and generally it defines version of the project. Like, initially it can be kept 1.0 and later with couple changes version changes to 1.1 or 2.0 depending on the major/minor releases. Together with <groupId> & <artifactId> elements acts like an address and timestamp in one

POM Major Elements

- **name** –> any meaningful name to the project
- **description** –> description of the project
- **packaging** –> this elements defines how exactly this project needs to be bundled or packaged after development. Various packaging standards in Java/JEE project developemtns are JAR or WAR or EAR. When no <*packaging*> element defined, then by default final products is packaged as JAR
- **url** –> this elements can be used to define where exactly project going to live after development (normally it points to SVN location)
- **dependencies/dependency** –> defines what are the required dependencies for this project into developments (**dependency scopes** and **transitive dependencies**)

POM Major Elements

- **build/plugins** –> various plugins along with their goals can be configured under <build> element (Maven plugins)
- **finalName** –> this is the child element under <build> element defining what will be final name of the project and a JAR or WAR or EAR will be generated under "***target***" folder with this name

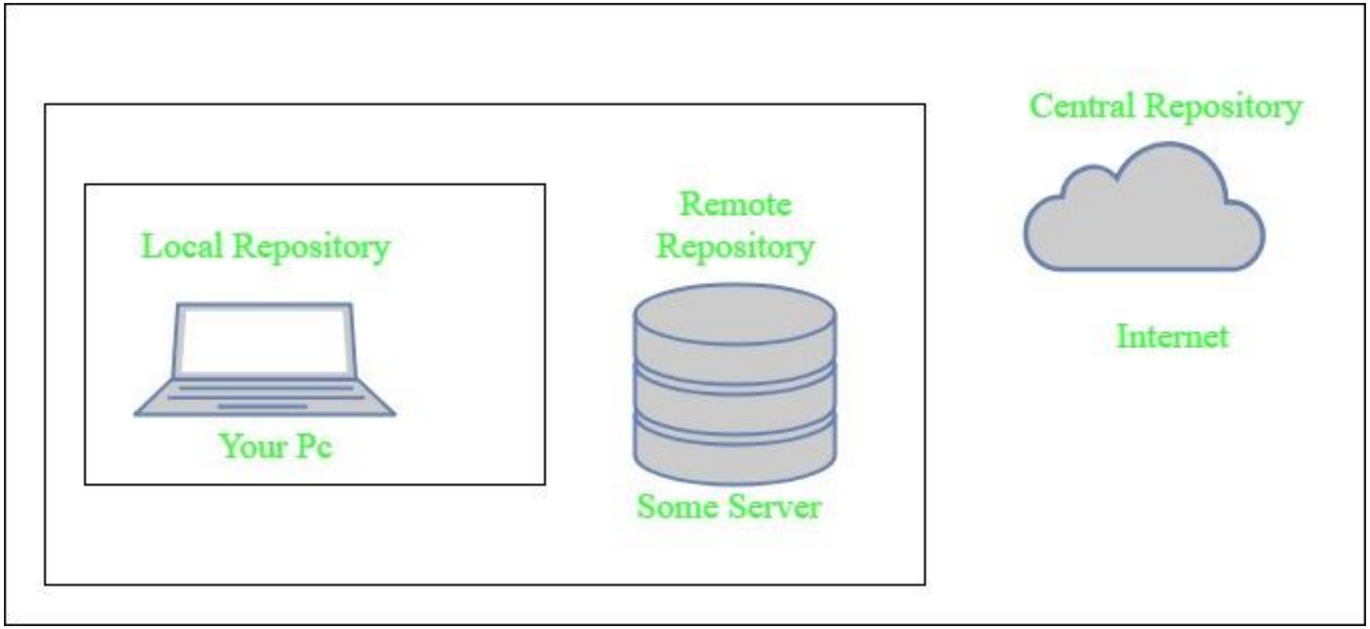
Maven Repository

- Maven repositories are directories of packaged JAR files with some metadata.
- The metadata are POM files related to the projects each packaged JAR file belongs to, including what external dependencies each packaged JAR has.
- This metadata enables Maven to download dependencies of your dependencies recursively until all dependencies are download and put into your local machine.
- Maven has three types of repository :
  - Local repository
  - Central repository
  - Remote repository

Maven Repository

Maven searches for dependencies in this repositories.

First maven searches in Local repository then Central repository then Remote repository if Remote repository specified in the POM.



Snapshot in Maven

- SNAPSHOT can be defined as a special version that indicates a current development copy.
- Unlike the regular versions, Maven checks for a new SNAPSHOT version in its remote repository. Maven does it for every build.

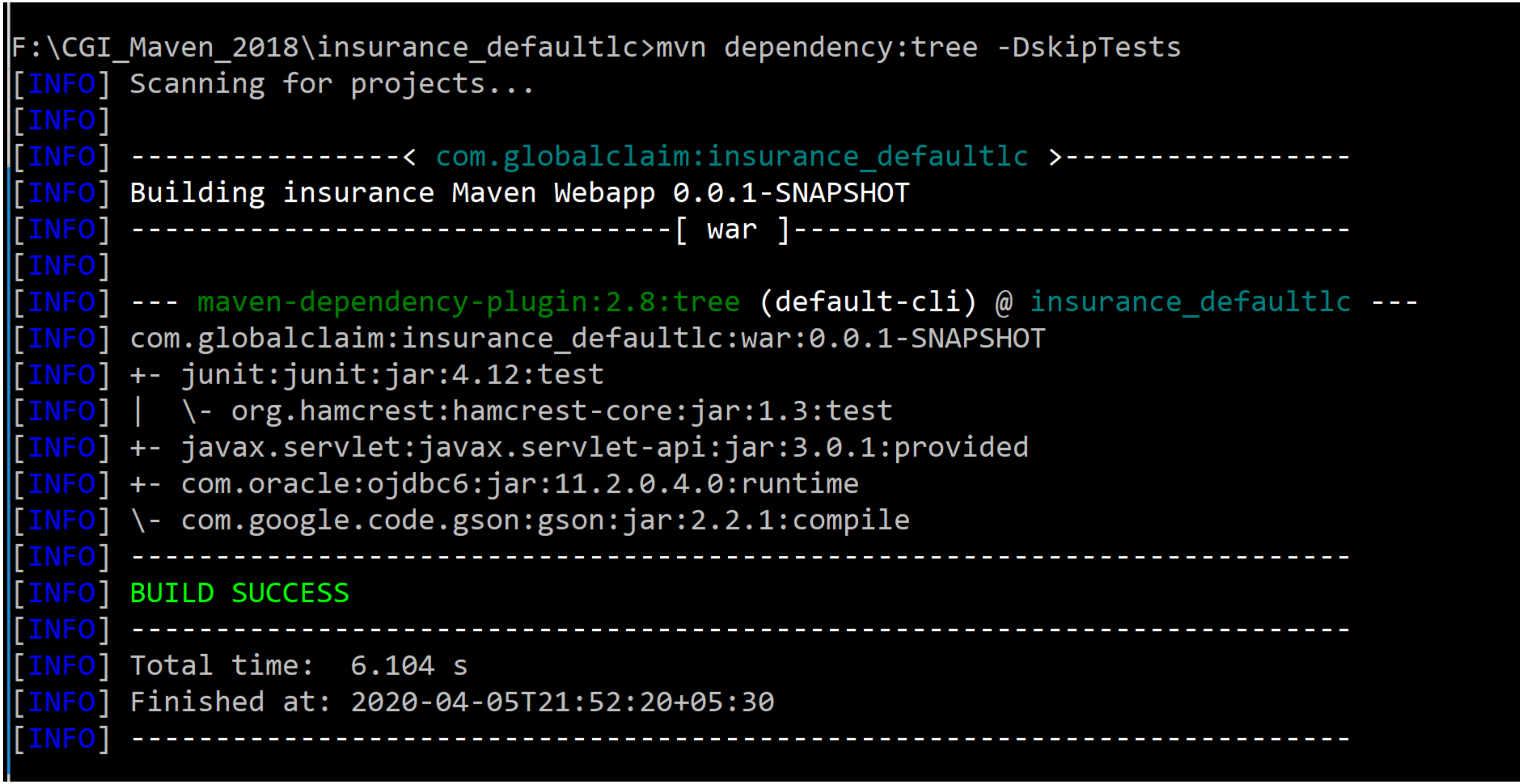
Snapshot in Maven

- In the case of Version, if Maven once downloads the mentioned version say data-service:1.0, it will never try to download a newer 1.0 available in the repository.
- To download the updated code, the data-service version is then upgraded to 1.1.
- In the case of SNAPSHOT, Maven will automatically fetch the latest SNAPSHOT (data-service:1.0-SNAPSHOT) every time the team builds its project.

Dependency Management in Maven

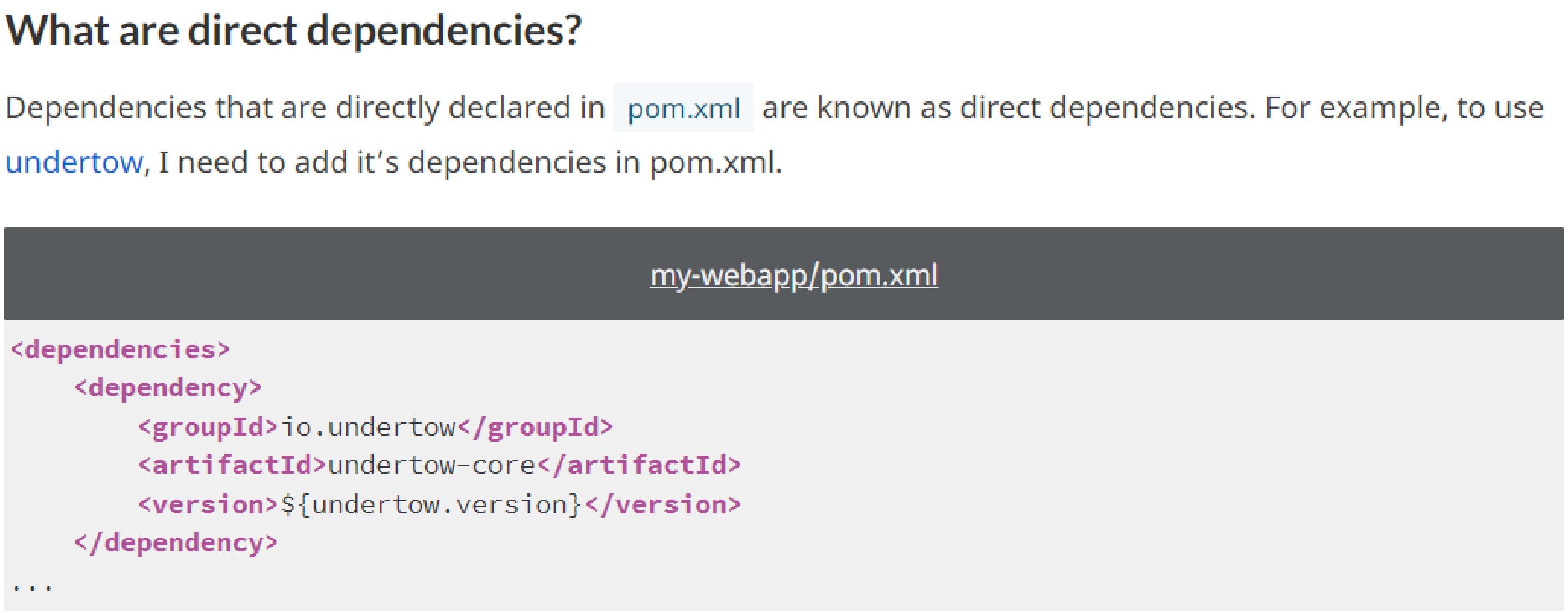
- Dependency management is a core feature of maven.
- It is powerful enough to manage dependencies from a simple single project to a complex multi-module project.
- There are basically 2 types of dependencies that maven manages, direct and transitive dependencies.

Maven dependency tree



```
F:\CGI_Maven_2018\insurance_defaultlc>mvn dependency:tree -DskipTests
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------< com.globalclaim:insurance_defaultlc >-----------------
[INFO] Building insurance Maven Webapp 0.0.1-SNAPSHOT
[INFO] --------------------------------[ war ]---------------------------------
[INFO]
[INFO] --- maven-dependency-plugin:2.8:tree (default-cli) @ insurance_defaultlc ---
[INFO] com.globalclaim:insurance_defaultlc:war:0.0.1-SNAPSHOT
[INFO] +- junit:junit:jar:4.12:test
[INFO] |  \- org.hamcrest:hamcrest-core:jar:1.3:test
[INFO] +- javax.servlet:javax.servlet-api:jar:3.0.1:provided
[INFO] +- com.oracle:ojdbc6:jar:11.2.0.4.0:runtime
[INFO] \- com.google.code.gson:gson:jar:2.2.1:compile
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  6.104 s
[INFO] Finished at: 2020-04-05T21:52:20+05:30
[INFO] ------------------------------------------------------------------------
```

Direct Dependencies



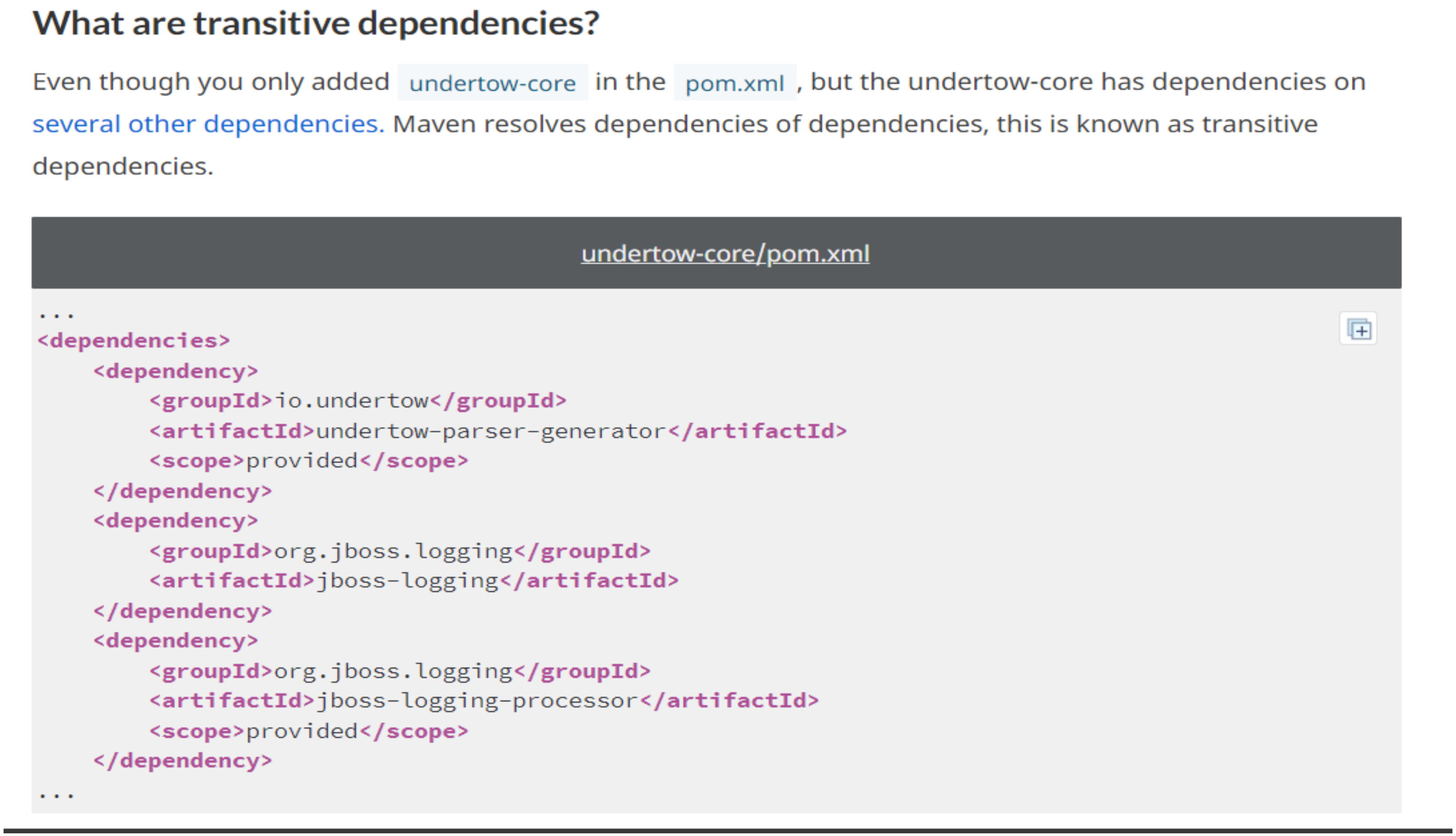
What are direct dependencies?

Dependencies that are directly declared in `pom.xml` are known as direct dependencies. For example, to use undertow, I need to add it's dependencies in pom.xml.

my-webapp/pom.xml

```
<dependencies>
    <dependency>
        <groupId>io.undertow</groupId>
        <artifactId>undertow-core</artifactId>
        <version>${undertow.version}</version>
    </dependency>
...
```

Transitive Dependencies



What are transitive dependencies?

Even though you only added `undertow-core` in the `pom.xml` , but the undertow-core has dependencies on several other dependencies. Maven resolves dependencies of dependencies, this is known as transitive dependencies.

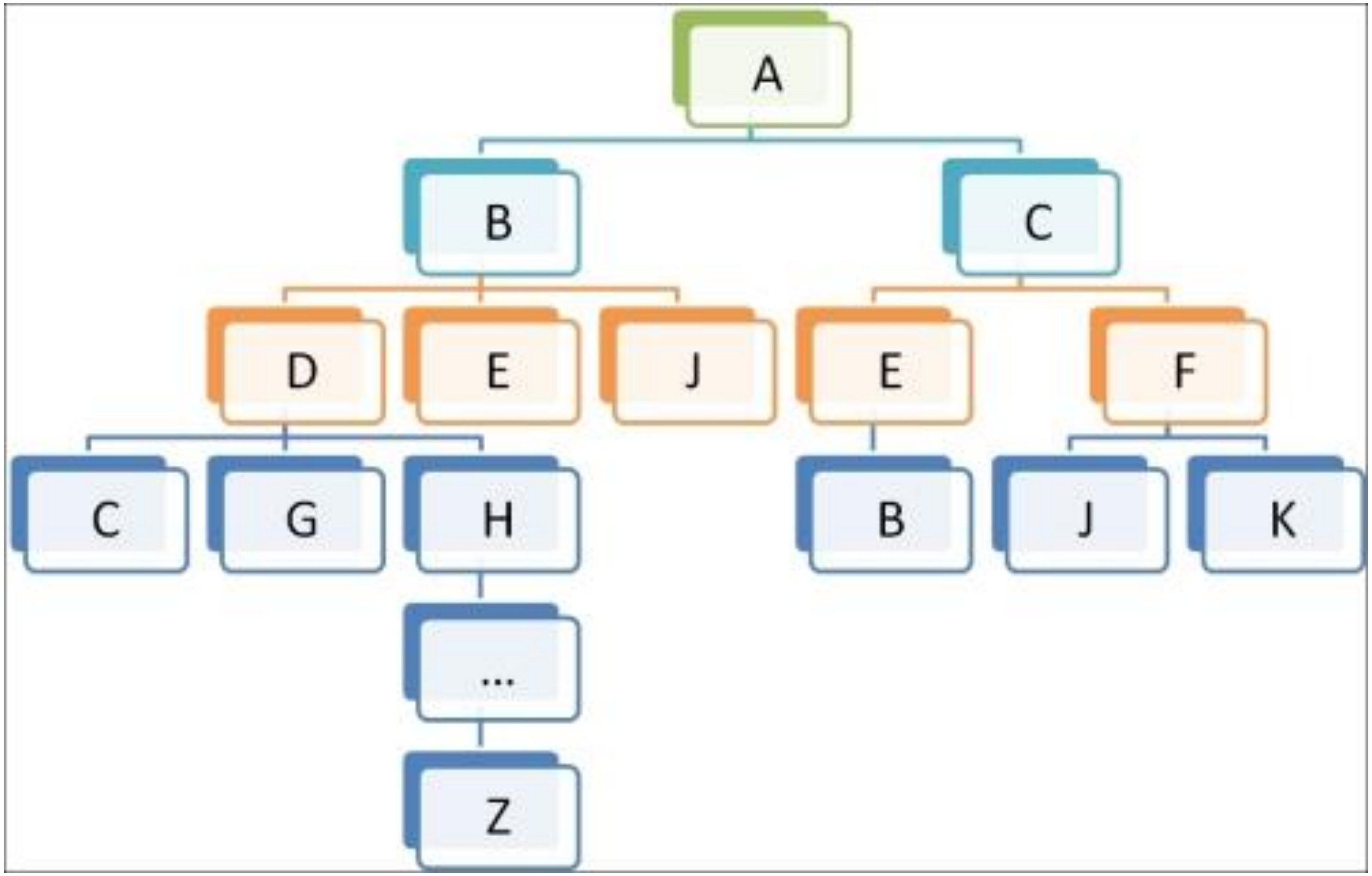
undertow-core/pom.xml

```
...
<dependencies>
    <dependency>
        <groupId>io.undertow</groupId>
        <artifactId>undertow-parser-generator</artifactId>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.jboss.logging</groupId>
        <artifactId>jboss-logging</artifactId>
    </dependency>
    <dependency>
        <groupId>org.jboss.logging</groupId>
        <artifactId>jboss-logging-processor</artifactId>
        <scope>provided</scope>
    </dependency>
...
```

Transitive Dependency

- Dependencies are transitive.
- This means that if A depends on B and B depends on C, then A depends on both B and C.
- Theoretically, there is no limit to the depth of dependency.
- So, if you observe the following diagram of the tree of dependencies, you will notice that by transitivity, A depends on B, C, D, … until Z:

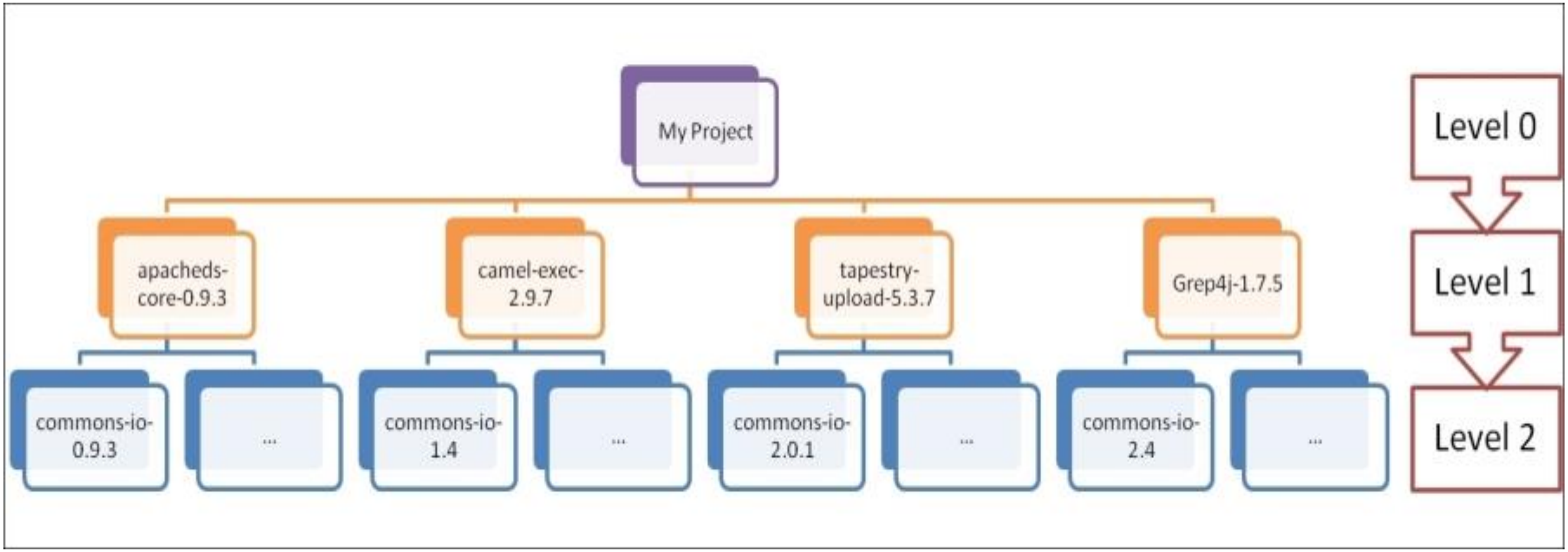
Transitive Dependency



Transitive Dependency

- Resolution
  - Maven carries out the following algorithm to choose between two different versions:
  - Nearest first: A dependency of lower level has priority over another of the higher depth. Hence, a direct dependency has priority over a transitive dependency.
  - First found: At the same level, the first dependency that is found is taken.
  - This algorithm is known as dependency mediation.

Transitive Dependency



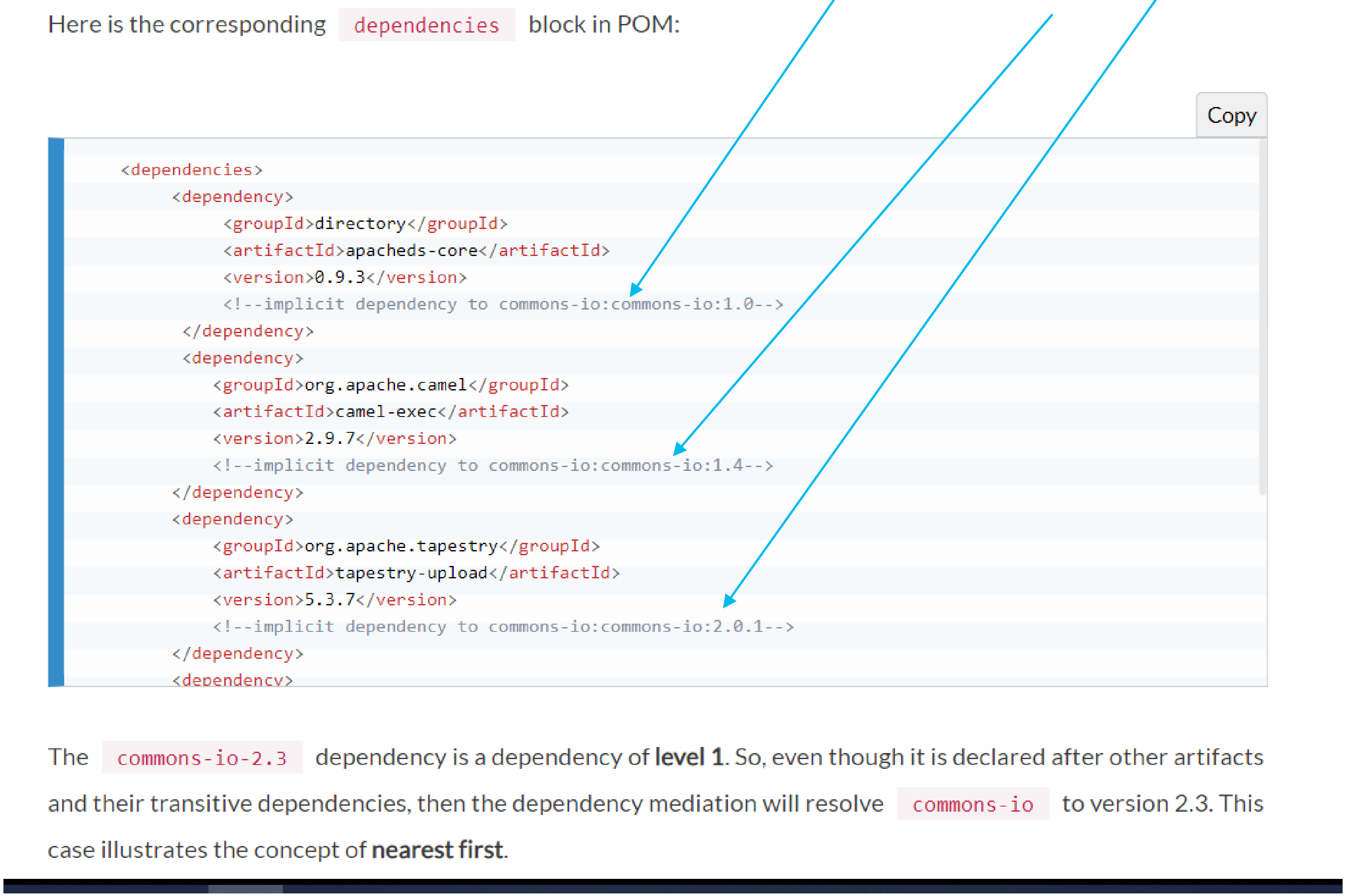
Transitive Dependency

```
<dependencies>
    <dependency>
       <groupId>directory</groupId>
       <artifactId>apacheds-core</artifactId>
       <version>0.9.3</version>
       <!--implicit dependency to commons-io:commons-io:1.0-->
     </dependency>
     <dependency>
      <groupId>org.apache.camel</groupId>
      <artifactId>camel-exec</artifactId>
      <version>2.9.7</version>
      <!--implicit dependency to commons-io:commons-io:1.4-->
    </dependency>
```

Transitive Dependency

```
<dependency>
      <groupId>org.apache.tapestry</groupId>
      <artifactId>tapestry-upload</artifactId>
      <version>5.3.7</version>
      <!--implicit dependency to commons-io:commons-io:2.0.1-->
    </dependency>
    <dependency>
      <groupId>com.googlecode.grep4j</groupId>
      <artifactId>grep4j</artifactId>
      <version>1.7.5</version>
      <!--implicit dependency to commons-io:commons-io:2.4-->
    </dependency>
    <dependency>
      <groupId>commons-io</groupId>
      <artifactId>commons-io</artifactId>
      <version>2.3</version>
    </dependency>
```

Transitive Dependency



Here is the corresponding `dependencies` block in POM:

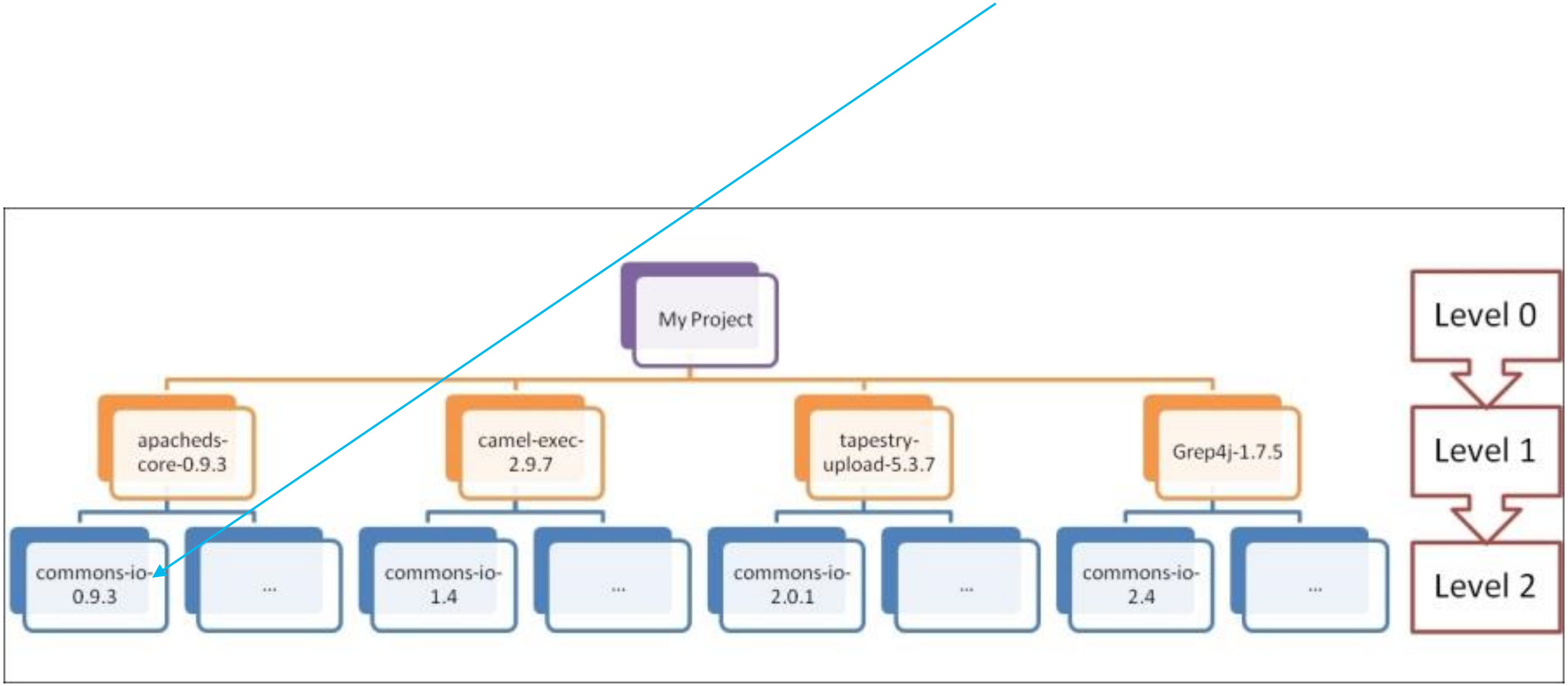
```
    <dependencies>
         <dependency>
              <groupId>directory</groupId>
              <artifactId>apacheds-core</artifactId>
              <version>0.9.3</version>
              <!--implicit dependency to commons-io:commons-io:1.0-->
          </dependency>
          <dependency>
             <groupId>org.apache.camel</groupId>
             <artifactId>camel-exec</artifactId>
             <version>2.9.7</version>
             <!--implicit dependency to commons-io:commons-io:1.4-->
         </dependency>
         <dependency>
             <groupId>org.apache.tapestry</groupId>
             <artifactId>tapestry-upload</artifactId>
             <version>5.3.7</version>
             <!--implicit dependency to commons-io:commons-io:2.0.1-->
         </dependency>
         <dependency>
```

The `commons-io-2.3` dependency is a dependency of **level 1**. So, even though it is declared after other artifacts and their transitive dependencies, then the dependency mediation will resolve `commons-io` to version 2.3. This case illustrates the concept of **nearest first**.

Transitive dependency

- The commons-io-2.3 dependency is a dependency of level 1.
- So, even though it is declared after other artifacts and their transitive dependencies, then the dependency mediation will resolve commons-io to version 2.3.
- This case illustrates the concept of **nearest first**.

Transitive Dependency



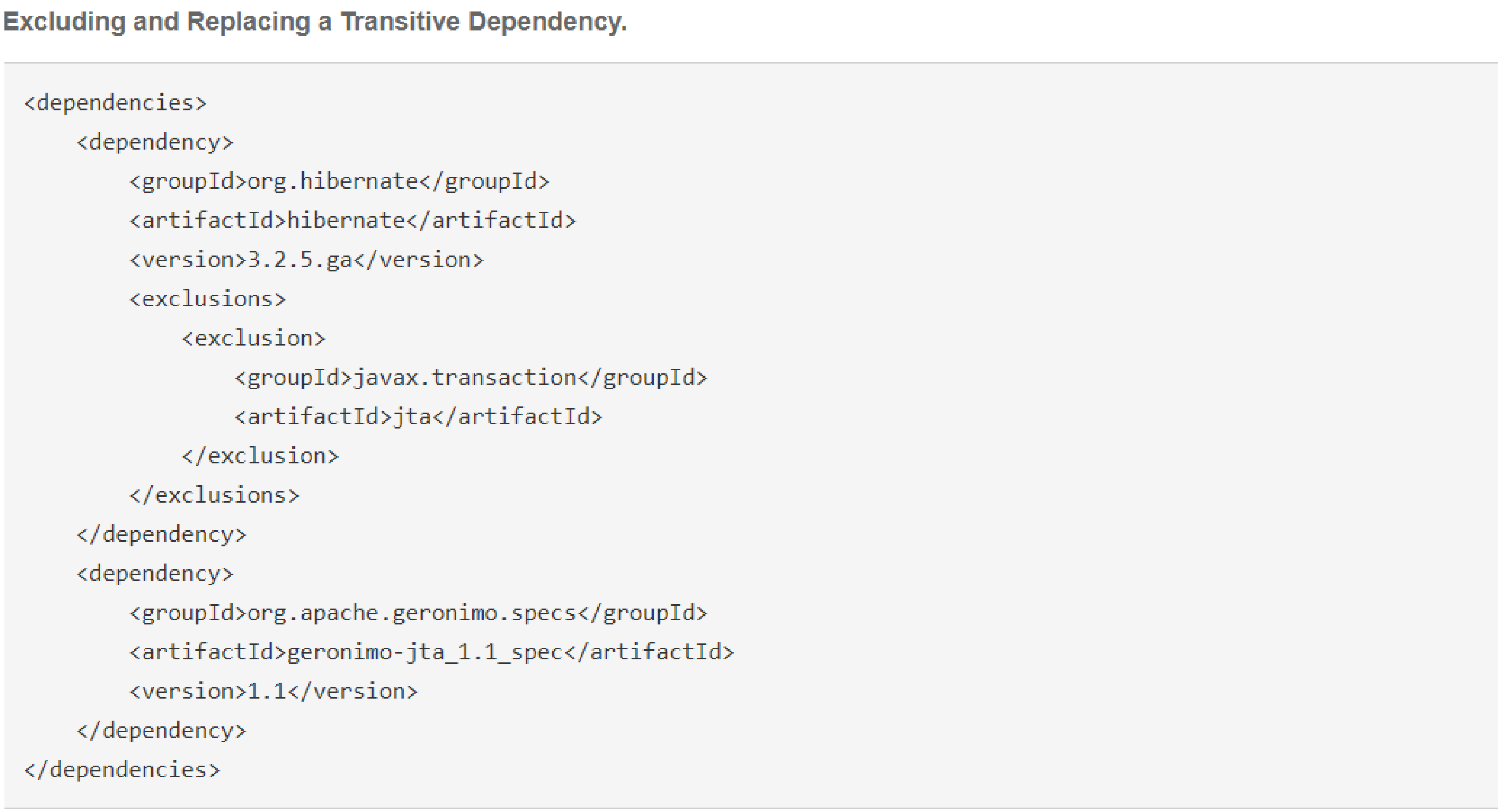
Transitive dependency

- All dependencies to commons-io are of level 2, and differ on the versions: 0.9.3 (via apacheds-core), 1.4 (via camel-exec), 2.0.1 (via tapestry-upload), and 2.4 (via grep4j).
- Unlike a popular belief, the resolution will not lead to take the greatest version number (that is, 2.4), but the first transitive version that appears in the dependency tree, in other terms 0.9.3.
- Had another dependency been declared before apacheds-core, its embed version of commons-io would have been resolved instead of version 0.9.3. This case illustrates the concept of **first found**.

Conflict Resolution

- There will be times when you need to exclude a transitive dependency, such as when you are depending on a project that depends on another project.
- But you would like to either exclude the dependency altogether or replace the transitive dependency with another dependency that provides the same functionality.
- Excluding a Transitive Dependency shows an example of a dependency element that adds a dependency on project-a, but excludes the transitive dependency project-b.

Conflict Resolution



Excluding and Replacing a Transitive Dependency.

```
<dependencies>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate</artifactId>
        <version>3.2.5.ga</version>
        <exclusions>
            <exclusion>
                <groupId>javax.transaction</groupId>
                <artifactId>jta</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.apache.geronimo.specs</groupId>
        <artifactId>geronimo-jta_1.1_spec</artifactId>
        <version>1.1</version>
    </dependency>
</dependencies>
```

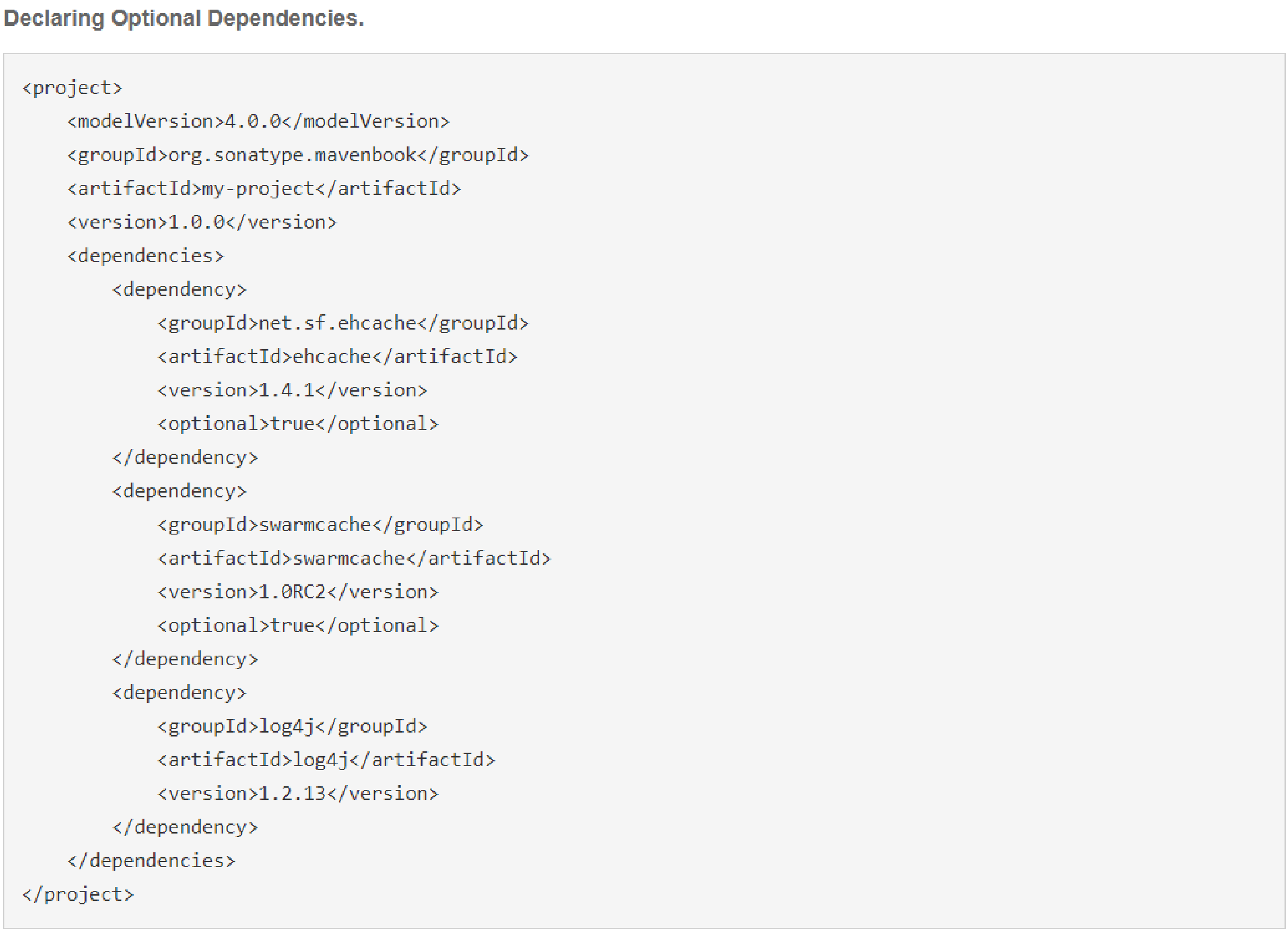
Optional Dependency

- Assume that you are working on a library that provides caching behavior.
- Instead of writing a caching system from scratch, you want to use some of the existing libraries that provide caching on the file system and distributed caches.
- Also assume that you want to give the end user an option to cache on the file system or to use an in-memory distributed cache.
- To cache on the file system, you'll want to use a freely available library called EHCache (http://ehcache.sourceforge.net/).
- To cache in a distributed in-memory cache, you want to use another freely available caching library named SwarmCache (http://swarmcache.sourceforge.net/).
- You'll code an interface and create a library that can be configured to use either EHCache or SwarmCache, but you want to avoid adding a dependency on both caching libraries to any project that depends on your library.

Optional Dependency

- In other words, you need both libraries to compile this library project, but you don’t want both libraries to show up as transitive runtime dependencies for the project that uses your library.
- You can accomplish this by using optional dependencies as shown in Declaring Optional Dependencies.

Optional Dependency



Declaring Optional Dependencies.

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.sonatype.mavenbook</groupId>
    <artifactId>my-project</artifactId>
    <version>1.0.0</version>
    <dependencies>
        <dependency>
            <groupId>net.sf.ehcache</groupId>
            <artifactId>ehcache</artifactId>
            <version>1.4.1</version>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>swarmcache</groupId>
            <artifactId>swarmcache</artifactId>
            <version>1.0RC2</version>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.13</version>
        </dependency>
    </dependencies>
</project>
```

Project Dependency Scope

- compile
  - This tells that dependency are needed for the compilation of main source files
  - Compile dependencies are available in all three classpaths mentioned above
  - These dependencies are propagated to dependent projects
  - Means transitive dependencies are included in the projects it is used and this can be cross-checked with project’s “*WEB-INF\lib*” folder
  - This is default scope, if nothing is specified in pom.*xml*

Maven dependency scope - compile

```xml
<dependencies>
    <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
        <version>1.2.14</version>
        <!-- You can omit this because it is default -->
        <scope>compile</scope>
    </dependency>
</dependencies>
```

Project Dependency Scope

- provided
  - This is similar to “*compile*” dependencies with only exception that, it’s not available in runtime-classpath.
  - It assumes that runtime environment like JDK or web container provides the required/dependent JARS for their executions after deployment
  - This is available only in compile-classpath and test-classpath
  - This is not transitive
  - So, once after building/packaging the project, we can neither find direct dependencies nor transitive dependencies in project’s “*WEB-INF\lib*” folder which are scoped to “*provided*”

Project Dependency Scope

- provided
  - For example, if you were developing a web application, you would need the Servlet API available on the compile classpath to compile a servlet, but you wouldn’t want to include the Servlet API in the packaged WAR; provided dependencies are available on the compilation classpath (not runtime). They are not transitive, nor are they packaged.

Maven dependency scope - provided

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>3.0.1</version>
    <scope>provided</scope>
</dependency>
```

Project Dependency Scope

- runtime
  - This dependency is not required for compilation, but very much required for their execution at runtime
  - Not exactly, but it is just opposite of “*provided*” scope which means dependency aren’t available in compile-classpath but available in runtime-classpath
  - This is available only in test-classpath and runtime-classpath
  - This is transitive in nature means transitive dependencies are packaged into project
  - We can cross-check at project’s “*WEB-INF\lib*” folder

Project Dependency Scope

  - runtime dependencies are required to execute and test the system, but they are not required for compilation.
  - For example, you may need a JDBC API JAR at compile time and the JDBC driver implementation only at runtime.
- Test
  - This dependency available for test compilation and for their executions
  - Available in only test-classpath
  - This is not transitive
  - If we cross-check, then direct & their transitive dependencies aren't available in project's "*WEB-INF\lib*" folder
  - Note: This is not for normal use of the application

Maven dependency scope - runtime

```
<dependency>
    <groupId>com.thoughtworks.xstream</groupId>
    <artifactId>xstream</artifactId>
    <version>1.4.4</version>
    <scope>runtime</scope>
 </dependency>
```

Maven dependency scope - test

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

Project Dependency Scope

  - test-scoped dependencies are not required during the normal operation of an application, and they are available only during test compilation and execution phases.

- system
  - The system scope is similar to provided except that you have to provide an explicit path to the JAR on the local file system. This is intended to allow compilation against native objects that may be part of the system libraries.
  - Import
    - import scope is only supported on a dependency of type pom in the dependencyManagement section. It indicates the dependency to be replaced with the effective list of dependencies in the specified POM’s dependencyManagement section.

Maven dependency scope - system

```
<dependency>
  <groupId>extDependency</groupId>
  <artifactId>extDependency</artifactId>
  <scope>system</scope>
  <version>1.0</version>
  <systemPath>${basedir}\war\WEB-INF\lib\extDependency.jar</systemPath>
</dependency>
```

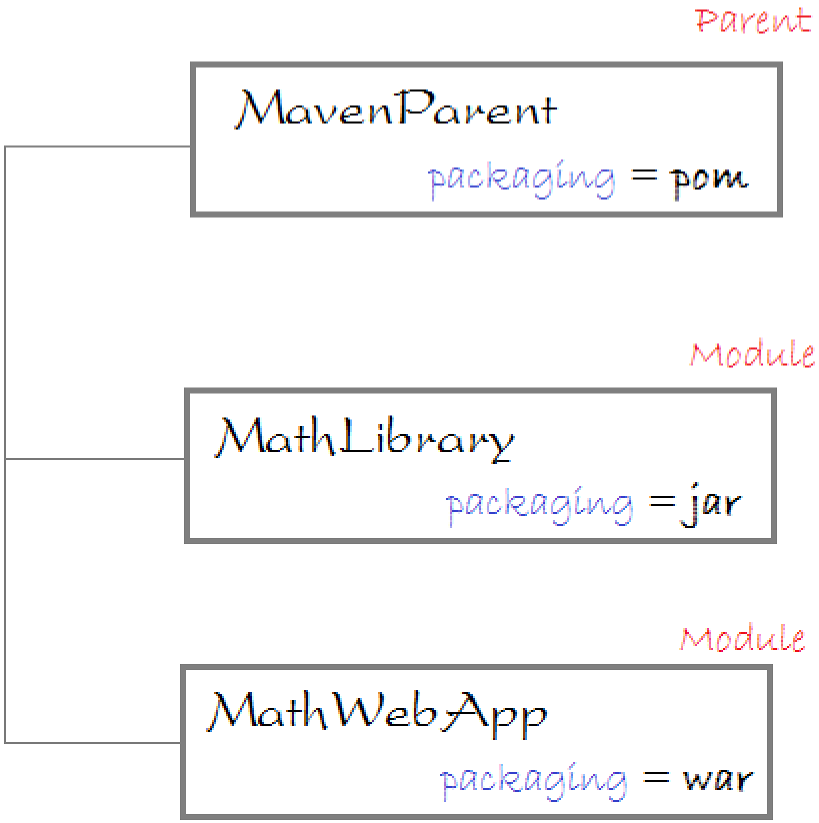
Maven dependency scope – import

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>other.pom.group.id</groupId>
            <artifactId>other-pom-artifact-id</artifactId>
            <version>SNAPSHOT</version>
            <scope>import</scope>
            <type>pom</type>
        </dependency>
    </dependencies>
</dependencyManagement>
```

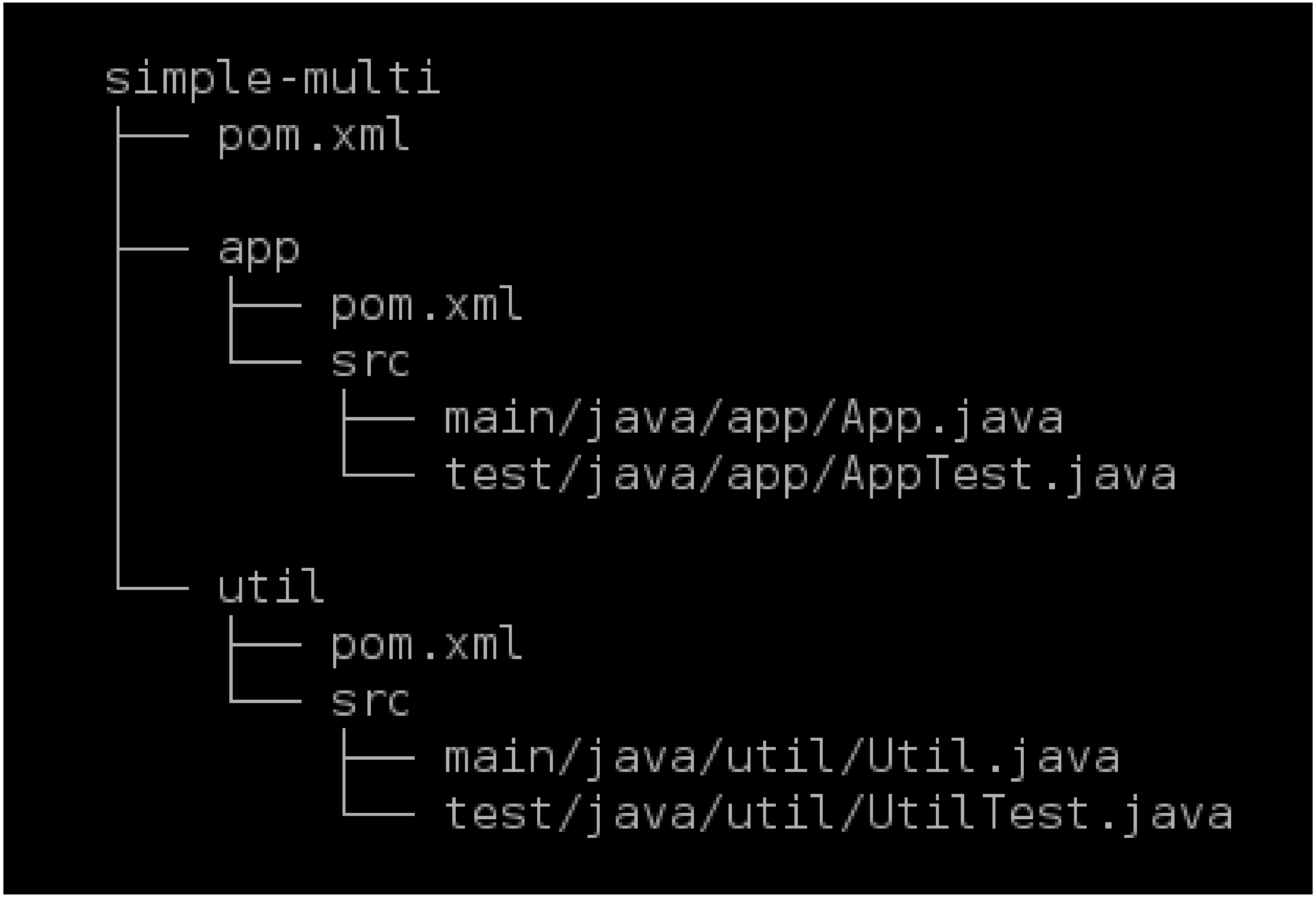
Maven Multi Module

- As project size and complexity increases, it makes sense to split the project into multiple modules.
- While it is always possible to split the project into multiple projects and link them as dependencies, it complicates the build process.
- In Maven, the preferred approach is to structure the project as multi module project and delegate the rest to Maven.

Multi Module Project



Maven Multi Module



```
simple-multi
├── pom.xml
│
├── app
│   ├── pom.xml
│   └── src
│       ├── main/java/app/App.java
│       └── test/java/app/AppTest.java
│
└── util
    ├── pom.xml
    └── src
        ├── main/java/util/Util.java
        └── test/java/util/UtilTest.java
```

Maven Multi Module

simple-multi/pom.xml

```
<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>org.codetab</groupId>

  <artifactId>simple-multi</artifactId>

  <version>1.0</version>

  <packaging>pom</packaging>

  <modules>

      <module>app</module>

      <module>util</module>

  </modules></project>
```

Maven Multi Module

Util/pom.xml

```xml
<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>

  <parent>

     <groupId>org.codetab</groupId>

     <artifactId>simple-multi</artifactId>

     <version>1.0</version>

  </parent>

  <artifactId>util</artifactId>

  <properties>
```

Maven Multi Module

```
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      <maven.compiler.source>1.8</maven.compiler.source>
      <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
  <dependencies>
      <dependency>
          <groupId>org.apache.commons</groupId>
          <artifactId>commons-lang3</artifactId>
          <version>3.6</version>
      </dependency>
```

Maven Multi Module

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

POM

- General project information
- This includes a project’s name, the URL for a project, the sponsoring organization, and a list of developers and contributors along with the license for a project.
- Build settings
- In this section, we customize the behaviour of the default Maven build.
- We can change the location of source and tests, we can add new plugins, we can attach plugin goals to the lifecycle, and we can customize the site generation parameters.

POM

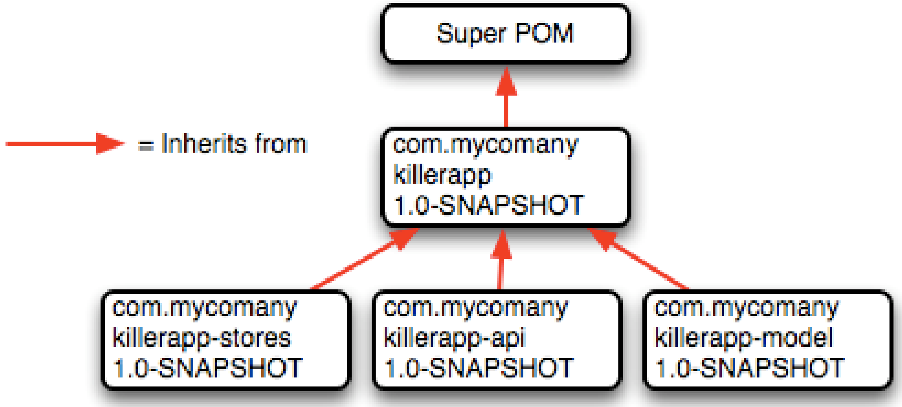
- Build environment
- The build environment consists of profiles that can be activated for use in different environments.
- For example, during development you may want to deploy to a development server, whereas in production you want to deploy to a production server.
- The build environment customizes the build settings for specific environments and is often supplemented by a custom settings.xml in ~/.m2.
- POM relationships
- A project rarely stands alone; it depends on other projects, inherits POM settings from parent projects, defines its own coordinates, and may include submodules.

The Super POM

- The Super POM is Maven's default POM.
- All POMs inherit from a parent or default (despite explicitly defined or not).
- This base POM is known as the Super POM, and contains values inherited by default.
- Ultimately, all the application POM files get extended from the super POM.
- The super POM file is at the top of the POM hierarchy and is bundled inside MAVEN_HOME/lib/maven-model-builder-3.3.3.jar - org/apache/maven/model/pom-4.0.0.xml.

The Super POM

- All the default configurations are defined in the super POM file.
- Even the simplest form of a POM file will inherit all the configurations defined in the super POM file.
- Whatever configuration you need to override, you can do it by redefining the same section in your application POM file.



The Super POM

- All Maven project
- POMs extend the Super POM, which defines a set of defaults shared by all projects.
- This Super POM is a part of the Maven installation.
- Depending on the Maven version it can be found in the maven-x.y.z-uber.jar or maven-model-builder-xy.z.jar file in ${M2_HOME}/lib.

Minimum POM

- The minimum requirement for a POM are the following:
  - project root
  - modelVersion - should be set to 4.0.0
  - groupId - the id of the project's group.
  - artifactId - the id of the artifact (project)
  - version - the version of the artifact under the specified group

The Simplest POM.

```
<project> <modelVersion>4.0.0</modelVersion>
<groupId>org.sonatype.mavenbook.ch08</groupId>
<artifactId>simplest-project</artifactId>
<version>1</version> </project>
```

Effective POM

- The merge between the Super POM and the POM from The Simplest POM.
- mvn help:effective-pom
- Executing the effective-pom goal should print out an XML document capturing the merge between the Super POM and the POM from The Simplest POM.

What is Maven's order of inheritance?

1. parent pom
2. project pom
3. settings
4. CLI parameters

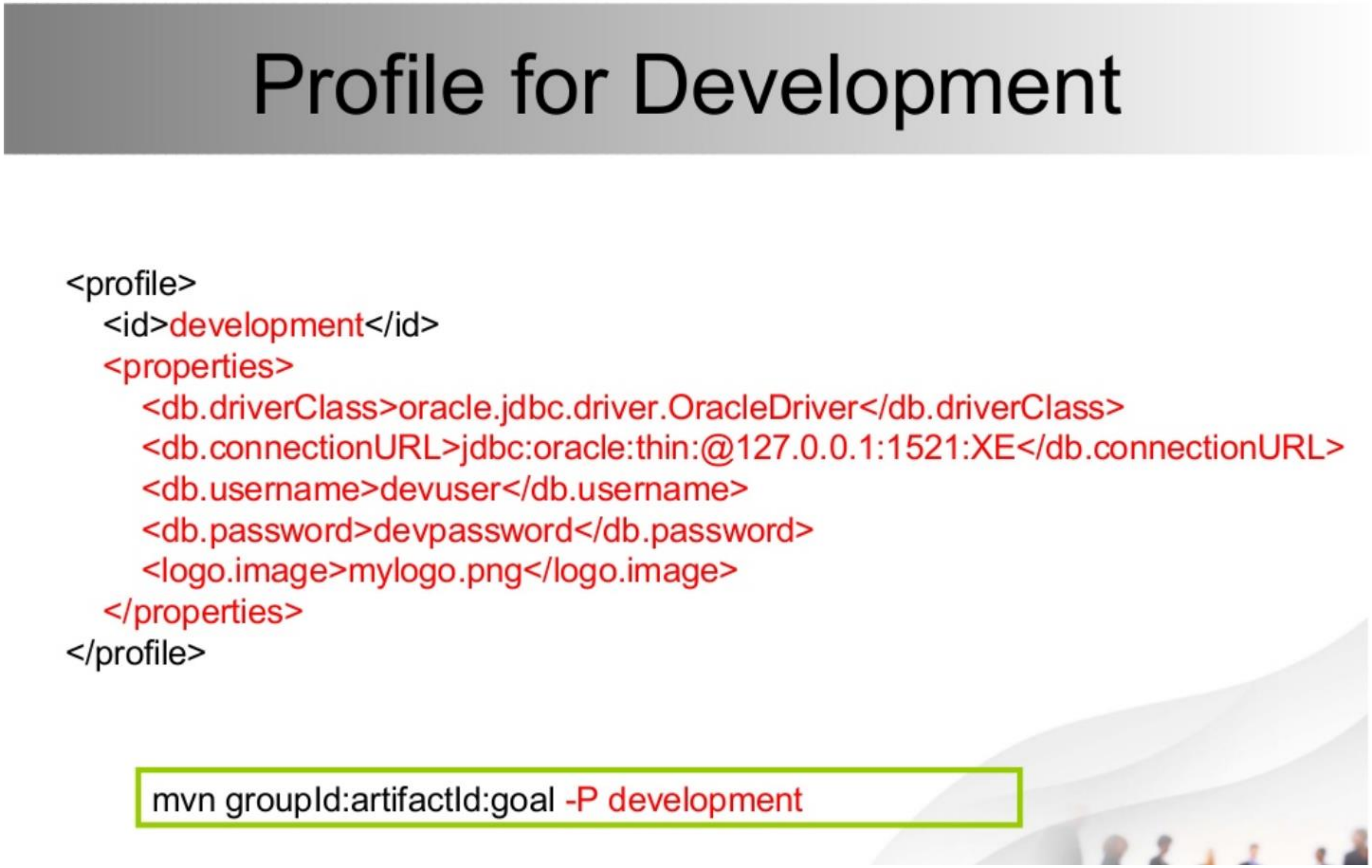
What is Build Profile?

- A Build profile is a set of configuration values, which can be used to set or override default values of Maven build.
- Using a build profile, you can customize build for different environments such as Production v/s Development environments.

Types of Build Profile

Type	Where it is defined
Per Project	Defined in the project POM file, pom.xml
Per User	Defined in Maven settings xml file (%USER_HOME%/.m2/settings.xml)
Global	Defined in Maven global settings xml file (%M2_HOME%/conf/settings.xml)

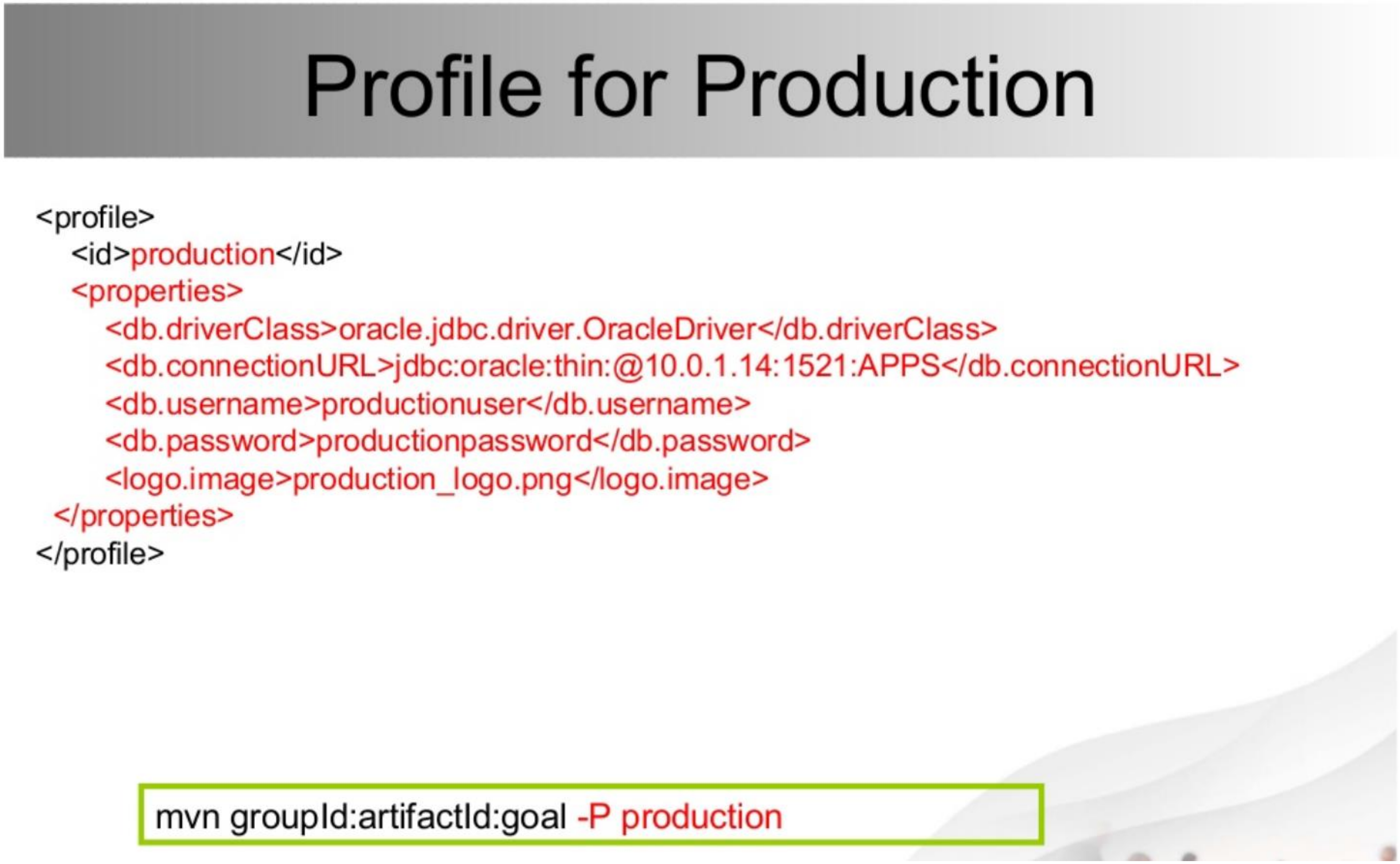
Types of Build Profile



Profile for Development

```
<profile>
   <id>development</id>
   <properties>
      <db.driverClass>oracle.jdbc.driver.OracleDriver</db.driverClass>
      <db.connectionURL>jdbc:oracle:thin:@127.0.0.1:1521:XE</db.connectionURL>
      <db.username>devuser</db.username>
      <db.password>devpassword</db.password>
      <logo.image>mylogo.png</logo.image>
   </properties>
</profile>
```

```
mvn groupId:artifactId:goal -P development
```



Profile for Production

```
<profile>
   <id>production</id>
   <properties>
      <db.driverClass>oracle.jdbc.driver.OracleDriver</db.driverClass>
      <db.connectionURL>jdbc:oracle:thin:@10.0.1.14:1521:APPS</db.connectionURL>
      <db.username>productionuser</db.username>
      <db.password>productionpassword</db.password>
      <logo.image>production_logo.png</logo.image>
  </properties>
</profile>
```

```
mvn groupId:artifactId:goal -P production
```

Resource Filtering

Enable resource filtering in specific directory
maven variable in files in the specific directory will be replaced

add plugin

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-resources-plugin</artifactId>
   <version>2.3</version>
   <configuration>
      <encoding>UTF-8</encoding>
   </configuration>
</plugin>
```

configure resource filtering

```
<resources>
   <resource>
      <directory>src/main/resources</directory>
      <filtering>true</filtering>
   </resource>
</resources>
```

Resource Filtering

db.properties source

```
database.jdbc.driverClass=${db.driverClass}
database.jdbc.connectionURL=${db.connectionURL}
database.jdbc.username=${db.username}
database.jdbc.password=${db.password}

model.application.name=MyWebApp
model.stylesheet=/mywebapp.css
```

db.properties source after resource filtering

```
database.jdbc.driverClass=oracle.jdbc.driver.OracleDriver
database.jdbc.connectionURL=jdbc:oracle:thin:@127.0.0.1:1521:XE
database.jdbc.username=myusername
database.jdbc.password=mypassword

model.application.name=MyWebApp
model.stylesheet=/mywebapp.css
```

Profile Activation

actived if env = dev

```
<profiles>
 <profile>
  <id>appserverConfig-dev</id>
  <activation>
   <property>
    <name>env</name>
    <value>dev</value>
   </property>
  </activation>
  <properties>
   <server.home>/appserver</server.home>
  </properties>
 </profile>
<profiles>
```

actived if env = dev2

```
<profiles>
 <profile>
  <id>appserverConfig-dev-2</id>
  <activation>
   <property>
    <name>env</name>
    <value>dev-2</value>
   </property>
  </activation>
  <properties>
   <server.home>/appserver2</server.home>
  </properties>
 </profile>
<profiles>
```

Profile Activation

- Explicitly using command console input.
- Through maven settings.
- Based on environment variables (User/System variables).
- OS Settings (for example, Windows family).

Maven cheat sheet

- Maven Switches

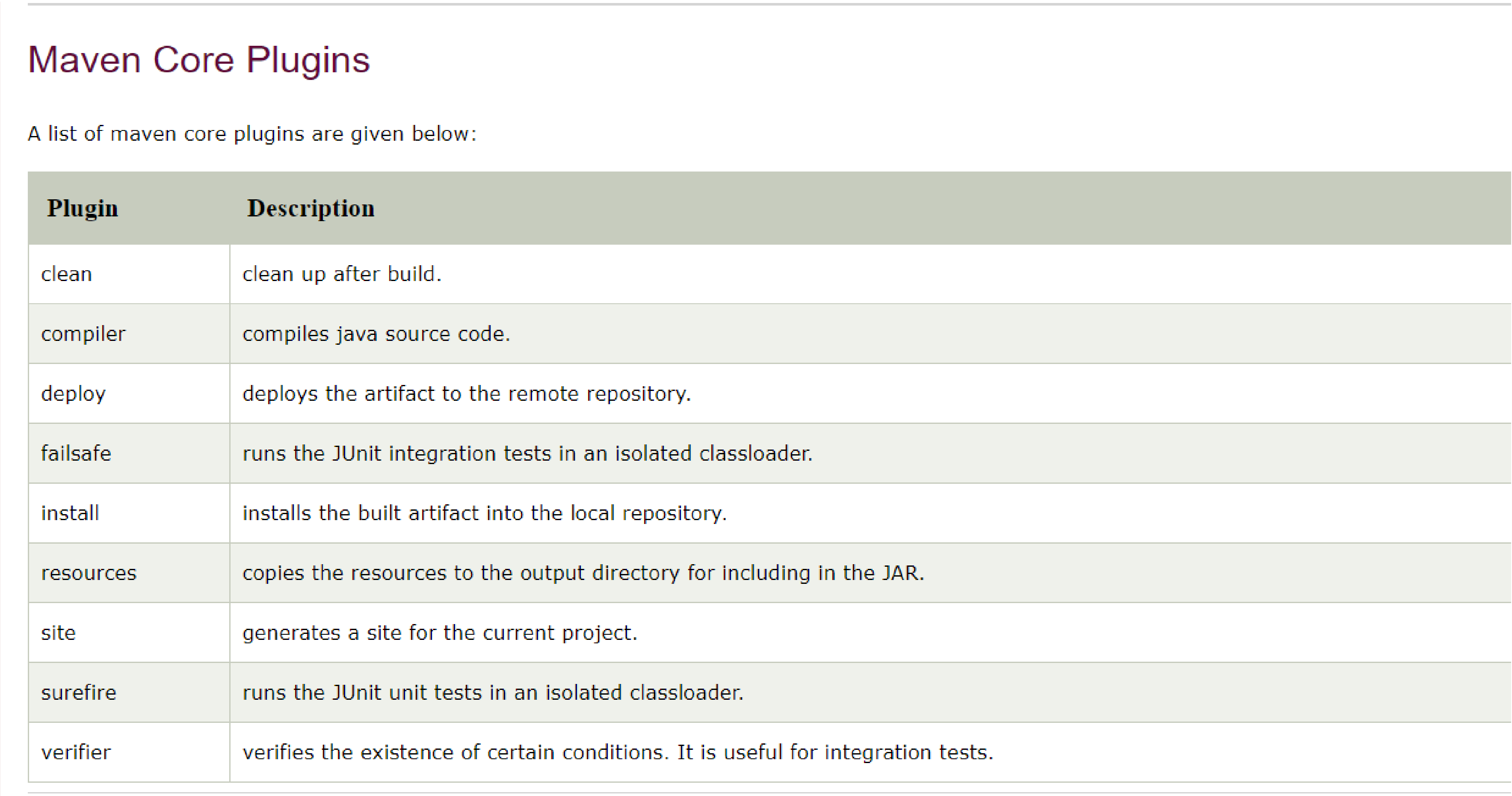
Maven Plugins

- The maven plugins are central part of maven framework, it is used to perform specific goal.
- According to Apache Maven, there are 2 types of maven plugins.
- Build Plugins
- Reporting Plugins

Maven Plugins

- Build Plugins
  - These plugins are executed at the time of build. These plugins should be declared inside the <build> element.
- Reporting Plugins
  - These plugins are executed at the time of site generation. These plugins should be declared inside the <reporting> element.

Maven Plugins

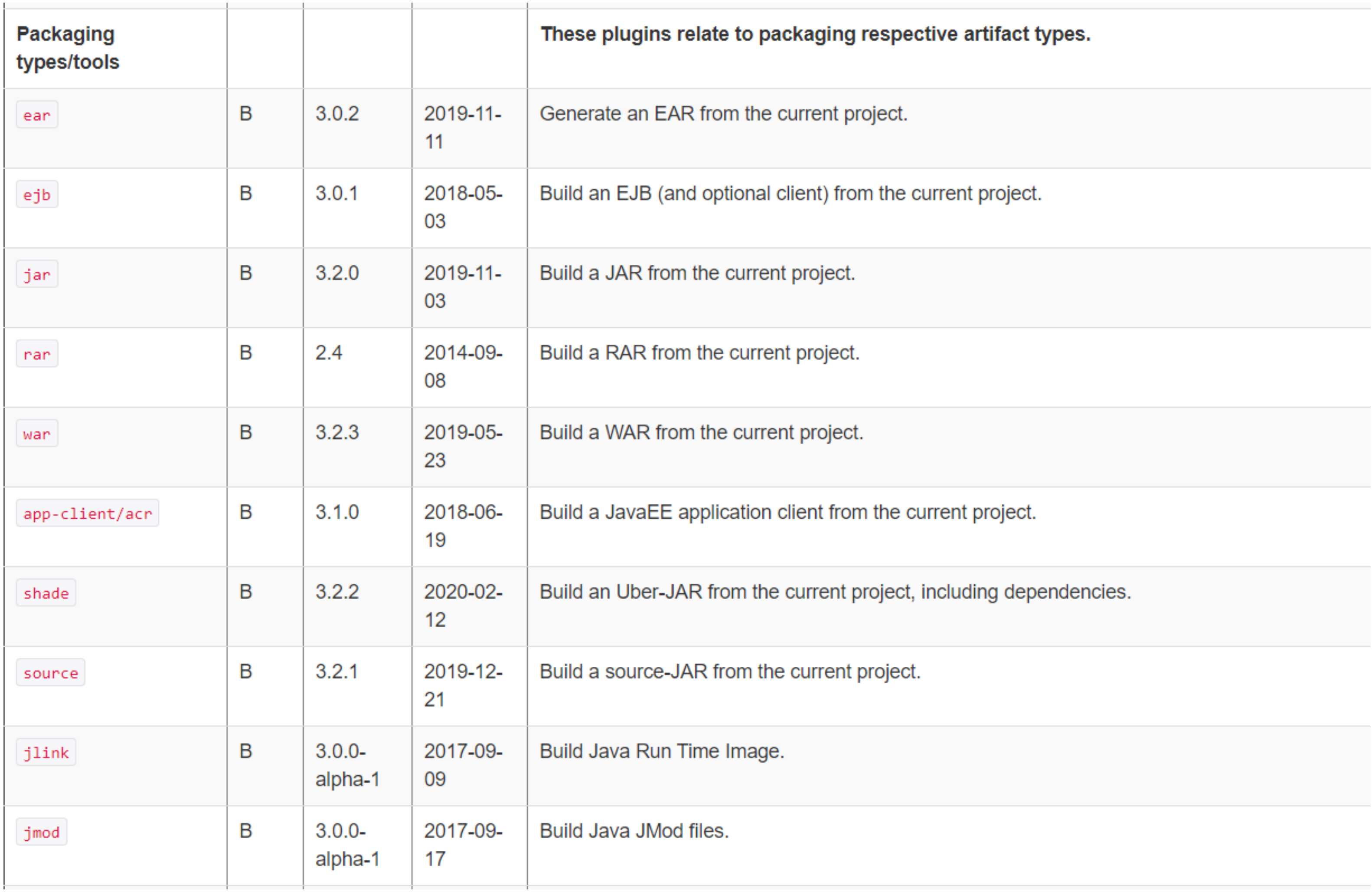


Maven Core Plugins

A list of maven core plugins are given below:

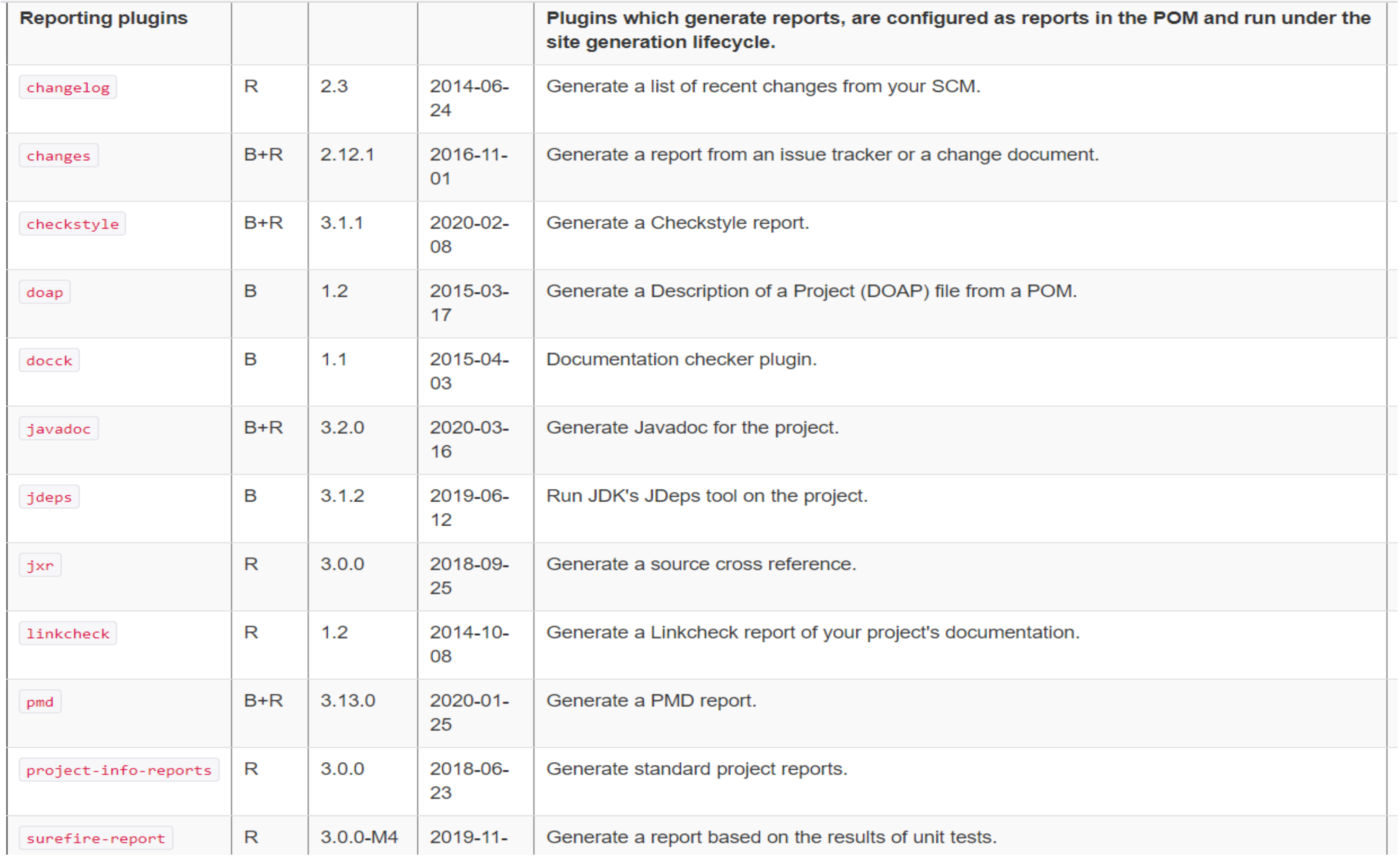
Plugin	Description
clean	clean up after build.
compiler	compiles java source code.
deploy	deploys the artifact to the remote repository.
failsafe	runs the JUnit integration tests in an isolated classloader.
install	installs the built artifact into the local repository.
resources	copies the resources to the output directory for including in the JAR.
site	generates a site for the current project.
surefire	runs the JUnit unit tests in an isolated classloader.
verifier	verifies the existence of certain conditions. It is useful for integration tests.

Maven Plugins



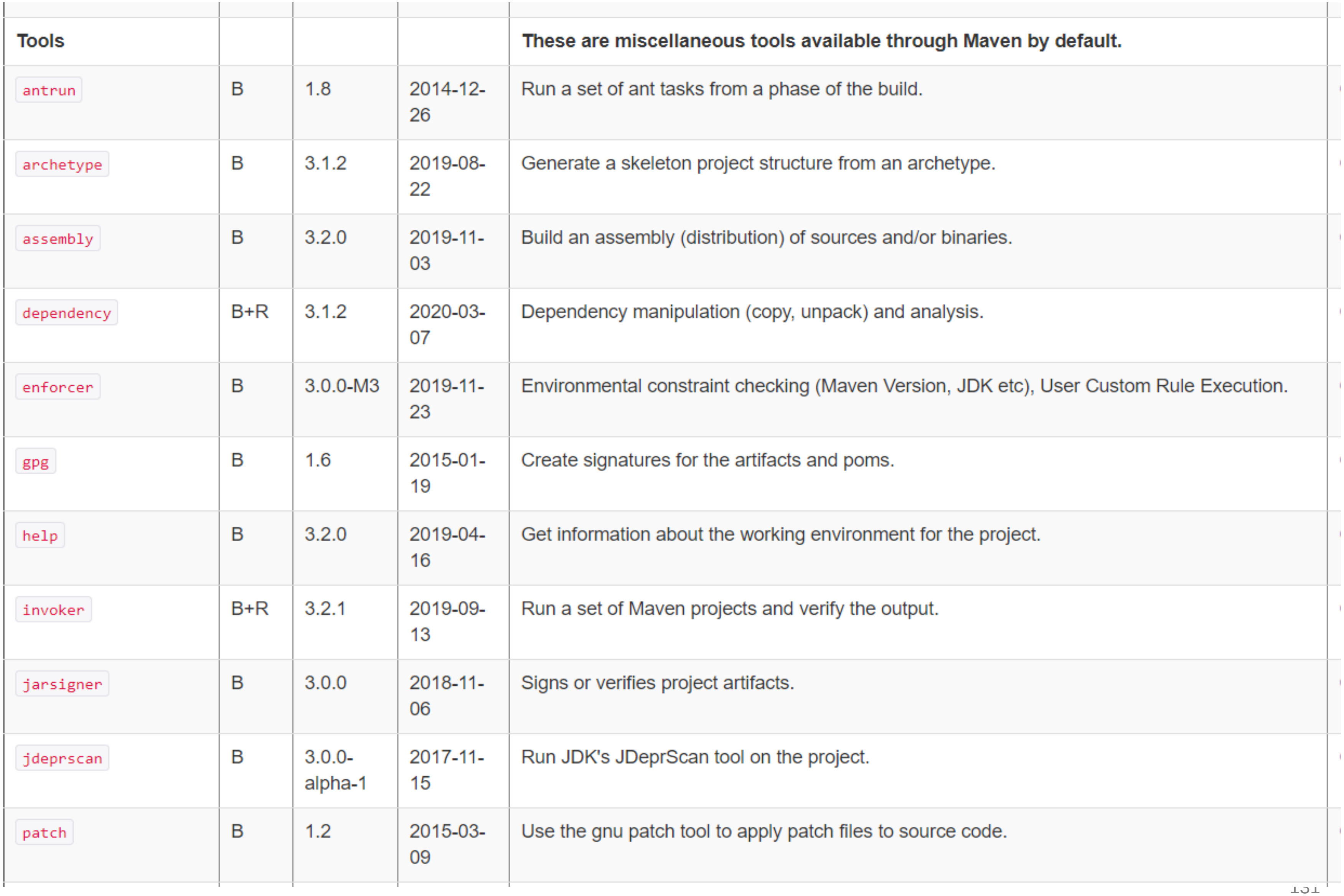
Packaging types/tools				**These plugins relate to packaging respective artifact types.**
`ear`	B	3.0.2	2019-11-11	Generate an EAR from the current project.
`ejb`	B	3.0.1	2018-05-03	Build an EJB (and optional client) from the current project.
`jar`	B	3.2.0	2019-11-03	Build a JAR from the current project.
`rar`	B	2.4	2014-09-08	Build a RAR from the current project.
`war`	B	3.2.3	2019-05-23	Build a WAR from the current project.
`app-client/acr`	B	3.1.0	2018-06-19	Build a JavaEE application client from the current project.
`shade`	B	3.2.2	2020-02-12	Build an Uber-JAR from the current project, including dependencies.
`source`	B	3.2.1	2019-12-21	Build a source-JAR from the current project.
`jlink`	B	3.0.0-alpha-1	2017-09-09	Build Java Run Time Image.
`jmod`	B	3.0.0-alpha-1	2017-09-17	Build Java JMod files.

Maven Plugins



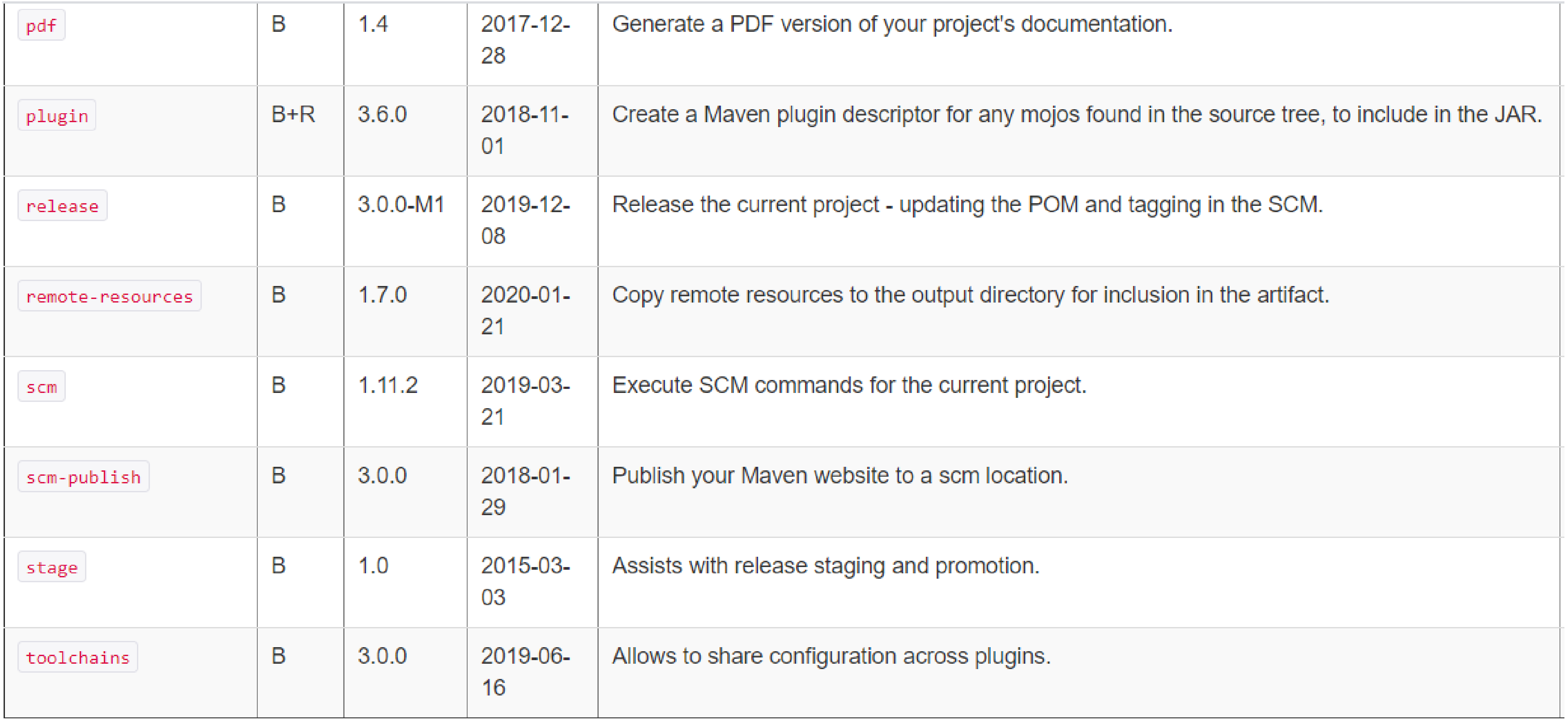
Reporting plugins				**Plugins which generate reports, are configured as reports in the POM and run under the site generation lifecycle.**
`changelog`	R	2.3	2014-06-24	Generate a list of recent changes from your SCM.
`changes`	B+R	2.12.1	2016-11-01	Generate a report from an issue tracker or a change document.
`checkstyle`	B+R	3.1.1	2020-02-08	Generate a Checkstyle report.
`doap`	B	1.2	2015-03-17	Generate a Description of a Project (DOAP) file from a POM.
`docck`	B	1.1	2015-04-03	Documentation checker plugin.
`javadoc`	B+R	3.2.0	2020-03-16	Generate Javadoc for the project.
`jdeps`	B	3.1.2	2019-06-12	Run JDK's JDeps tool on the project.
`jxr`	R	3.0.0	2018-09-25	Generate a source cross reference.
`linkcheck`	R	1.2	2014-10-08	Generate a Linkcheck report of your project's documentation.
`pmd`	B+R	3.13.0	2020-01-25	Generate a PMD report.
`project-info-reports`	R	3.0.0	2018-06-23	Generate standard project reports.
`surefire-report`	R	3.0.0-M4	2019-11-	Generate a report based on the results of unit tests.

Maven Plugins



Tools				**These are miscellaneous tools available through Maven by default.**
`antrun`	B	1.8	2014-12-26	Run a set of ant tasks from a phase of the build.
`archetype`	B	3.1.2	2019-08-22	Generate a skeleton project structure from an archetype.
`assembly`	B	3.2.0	2019-11-03	Build an assembly (distribution) of sources and/or binaries.
`dependency`	B+R	3.1.2	2020-03-07	Dependency manipulation (copy, unpack) and analysis.
`enforcer`	B	3.0.0-M3	2019-11-23	Environmental constraint checking (Maven Version, JDK etc), User Custom Rule Execution.
`gpg`	B	1.6	2015-01-19	Create signatures for the artifacts and poms.
`help`	B	3.2.0	2019-04-16	Get information about the working environment for the project.
`invoker`	B+R	3.2.1	2019-09-13	Run a set of Maven projects and verify the output.
`jarsigner`	B	3.0.0	2018-11-06	Signs or verifies project artifacts.
`jdeprscan`	B	3.0.0-alpha-1	2017-11-15	Run JDK's JDeprScan tool on the project.
`patch`	B	1.2	2015-03-09	Use the gnu patch tool to apply patch files to source code.

Maven Plugins



`pdf`	B	1.4	2017-12-28	Generate a PDF version of your project's documentation.
`plugin`	B+R	3.6.0	2018-11-01	Create a Maven plugin descriptor for any mojos found in the source tree, to include in the JAR.
`release`	B	3.0.0-M1	2019-12-08	Release the current project - updating the POM and tagging in the SCM.
`remote-resources`	B	1.7.0	2020-01-21	Copy remote resources to the output directory for inclusion in the artifact.
`scm`	B	1.11.2	2019-03-21	Execute SCM commands for the current project.
`scm-publish`	B	3.0.0	2018-01-29	Publish your Maven website to a scm location.
`stage`	B	1.0	2015-03-03	Assists with release staging and promotion.
`toolchains`	B	3.0.0	2019-06-16	Allows to share configuration across plugins.

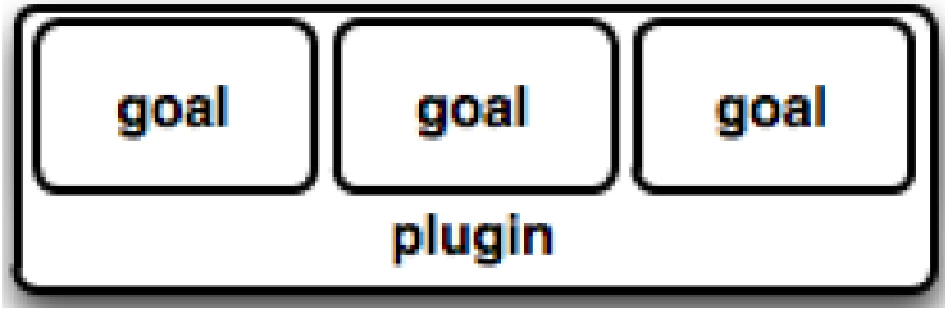
Maven Plugins (Hands on)

- <reporting>
- <plugins>
- <plugin>
- <groupId>org.apache.maven.plugins</groupId>
- <!-- Programming Mistake Detector -->
- <artifactId>maven-pmd-plugin</artifactId>
- <version>3.11.0</version>
- </plugin>
- </plugins>
- </reporting>
- Mvn site

Maven Plugins and Goals

- To execute a single Maven plugin goal, we used the syntax mvn archetype:generate, where archetype is the identifier of a plugin and generate is the identifier of a goal.
- $ mvn archetype:generate -DgroupId=org.sonatype.mavenbook.simple
- A Maven Plugin is a collection of one or more goals.
- Examples of Maven plugins can be simple core plugins like the Jar plugin, which contains goals for creating JAR files, Compiler plugin, which contains goals for compiling source code and unit tests, or the Surefire plugin, which contains goals for executing unit tests and generating reports.

Maven Plugins and Goals



- A goal is a specific task that may be executed as a standalone goal or along with other goals as part of a larger build.
- A goal is a “unit of work” in Maven.

Maven Repositories

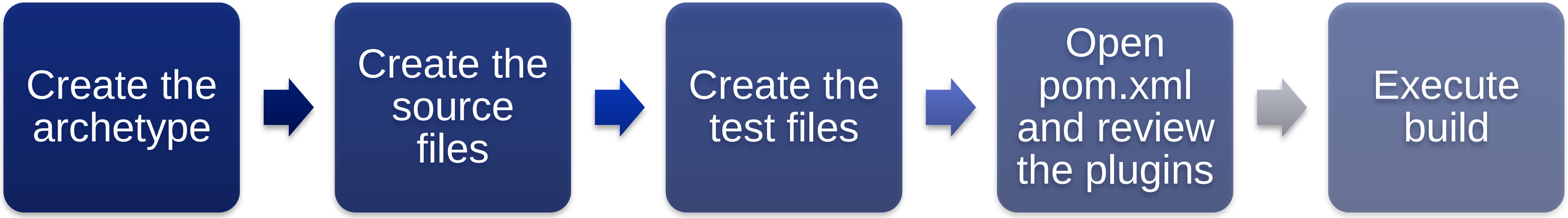
- Maven repositories store a set of artifacts which are used by Maven during dependency resolution for a project.
- Local repositories can be accessed on the local hard disk.
- Remote repositories can be accessed through the network.
- An artifact is bundled as a JAR file which contains the binary library or executable.
- An artifact can also be a war or an ear.

Maven 3 Archetypes

- An archetype is a complete project template
- Using an archetype, a project template can be created with a simple command
- Following is a list of archetypes and their purposes.

Archetype	Purpose
maven-archetype-archetype	To create our own project template (archetype)
Maven-archetype-j2ee-simple	To create a J2EE project (EAR).
Maven-archetype-webapp	To create a web application (WAR). This contains a HelloWorld JSP
Maven-archetype-quickstart	To create a simple Java project. Can be used to generate JAR. Default of Maven 2.

First Maven 3 project



Create the archetype

Create the source files

Create the test files

Open pom.xml and review the plugins

Execute build

First Maven 3 project

- In order to create everything which is needed for a simple Java project that can be built using Maven, you can use **Archetype** plug-in.
- Archetype is a standard plug-in which comes with Maven3.
- The Archetype plug-in runs outside of a Maven project build life cycle and is used for creating Maven projects.

First Maven 3 project

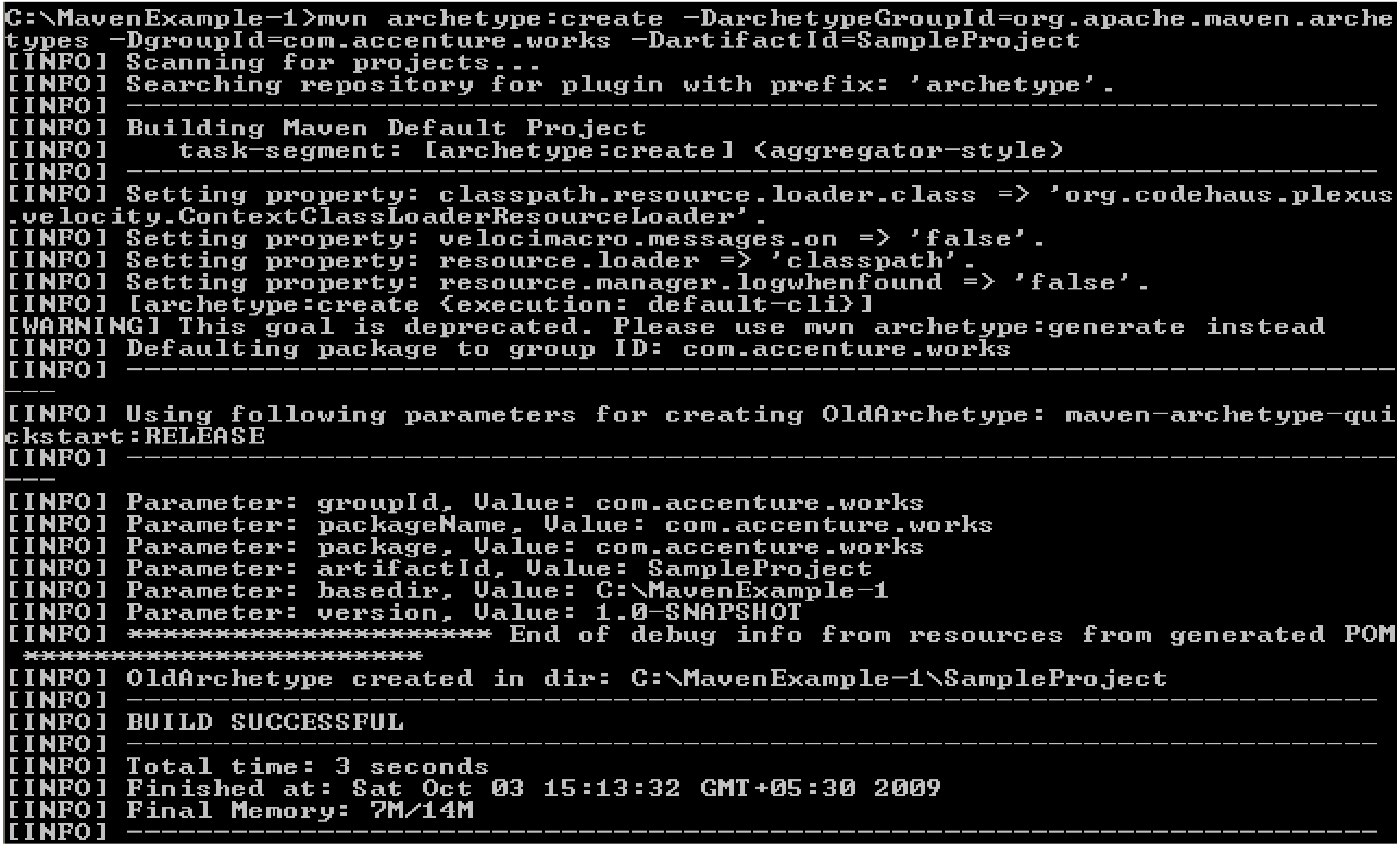
- Issue the following command from the directory that you want to contain **SampleProject**:

```
mvn archetype:generate -DgroupId=com.cgi.banking  -DartifactId=BankingAppln -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

- The above command provides the archetype plug-in with the coordinates of your module.
- The command also creates a starter pom.xml file for the project along with the conventional Maven 3 directory structure.

First Maven 3 project (Continued…)

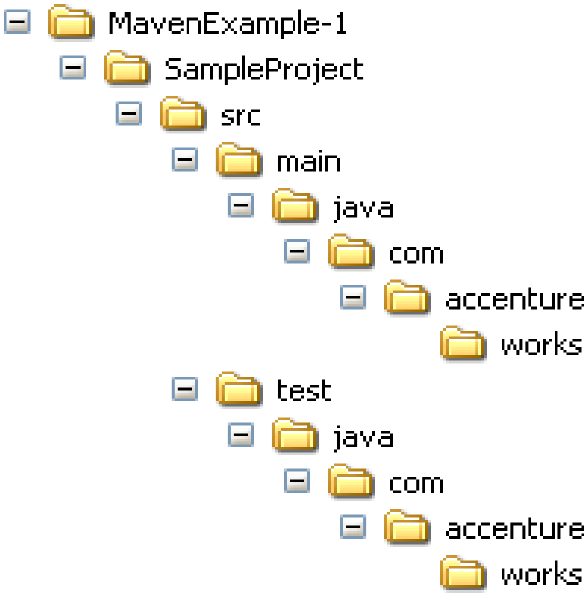
- The output should be similar to the following:



```
C:\MavenExample-1>mvn archetype:create -DarchetypeGroupId=org.apache.maven.arche
types -DgroupId=com.accenture.works -DartifactId=SampleProject
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Default Project
[INFO]    task-segment: [archetype:create] (aggregator-style)
[INFO] ------------------------------------------------------------------------
[INFO] Setting property: classpath.resource.loader.class => 'org.codehaus.plexus
.velocity.ContextClassLoaderResourceLoader'.
[INFO] Setting property: velocimacro.messages.on => 'false'.
[INFO] Setting property: resource.loader => 'classpath'.
[INFO] Setting property: resource.manager.logwhenfound => 'false'.
[INFO] [archetype:create {execution: default-cli}]
[WARNING] This goal is deprecated. Please use mvn archetype:generate instead
[INFO] Defaulting package to group ID: com.accenture.works
[INFO] -------------------------------------------------------------------------
---
[INFO] Using following parameters for creating OldArchetype: maven-archetype-qui
ckstart:RELEASE
[INFO] -------------------------------------------------------------------------
---
[INFO] Parameter: groupId, Value: com.accenture.works
[INFO] Parameter: packageName, Value: com.accenture.works
[INFO] Parameter: package, Value: com.accenture.works
[INFO] Parameter: artifactId, Value: SampleProject
[INFO] Parameter: basedir, Value: C:\MavenExample-1
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] ********************* End of debug info from resources from generated POM
 ***********************
[INFO] OldArchetype created in dir: C:\MavenExample-1\SampleProject
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 3 seconds
[INFO] Finished at: Sat Oct 03 15:13:32 GMT+05:30 2009
[INFO] Final Memory: 7M/14M
[INFO] ------------------------------------------------------------------------
```

First Maven 3 project (Continued…)

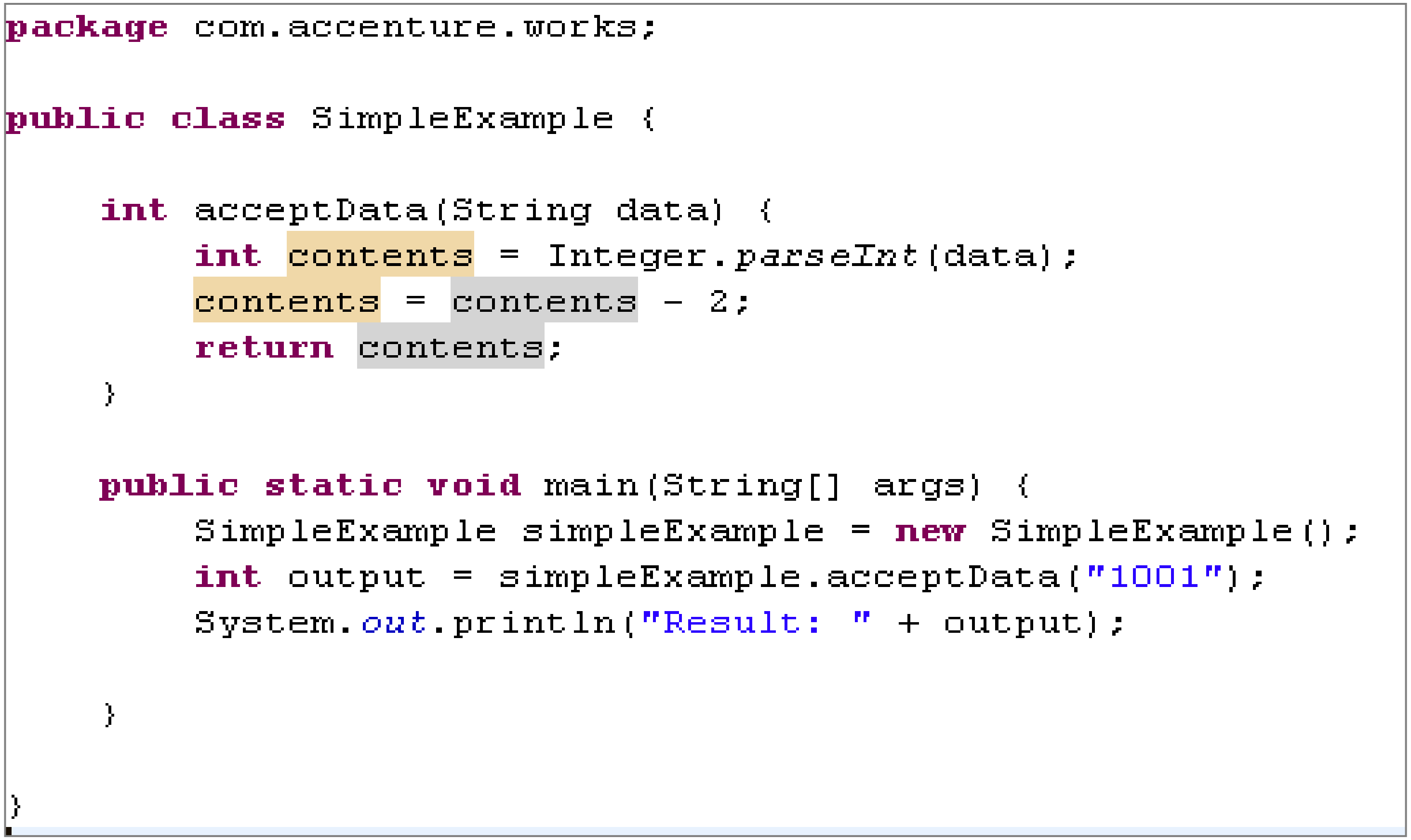
- The Archetype plug-in creates a directory tree, a pom.xml file and a placeholder App.java application.
- It also creates a directory tree for unit test source code and a placeholder AppTest.java unit test.
- The following figures shows the directory structure created:



```
MavenExample-1
  SampleProject
    src
      main
        java
          com
            accenture
              works
      test
        java
          com
            accenture
              works
```

First Maven 3 project (Continued…)

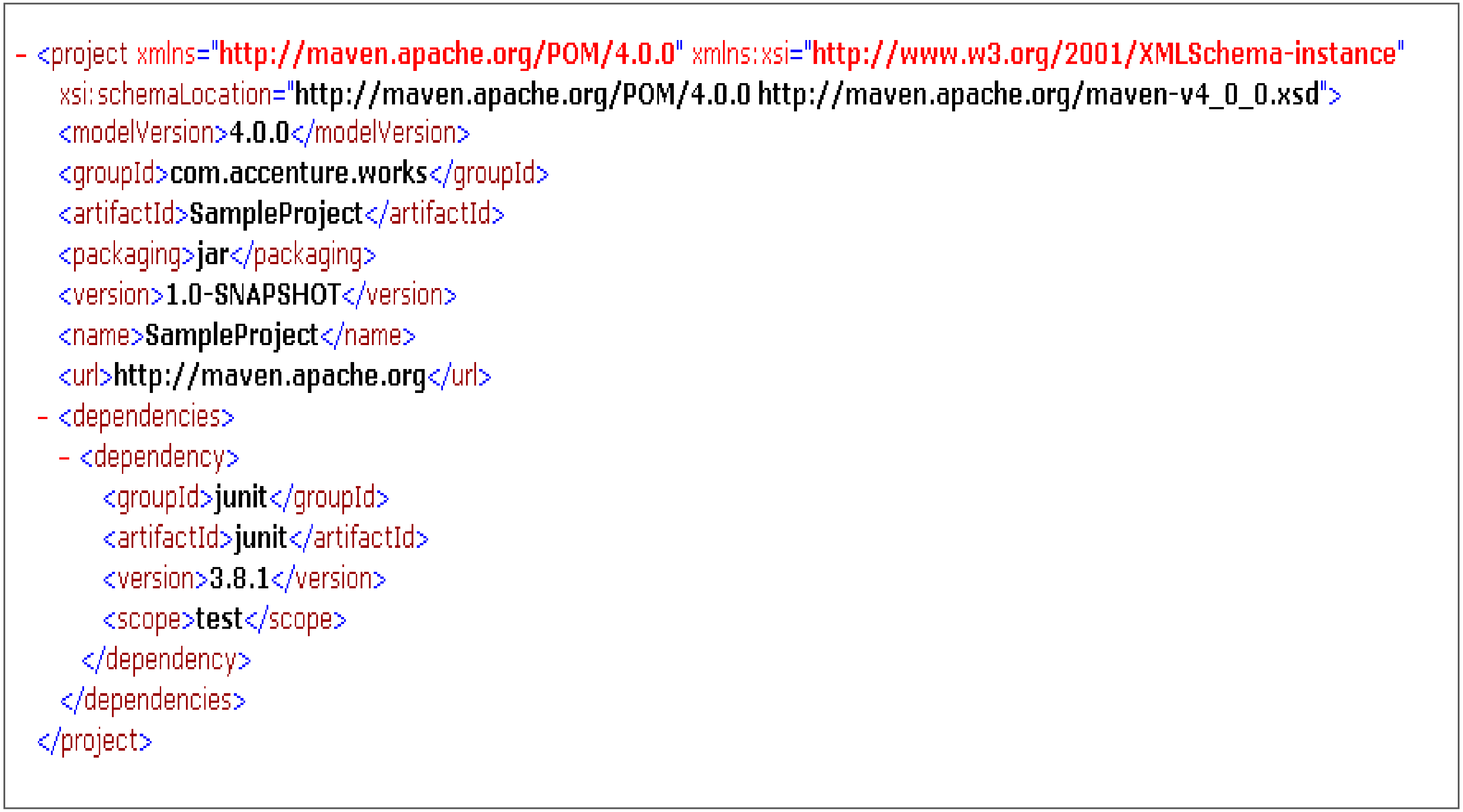
- Remove App.java and place the following SimpleExample.java :



```
package com.accenture.works;

public class SimpleExample {

    int acceptData(String data) {
        int contents = Integer.parseInt(data);
        contents = contents - 2;
        return contents;
    }

    public static void main(String[] args) {
        SimpleExample simpleExample = new SimpleExample();
        int output = simpleExample.acceptData("1001");
        System.out.println("Result: " + output);

    }

}
```

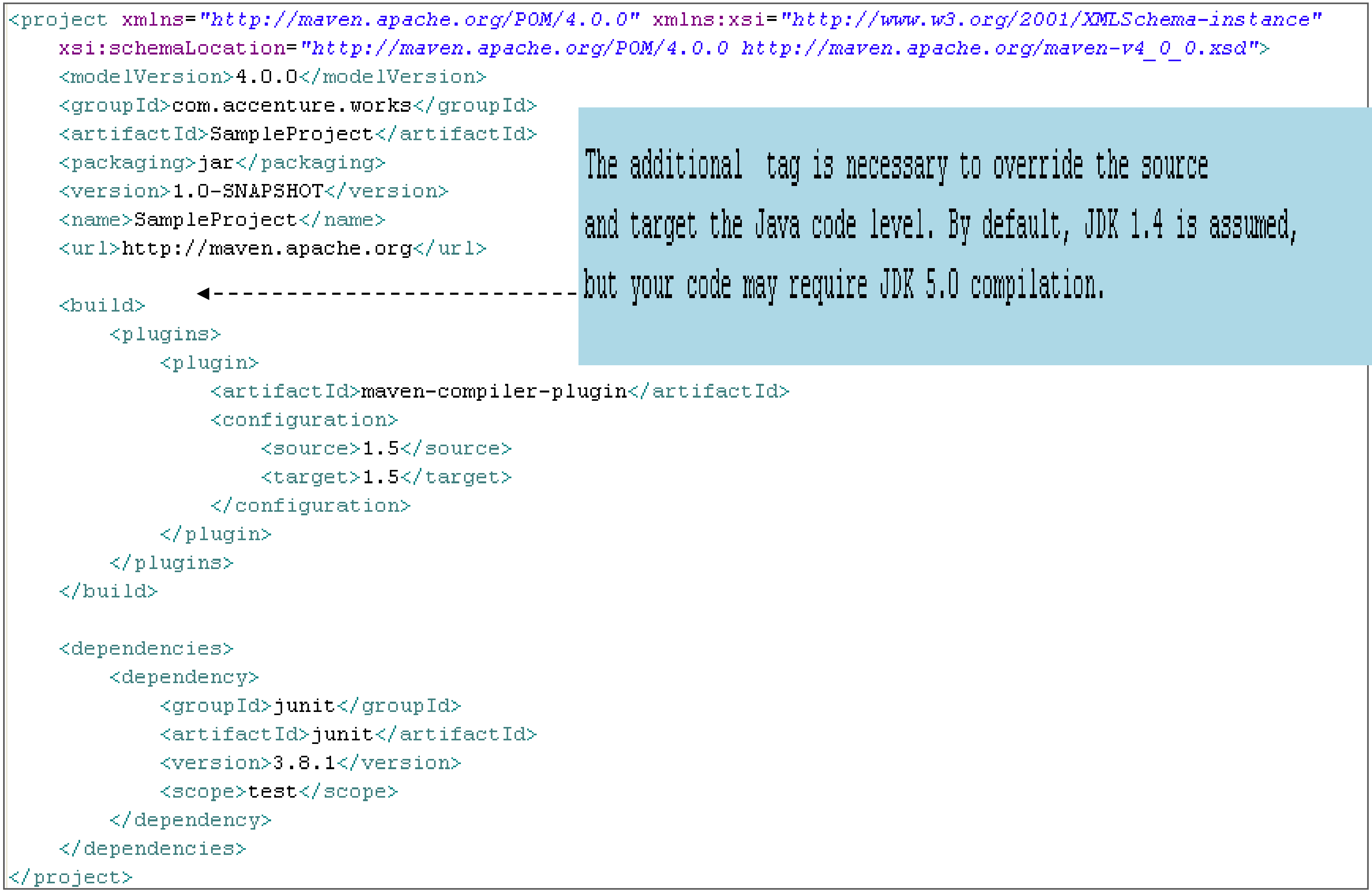
Sample pom.xml file

- The file generated by the Archetype plug-in for SampleProject is shown below:



```
- <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/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.accenture.works</groupId>
    <artifactId>SampleProject</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>SampleProject</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>
```

Customizing the generated pom.xml file

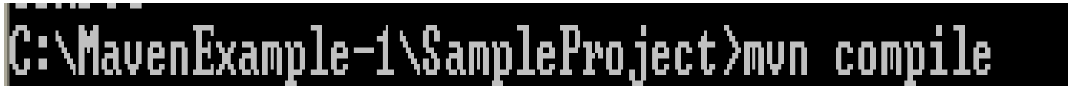


```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.accenture.works</groupId>
    <artifactId>SampleProject</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>SampleProject</name>
    <url>http://maven.apache.org</url>

    <build>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.5</source>
                    <target>1.5</target>
                </configuration>
            </plugin>
        </plugins>
    </build>

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

The additional tag is necessary to override the source and target the Java code level. By default, JDK 1.4 is assumed, but your code may require JDK 5.0 compilation.

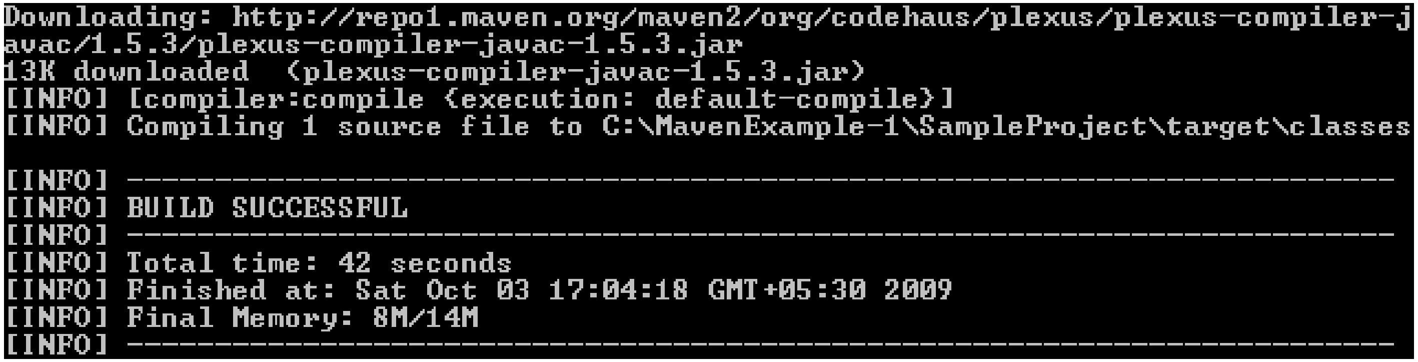
Compiling the customized Project

- You can now compile the SampleProject using mvn compile command.



```
C:\MavenExample-1\SampleProject>mvn compile
```

- You should see the report of a successful build, creating one class file in the target tree.
- This can take a little while if it is the first time you run it because some dependencies might need to be downloaded from the central repository over the Internet.



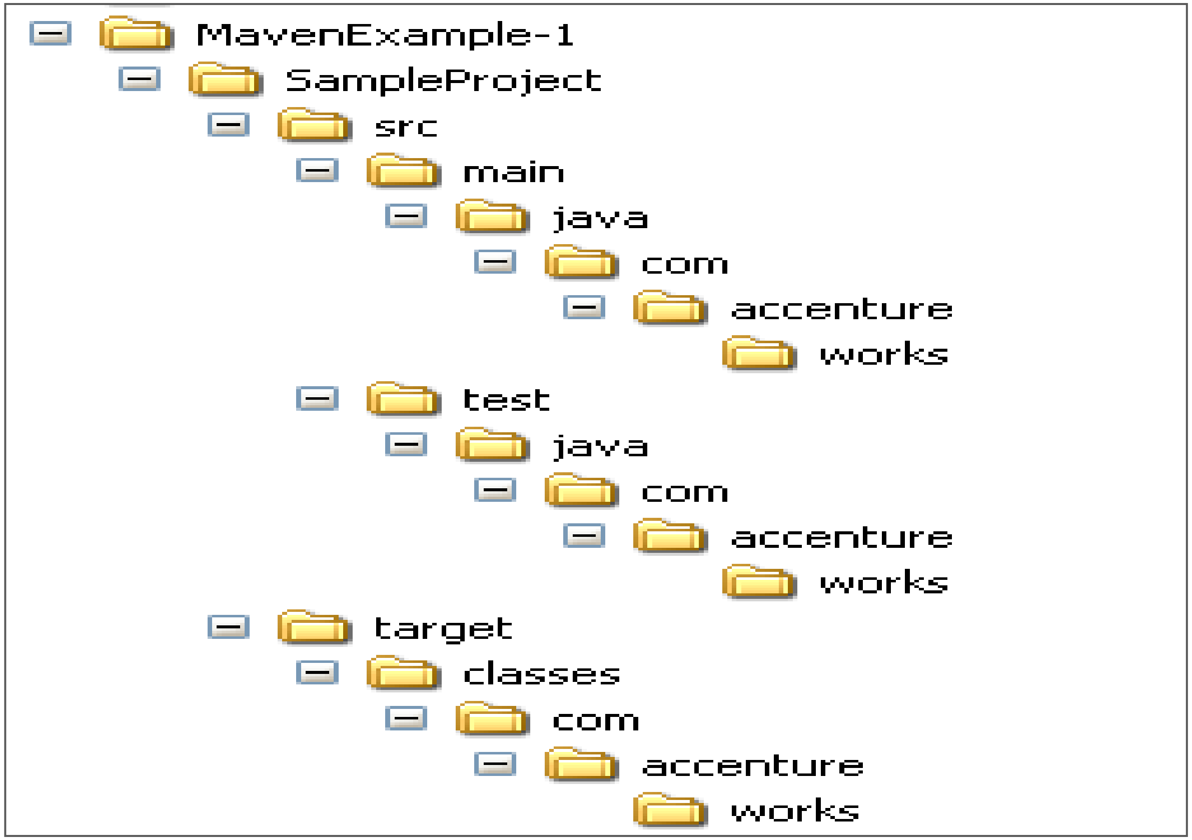
```
Downloading: http://repo1.maven.org/maven2/org/codehaus/plexus/plexus-compiler-j
avac/1.5.3/plexus-compiler-javac-1.5.3.jar
13K downloaded  (plexus-compiler-javac-1.5.3.jar)
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Compiling 1 source file to C:\MavenExample-1\SampleProject\target\classes

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 42 seconds
[INFO] Finished at: Sat Oct 03 17:04:18 GMT+05:30 2009
[INFO] Final Memory: 8M/14M
[INFO] ------------------------------------------------------------------------
```

Test jar

-
  ```
  java -cp target/my-app-1.0-SNAPSHOT.jar com.mycompany.app.App
  ```

Directory structure after compilation

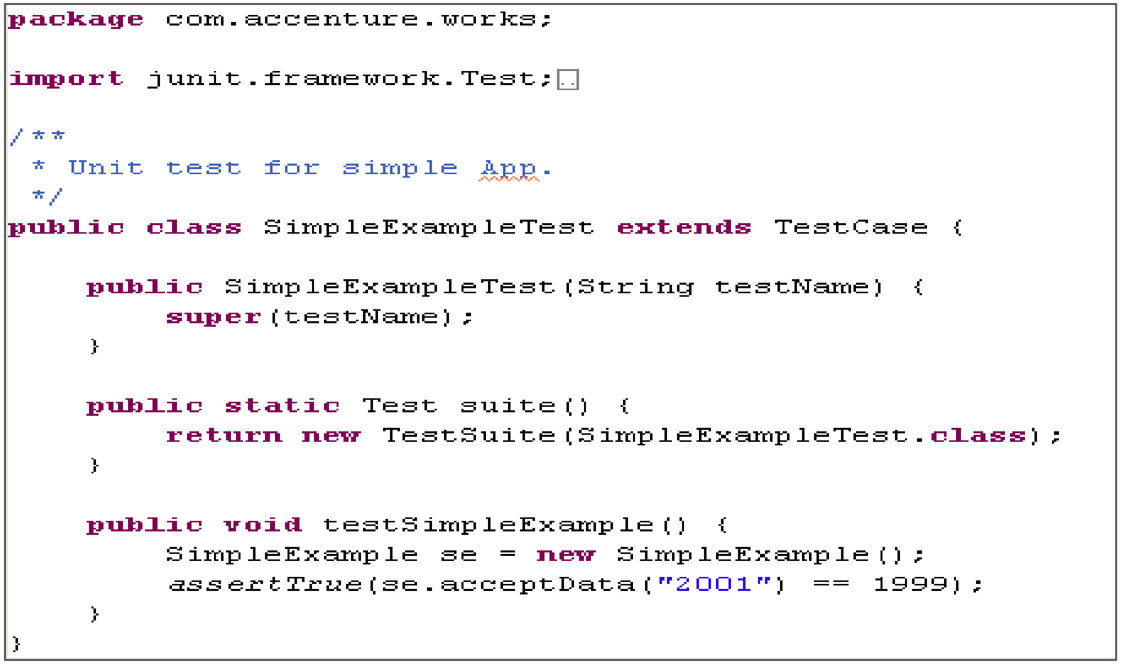
- After compilation, the .class files (SimpleExample.class) file is created in the target folder shown below:



```
MavenExample-1
  SampleProject
    src
      main
        java
          com
            accenture
              works
      test
        java
          com
            accenture
              works
    target
      classes
        com
          accenture
            works
```

Adding a Unit test

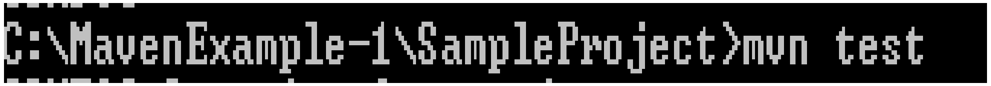
- It is a development best practice to provide unit tests for all code modules.
- Maven 2 created a placeholder AppTest.java unit test for you. Remove the file and place the following SimpleExampleTest.java :



```
package com.accenture.works;

import junit.framework.Test;□

/**
 * Unit test for simple App.
 */
public class SimpleExampleTest extends TestCase {

    public SimpleExampleTest(String testName) {
        super(testName);
    }

    public static Test suite() {
        return new TestSuite(SimpleExampleTest.class);
    }

    public void testSimpleExample() {
        SimpleExample se = new SimpleExample();
        assertTrue(se.acceptData("2001") == 1999);
    }
}
```

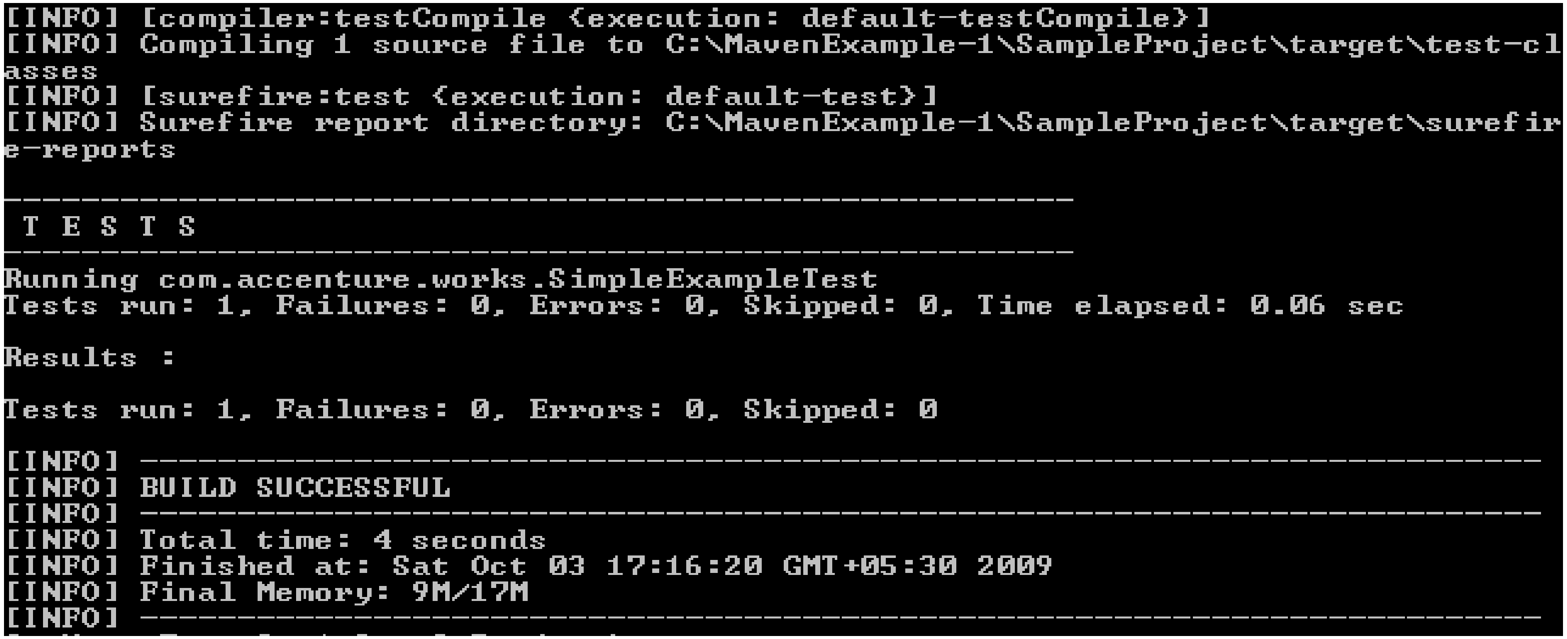
Running a unit test

- You can use mvn test command.



```
C:\MavenExample-1\SampleProject>mvn test
```

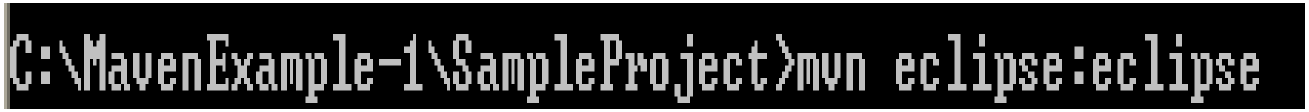
- Maven 2 compiles the source and the unit test. It then runs the tests, reporting on the number of successes, failures, and errors, as shown in the following figure:



```
[INFO] [compiler:testCompile {execution: default-testCompile}]
[INFO] Compiling 1 source file to C:\MavenExample-1\SampleProject\target\test-cl
asses
[INFO] [surefire:test {execution: default-test}]
[INFO] Surefire report directory: C:\MavenExample-1\SampleProject\target\surefir
e-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.accenture.works.SimpleExampleTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.06 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 4 seconds
[INFO] Finished at: Sat Oct 03 17:16:20 GMT+05:30 2009
[INFO] Final Memory: 9M/17M
[INFO] ------------------------------------------------------------------------
```

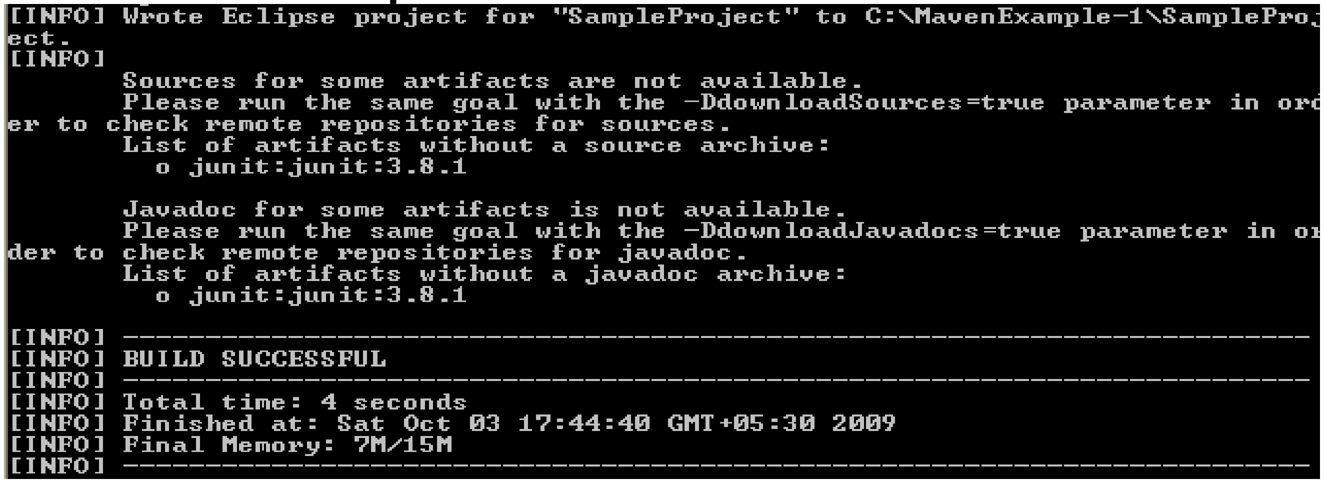
Creating eclipse specific files

- It is easier for developers to work with Eclipse, hence Maven supports creation of eclipse specific files
- To generate the eclipse metadata files from your **pom.xml** you execute the following command:



```
C:\MavenExample-1\SampleProject>mvn eclipse:eclipse
```

- The above command generates eclipse related files like .project and .classpath .



```
[INFO] Wrote Eclipse project for "SampleProject" to C:\MavenExample-1\SampleProj
ect.
[INFO]
       Sources for some artifacts are not available.
       Please run the same goal with the -DdownloadSources=true parameter in ord
er to check remote repositories for sources.
       List of artifacts without a source archive:
         o junit:junit:3.8.1

       Javadoc for some artifacts is not available.
       Please run the same goal with the -DdownloadJavadocs=true parameter in or
der to check remote repositories for javadoc.
       List of artifacts without a javadoc archive:
         o junit:junit:3.8.1

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 4 seconds
[INFO] Finished at: Sat Oct 03 17:44:40 GMT+05:30 2009
[INFO] Final Memory: 7M/15M
[INFO] ------------------------------------------------------------------------
```

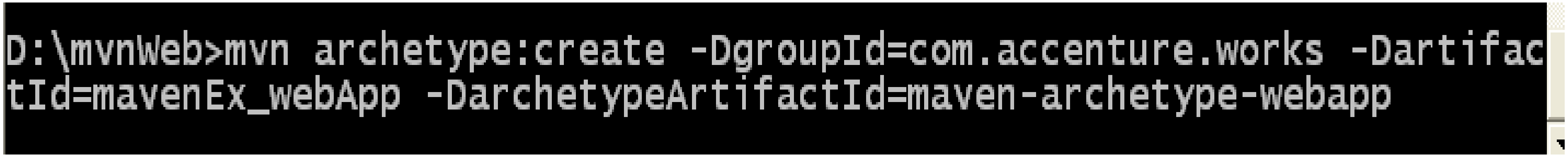
Importing the project in Eclipse

- After generating the Eclipse Classpath and Project files for your project, import the project into Eclipse with the following steps:

1. From the Eclipse workspace menu select File > Import
2. Expand General and select Existing Projects into Workspace
3. Mark Select root directory and hit Browse. Browse for the directory in which your POM file is in. Press OK.
4. You should now see your project checked under the Projects: box. Press Finish.
5. You should now see your project in Eclipse.

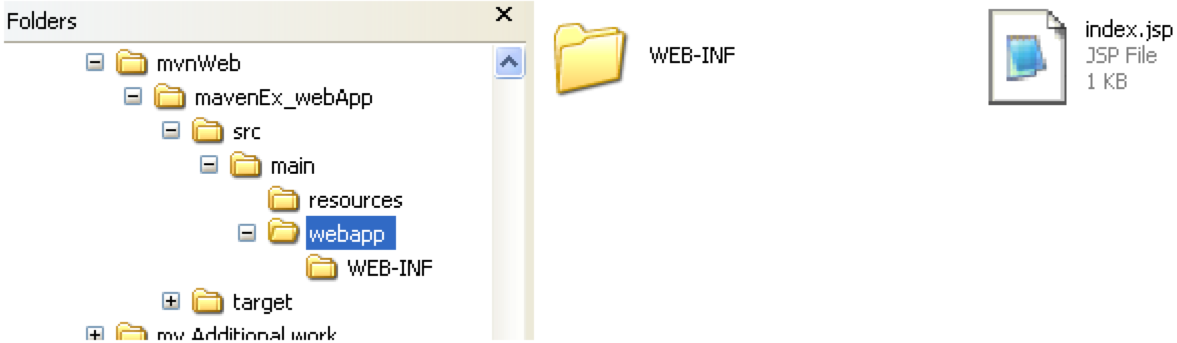
Creating a WAR file

- Now let’s see how to create a simple Web application:
- Execute the following command in the directory where you want the web application to be created.



```
D:\mvnWeb>mvn archetype:create -DgroupId=com.accenture.works -Dartifac
tId=mavenEx_webApp -DarchetypeArtifactId=maven-archetype-webapp
```

- The following directory structure will be created

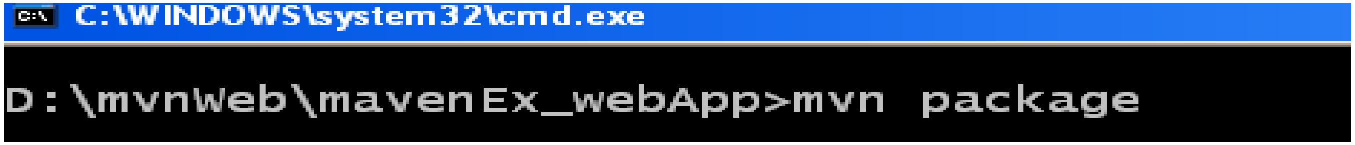


Web Project Create

- mvn archetype:generate -DarchetypeArtifactId=Maven-archetype-webapp -DgroupId=com.citi.fundtransfer -DartifactId=CITIMAVENFT

Creating a WAR file (Continued…)

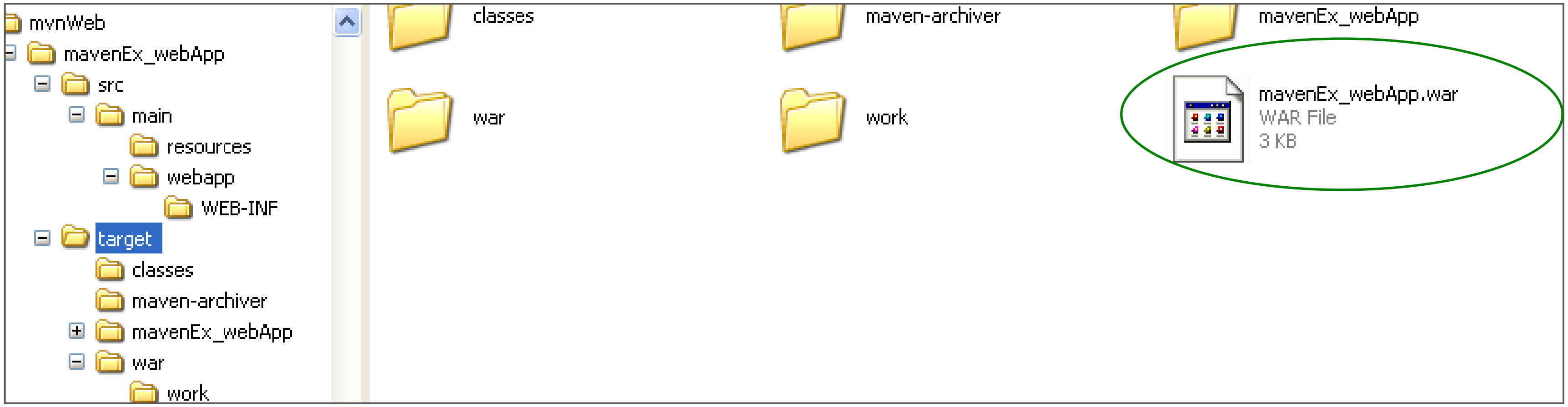
- The directory structure created will be as per WAR file.
- Maven provides us the required directory structure and template files.
- We have to create web pages and place them in the proper locations.
- For our application, lets use **index.jsp** file which is created by default.
- Add your contents to index.jsp and save it.
- Give following command in the command prompt.



```
D:\mvnWeb\mavenEx_webApp>mvn package
```

Creating a WAR file (Continued…)

- Notice a war file created in the target folder



- Deploy the same in a web server and request for **index.jsp** to see the output

Maven site

- Mvn site

Custom library

- mvn install:install-file -Dfile=c:\kaptcha-2.3..2.jar -DgroupId=com.google.code
- -DartifactId=kaptcha -Dversion=2.3 -Dpackaging=jar

- In POM.xml

- <dependency>
-     <groupId>com.google.code</groupId>
-     <artifactId>kaptcha</artifactId>
-     <version>2.3.2</version>
- </dependency>

Custom library

- mvn install:install-file -DgroupId=com.oracle -DartifactId=ojdbc6 -Dpackaging=jar -Dversion=11.2.0.4.0 -Dfile=D:\Oracle\app\oracle\product\11.2.0\server\jdbc\lib\ojdbc6.jar -DgeneratePom=true
- In POM.xml
- <!-- https://mvnrepository.com/artifact/com.oracle/ojdbc6 -->
- <dependency>
-     <groupId>com.oracle</groupId>
-     <artifactId>ojdbc6</artifactId>
-     <version>11.2.0.4.0</version>
-     <scope>runtime</scope>
- </dependency>

Local Repository

- {M2_HOME}\conf\setting.xml
- <settings>
-   <!-- localRepository
-    | The path to the local repository maven will use to store artifacts.
-    |
-    | Default: ~/.m2/repository
-   <localRepository>/path/to/local/repo</localRepository>
-   -->

- <localRepository>D:/maven_repo</localRepository>
- <interactiveMode>true</interactiveMode>
- <usePluginRegistry>false</usePluginRegistry>
- <offline>false</offline>

Maven: See It

- **Demonstration:**

  Faculty will demonstrate how to create a simple Java program using Maven and the Eclipse IDE

- **Time Allocated:** 10 minutes
- **Environment or File :** Eclipse
- **Steps:**
  1. Open Eclipse
  2. Create a new Maven project without using any archetype
  3. Expand the source tree and target tree and provide a walk through.
  4. Open pom.xml and provide a walkthrough
  5. Create a new package com.accenture.adfx.module1.sample
  6. Create a new Java file with a main method printing “Hello World” to the console
  7. Build and Run the file using Maven

Maven: Try - It

- **Now Try It:**

  Create a simple Java project using Maven

- **Time Allocated:** 15 minutes
- **Environment or File :** Eclipse
- **Steps:**
  1. Open Eclipse
  2. Create a new Maven project without using any archetype
  3. Expand and view the source tree and target tree
  4. Open and observe the generated pom.xml
  5. Create a new Java file in the com.accenture.adfx.module1.sample package with a main method printing “Hello World” to the console
  6. Build and Run the file using Maven

Maven: Activity

- **Objective:**

  Create a Maven project using maven-archetype-webapp and learn how to build and package the web application into a war
- **Time Allocated:** 30 min
- **Environment or File:** Eclipse
- **Instructions:**
  1. Open Eclipse
  2. Create a Maven using maven-archetype-webap archetype.
  3. Expand and observe the source tree and target tree.
  4. Open and observe the generated pom.xml
  5. Open index.jsp and modify the default message.
  6. Build the project using Maven.
  7. Package into a war file using Maven.
  8. Deploy and run on tomcat.

Questions

Module Summary

- In this module we discussed
  - Overview of Maven
  - Maven archetypes
  - Maven life cycle phases
  - The pom.xml file
  - Creation of Java projects using Maven
  - Creation of war files