Maven

https://www.youtube.com/watch?v=KNGQ9JBQWhQ
https://www.tutorialspoint.com/maven/index.htm
https://maven.apache.org/guides/ - For details

Agenda

- Maven: What & Why
- POM
  - POM Schema
  - Super/Parent/Effective POM
- Build Life Cycle
  - Phases
  - Goals
  - Plugins
  - Bind Goals to Phase
- Build Profiles
- Repository
- Dependency Management
- Maven Archetype to create maven project
- Sample Output for “mvn clean install”
- Common Maven Commands/Arguments
- Best Practices

Maven: What & Why

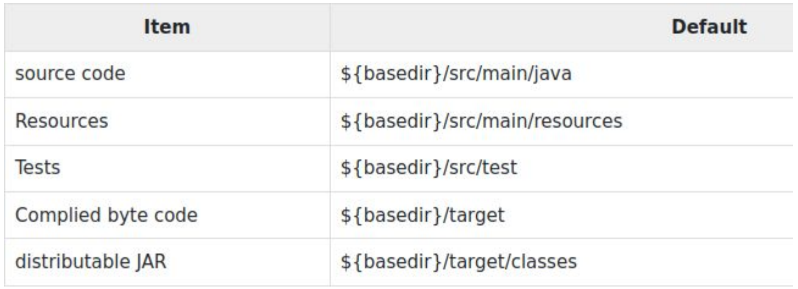
What: Maven is a build automation tool used primarily for Java projects. Maven can also be used to build and manage projects written in C#, Ruby, Scala, and other languages. The Maven project is hosted by the Apache Software Foundation. Maven provides developers ways to manage the following: Dependencies, Builds, Testing, Releases, Distribution, Documentation, Reporting. To summarize, Maven simplifies, standardizes & automate the project build process.

Why: Convention over Configuration: Different from ancestors like Ant.

Maven uses **Convention** over **Configuration**, which means developers are not required to create build process themselves. Developers do not have to mention each and every configuration detail. Maven provides sensible default behavior for projects. For example:

- When a Maven project is created, Maven creates default project structure. Developer is only required to place files accordingly and he/she need not to define any configuration in pom.xml. As an example, following table shows the default values for project source code files, resource files and other configurations. Assuming, **${basedir}** denotes the project location −

- Also maven provide pre-defined build life cycle and goals. Much of the project management and build related tasks are maintained by Maven plugins. Developers can build any given Maven project without the need to understand how the individual plugins work.



Item	Default
source code	${basedir}/src/main/java
Resources	${basedir}/src/main/resources
Tests	${basedir}/src/test
Complied byte code	${basedir}/target
distributable JAR	${basedir}/target/classes

POM

POM stands for Project Object Model. It is fundamental unit of work in Maven. It is an XML file that resides in the base directory of the project as pom.xml. The POM contains information about the project and various configuration detail used by Maven to build the project(s).

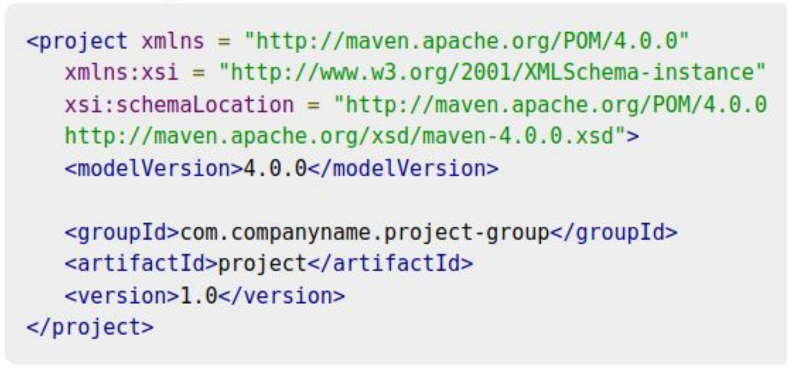
Some of the configuration that can be specified in the POM are following –

- project dependencies
- plugins
- goals
- build profiles
- Metadata including: project version, developers, mailing list

POM Identifier:

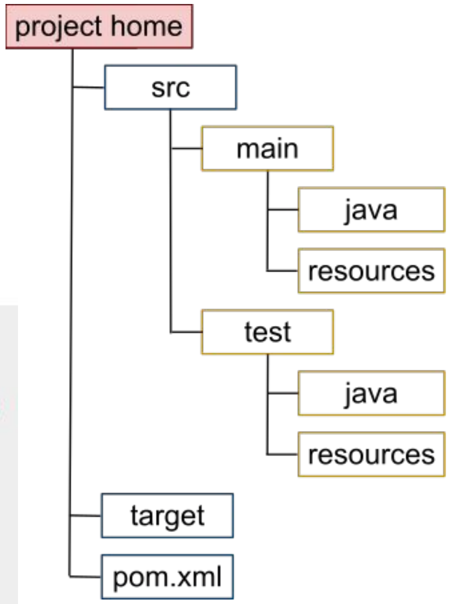
Before creating a POM, we should first decide the project **group** (groupId), its **name** (artifactId) and its **version** as these attributes help in uniquely identifying the project in repository.

POM Example



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

   <groupId>com.companyname.project-group</groupId>
   <artifactId>project</artifactId>
   <version>1.0</version>
</project>
```

Maven expected project structure

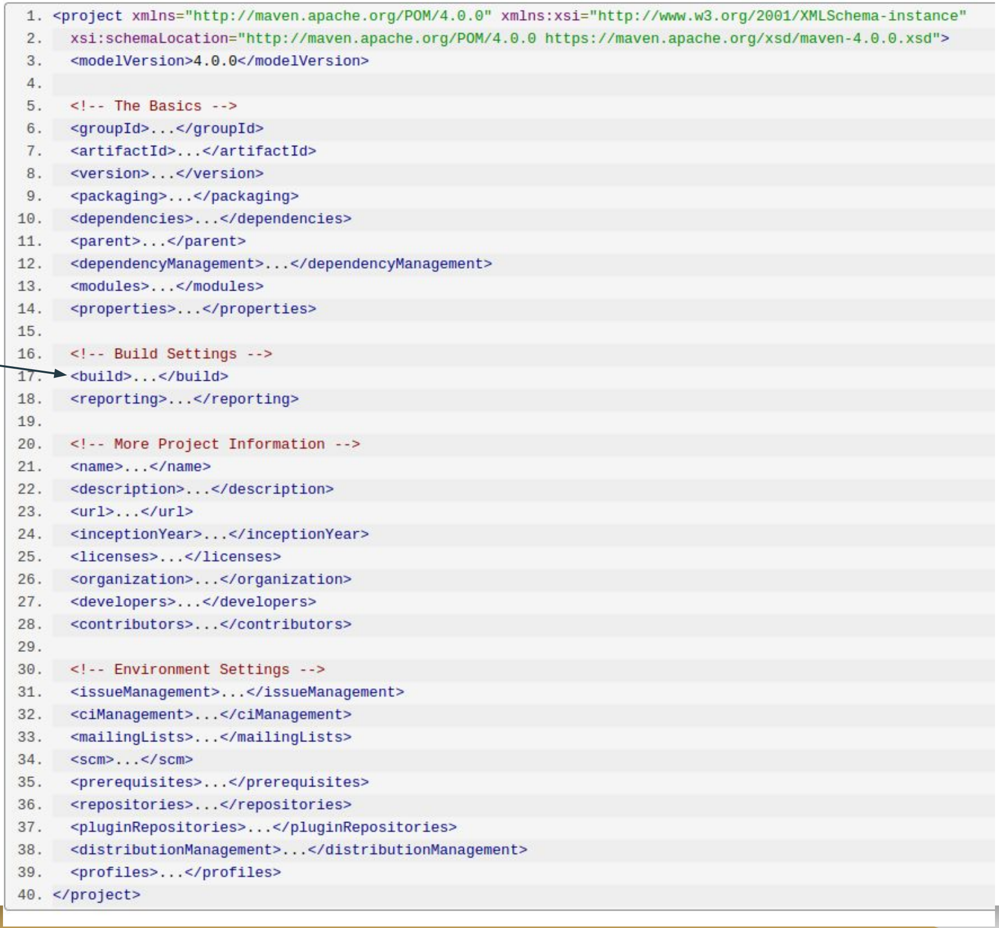


POM Top level Schema

https://maven.apache.org/pom.html

Further Contains

- Resource
- Plugins
- PluginManagement



```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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <!-- The Basics -->
  <groupId>...</groupId>
  <artifactId>...</artifactId>
  <version>...</version>
  <packaging>...</packaging>
  <dependencies>...</dependencies>
  <parent>...</parent>
  <dependencyManagement>...</dependencyManagement>
  <modules>...</modules>
  <properties>...</properties>

  <!-- Build Settings -->
  <build>...</build>
  <reporting>...</reporting>

  <!-- More Project Information -->
  <name>...</name>
  <description>...</description>
  <url>...</url>
  <inceptionYear>...</inceptionYear>
  <licenses>...</licenses>
  <organization>...</organization>
  <developers>...</developers>
  <contributors>...</contributors>

  <!-- Environment Settings -->
  <issueManagement>...</issueManagement>
  <ciManagement>...</ciManagement>
  <mailingLists>...</mailingLists>
  <scm>...</scm>
  <prerequisites>...</prerequisites>
  <repositories>...</repositories>
  <pluginRepositories>...</pluginRepositories>
  <distributionManagement>...</distributionManagement>
  <profiles>...</profiles>
</project>
```

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.

Maven use the **effective POM** (configuration from super pom plus project configuration) to execute relevant goal. It helps developers to specify minimum configuration detail in his/her pom.xml. Although configurations can be overridden easily.

An easy way to look at the default configurations of the super POM is by running the following command: **mvn help:effective-pom**

```
<repositories>
   <repository>
      <snapshots>
         <enabled>false</enabled>
      </snapshots>
      <id>central</id>
      <name>Central Repository</name>
      <url>https://repo.maven.apache.org/maven2</url>
   </repository>
</repositories>
<pluginRepositories>
   <pluginRepository>
      <releases>
         <updatePolicy>never</updatePolicy>
      </releases>
      <snapshots>
         <enabled>false</enabled>
      </snapshots>
      <id>central</id>
      <name>Central Repository</name>
      <url>https://repo.maven.apache.org/maven2</url>
   </pluginRepository>
```

```
<build>
   <sourceDirectory>C:\MVN\src\main\java</sourceDirectory>
   <scriptSourceDirectory>C:\MVN\src\main\scripts</scriptSourceDirectory>
   <testSourceDirectory>C:\MVN\src\test\java</testSourceDirectory>
   <outputDirectory>C:\MVN\target\classes</outputDirectory>
   <testOutputDirectory>C:\MVN\target\test-classes</testOutputDirectory>
   <resources>
      <resource>
         <directory>C:\MVN\src\main\resources</directory>
      </resource>
   </resources>
   <testResources>
      <testResource>
         <directory>C:\MVN\src\test\resources</directory>
      </testResource>
   </testResources>
   <directory>C:\MVN\target</directory>
```

```
<pluginManagement>
   <plugins>
      <plugin>
         <artifactId>maven-antrun-plugin</artifactId>
         <version>1.3</version>
      </plugin>
      <plugin>
         <artifactId>maven-assembly-plugin</artifactId>
         <version>2.2-beta-5</version>
      </plugin>
      <plugin>
         <artifactId>maven-dependency-plugin</artifactId>
         <version>2.8</version>
      </plugin>
      <plugin>
         <artifactId>maven-release-plugin</artifactId>
         <version>2.5.3</version>
      </plugin>
   </plugins>
</pluginManagement>
<plugins>
   <plugin>
      <artifactId>maven-clean-plugin</artifactId>
      <version>2.5</version>
```

```
<reporting>
   <outputDirectory>C:\MVN\target\site</outputDirectory>
</reporting>
```

In above pom.xml, you can see the inherited project source folders structure, output directory, plug-ins required, repositories, reporting directory, which Maven will be using while executing the desired goals.

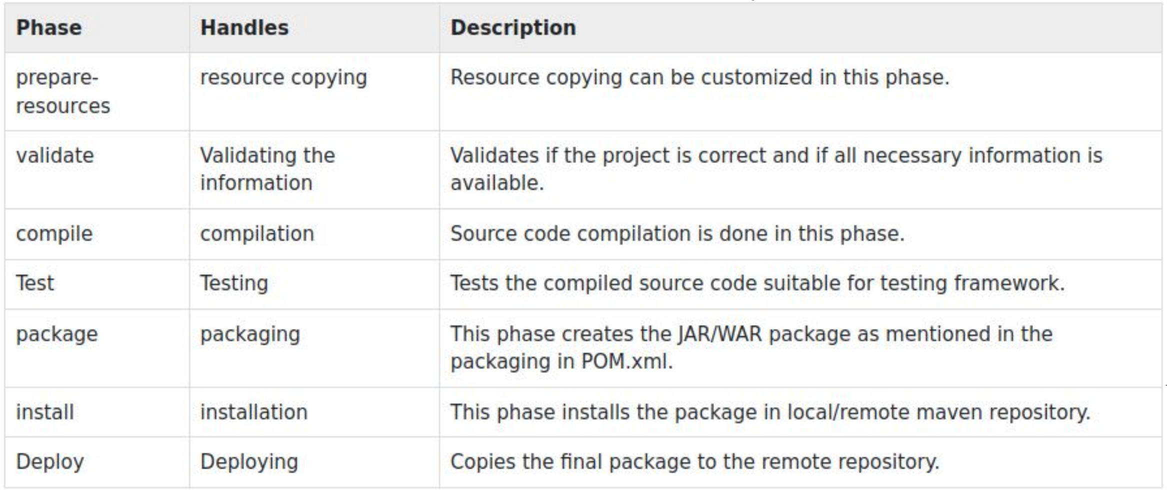
For Phases Details: https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html
For maven site - https://jstobigdata.com/maven/maven-site-project-documentation/

Build Life Cycle: Phases

A Build Lifecycle is a well-defined sequence of phases, which define the order in which the goals are to be executed. Here phase represents a stage in life cycle. Maven has the following three standard lifecycles –

- Clean: Deletes target folder (build directory) and other created artifacts during build
- default(or build): This is the primary life cycle of Maven and is used to build the application. It has the following 21 phases.
- Site: Maven Site plugin is generally used to create fresh documentation to create reports, deploy site (will be accessible at port 8080), etc.

Summarized version of build phases



Phase	Handles	Description
prepare-resources	resource copying	Resource copying can be customized in this phase.
validate	Validating the information	Validates if the project is correct and if all necessary information is available.
compile	compilation	Source code compilation is done in this phase.
Test	Testing	Tests the compiled source code suitable for testing framework.
package	packaging	This phase creates the JAR/WAR package as mentioned in the packaging in POM.xml.
install	installation	This phase installs the package in local/remote maven repository.
Deploy	Deploying	Copies the final package to the remote repository.

verify - run any checks on results of integration tests to ensure quality criteria are met like checkstyle

(Runs after package, before Install)

For Details: https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

Build Life Cycle: Goals

Each phase is a sequence of goals, and each goal is responsible for a specific task.

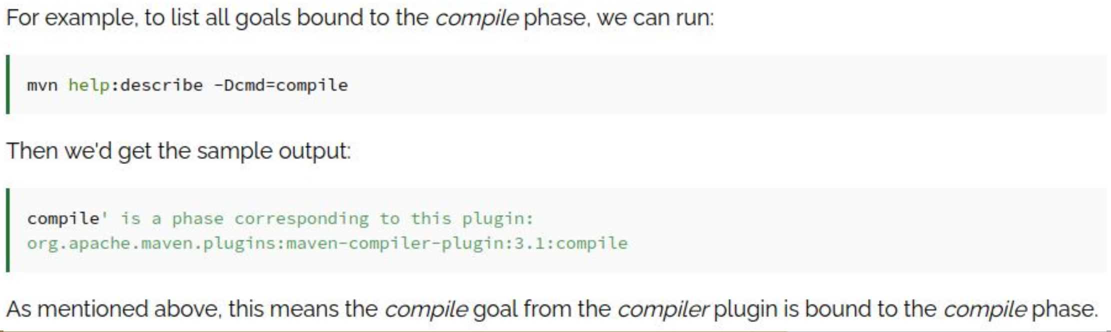
Even though a build phase is responsible for a specific step in the build lifecycle, the manner in which it carries out those responsibilities may vary. And this is done by declaring the plugin goals bound to those build phases. A goal may be bound to zero or more build phases.

Here are some of the phases and default goals bound to them:

- ***compiler**:compile* – the *compile* goal from the *compiler* plugin is bound to the *compile* phase
- ***compiler**:testCompile* is bound to the *test-compile* phase
- ***surefire**:test* is bound to the *test* phase
- ***install**:install* is bound to the *install* phase
- ***jar**:jar* and ***war**:war* is bound to the *package* phase

We can list all goals bound to a specific phase and their plugins using the command:

mvn help:describe -Dcmd=PHASENAME



For example, to list all goals bound to the *compile* phase, we can run:

```
mvn help:describe -Dcmd=compile
```

Then we'd get the sample output:

```
compile' is a phase corresponding to this plugin:
org.apache.maven.plugins:maven-compiler-plugin:3.1:compile
```

As mentioned above, this means the *compile* goal from the *compiler* plugin is bound to the *compile* phase.

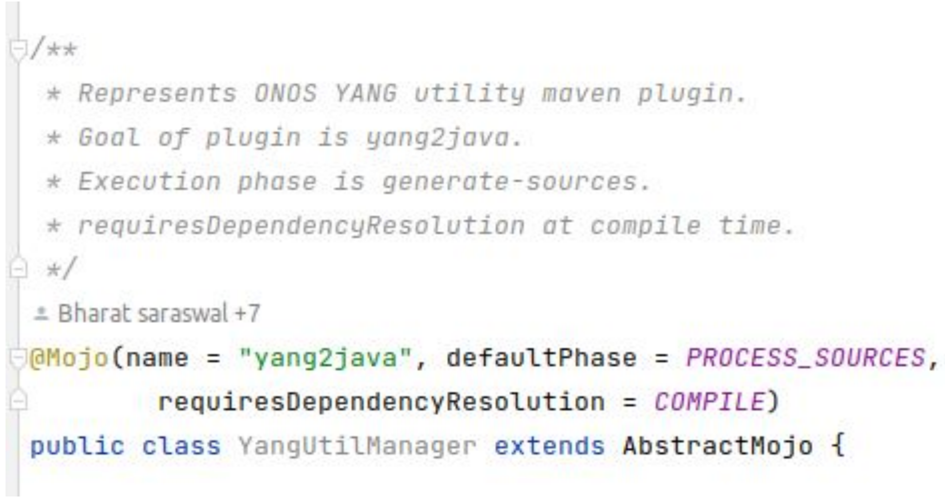
Build Life Cycle: Plugins

https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html
Commonly Used Plugins: https://maven.apache.org/plugins/index.html

Maven is actually **a plugin execution framework where every task is actually done by plugins**. **A Maven plugin is an implementation of group of goals**.

There are predefined set of plugins also users can define their plugins

ONOS YANG Tools - Yang2Java plugin



```
/**
 * Represents ONOS YANG utility maven plugin.
 * Goal of plugin is yang2java.
 * Execution phase is generate-sources.
 * requiresDependencyResolution at compile time.
 */
Bharat saraswal +7
@Mojo(name = "yang2java", defaultPhase = PROCESS_SOURCES,
        requiresDependencyResolution = COMPILE)
public class YangUtilManager extends AbstractMojo {
```

Plugin Usage

```
<build>
  <plugins>
    <plugin>
      <groupId>org.onosproject</groupId>
      <artifactId>onos-yang-maven-plugin</artifactId>
      <version>1.9</version>
      <executions>
        <execution>
         <configuration>
             <classFileDir>src/main/java</classFileDir>
          </configuration>
          <goals>
            <goal>yang2java</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Furthermore, a plugin may have one or more goals wherein each goal represents a capability of that plugin. For example, the Compiler plugin has two goals: compile and testCompile. The former compiles the source code of your main code, while the latter compiles the source code of your test code.

Build Life Cycle: Bind Goal to Phases

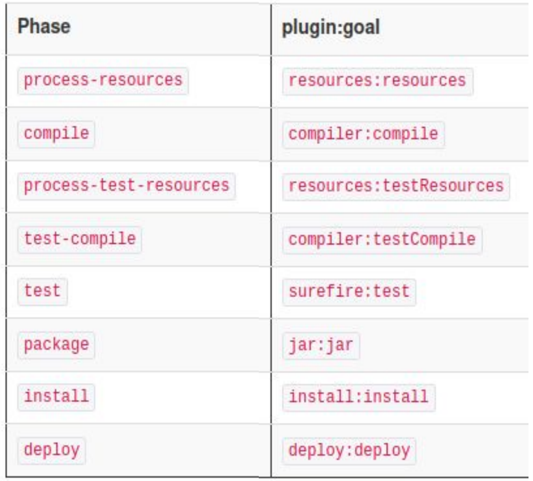
https://www.baeldung.com/maven-goals-phases

Build lifecycle is simple enough to use, but when you are constructing a Maven build for a project, how to assign tasks/goals to each of those build phases?

1. Packaging
2. Plugins

The first, and most common way, is to set the packaging for your project via the equally named POM element <packaging>. Some of the valid packaging values are jar, war, ear and pom. If no packaging value has been specified, it will default to jar.

Each packaging contains a list of goals to bind to a particular phase. For example, the jar packaging will bind the following goals to build phases of the default lifecycle.



Phase	plugin: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`

This is an almost standard set of bindings; however, some packagings handle them differently. For example, a project that is purely metadata (packaging value is pom) only binds goals to the install and deploy phases (for a complete list of goal-to-build-phase bindings of some of the packaging types, refer to the Lifecycle Reference).

Note that for some packaging types to be available, you may also need to include a particular plugin in the <build> section of your POM and specify <extensions>true</extensions> for that plugin. One example of a plugin that requires this is the Plexus plugin, which provides a plexus-application and plexus-service packaging.

Build Life Cycle: Bind Goal to Phases

Plugins

The second way to add goals to phases is to configure plugins in your project. Plugins are artifacts that provide goals to Maven.

While defining plugin phase is specified (can refer Yang2Java plugin for example). But some goals can be used in more than one phase, and there may not be a sensible default. For those, you can specify the phase yourself. For example, let's say you have a goal display:time that echos the current time to the command line, and you want it to run in the process-test-resources phase to indicate when the tests were started. This would be configured like so:

```
<plugin>
  <groupId>com.mycompany.example</groupId>
  <artifactId>display-maven-plugin</artifactId>
  <version>1.0</version>
  <executions>
    <execution>
      <phase>process-test-resources</phase>
      <goals>
        <goal>time</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Purpose of execution: So that you can run the same goal multiple times with different configuration if needed.

Purpose of explicitly specifying goal: A plugin may have one or more goals wherein each goal represents a capability of that plugin. For example, the Compiler plugin has two goals: compile and testCompile. The former compiles the source code of your main code, while the latter compiles the source code of your test code

Build Life Cycle: Execution Order

1. **The order of execution depends on the order in which the goal(s) and the build phase(s) are invoked.**

For example:

`mvn clean dependency:copy-dependencies package`   (mvn [plugin-name]:[goal-name] - To execute goal)

The **clean** and **package** arguments are build phases while the **dependency:copy-dependencies** is a goal. Here the *clean* phase will be executed first, followed by the **dependency:copy-dependencies goal**, and finally *package* phase will be executed.

2. When a phase is called via Maven command, for example **mvn compile**, only phases up to and including that phase will execute.

So in previous example “clean” will execute “pre-clean” and “clean” phase, but will not execute “post-clean” phase

Note: Goals which will be executed depends on 1. Packaging type 2. Plugins used (as mentioned in previous slides)

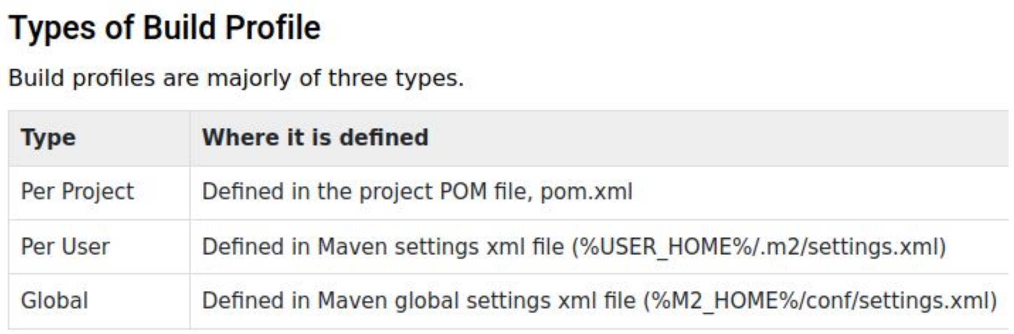
Build Profiles

https://maven.apache.org/guides/introduction/introduction-to-profiles.html
https://www.tutorialspoint.com/maven/maven_build_profiles.htm

Why: Apache Maven goes to great lengths to ensure that builds are portable. Among other things, this means allowing build configuration inside the POM, avoiding **all** filesystem references (in inheritance, dependencies, and other places), and leaning much more heavily on the local repository to store the metadata needed to make this possible.

However, sometimes portability is not entirely possible. Under certain conditions, plugins may need to be configured with **local filesystem paths**. Under other circumstances, a slightly **different dependency set** will be required, and the **project's artifact name may need to be adjusted slightly**. And at still other times, you may even need to include a whole plugin in the build lifecycle depending on the detected build environment.

To address these circumstances, Maven supports build profiles.



Types of Build Profile

Build profiles are majorly of three types.

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)

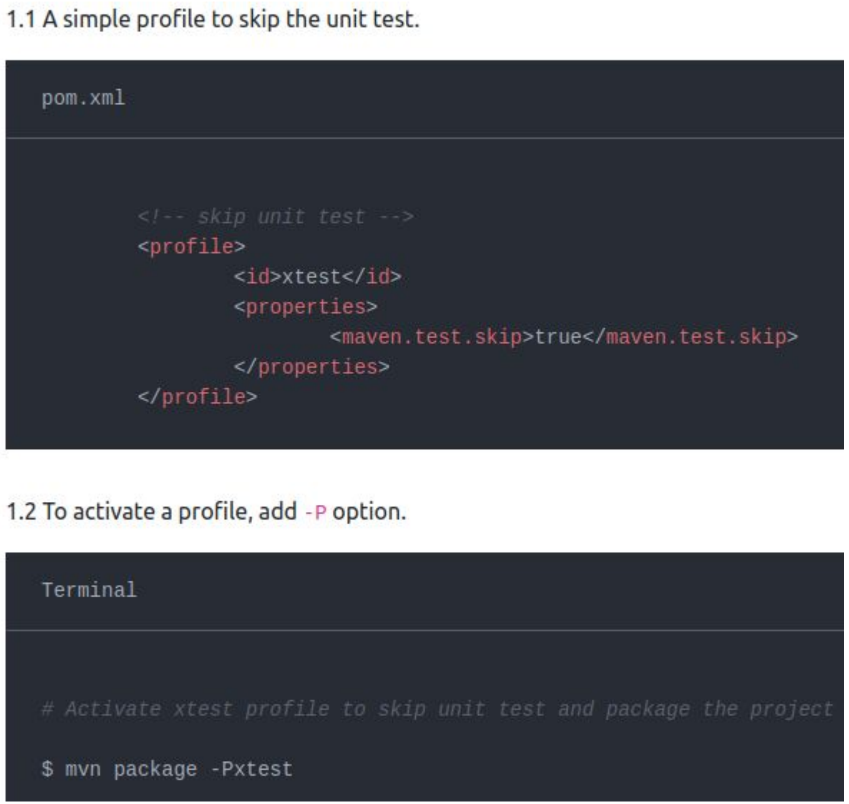
A profile can be activated in several ways:

- From the command line
- Through Maven settings
- Based on environment variables
- OS settings
- Present or missing files

https://mkyong.com/maven/maven-profiles-example/

Build Profiles: Examples

Example 1: Define custom profile to skip test and activate it using command line



1.1 A simple profile to skip the unit test.

pom.xml

```xml
	<!-- skip unit test -->
	<profile>
		<id>xtest</id>
		<properties>
			<maven.test.skip>true</maven.test.skip>
		</properties>
	</profile>
```

1.2 To activate a profile, add `-P` option.

Terminal

```
# Activate xtest profile to skip unit test and package the project

$ mvn package -Pxtest
```

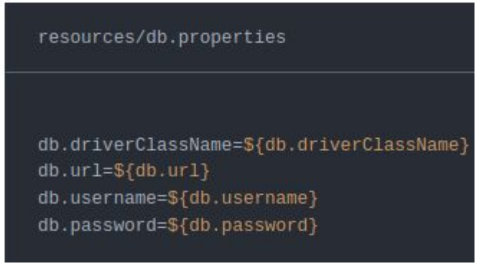
Other examples could include defining different repositories for dev & production environment etc. refer onap/onos settings.xml for details

Build Profiles: Examples

https://mkyong.com/maven/maven-profiles-example/

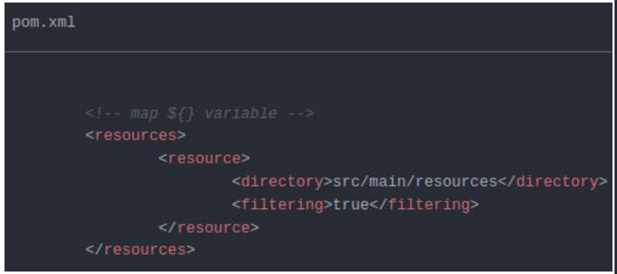
Define different config properties for dev and prod

2.1 A properties file.



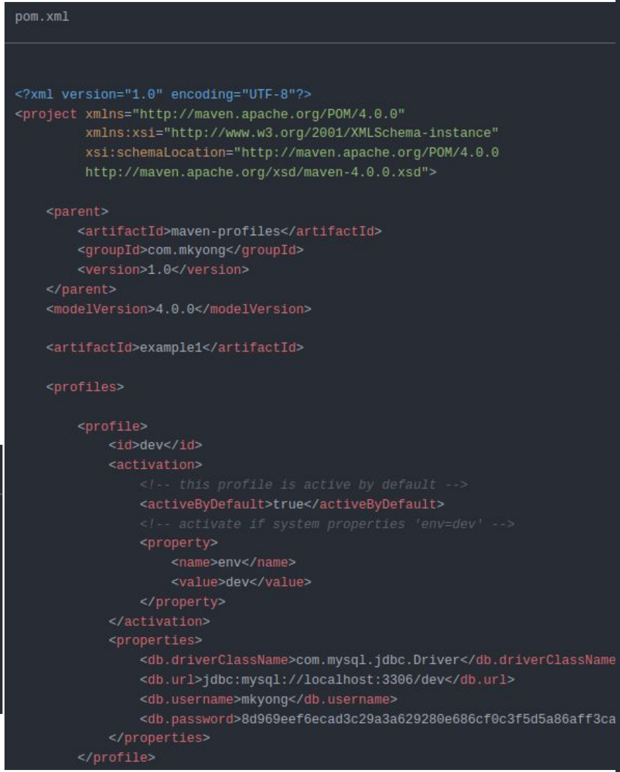
```
resources/db.properties

db.driverClassName=${db.driverClassName}
db.url=${db.url}
db.username=${db.username}
db.password=${db.password}
```

2.2 Enable the filtering. Maven will map the ${} in resources/db.properties with the active Maven profile properties.

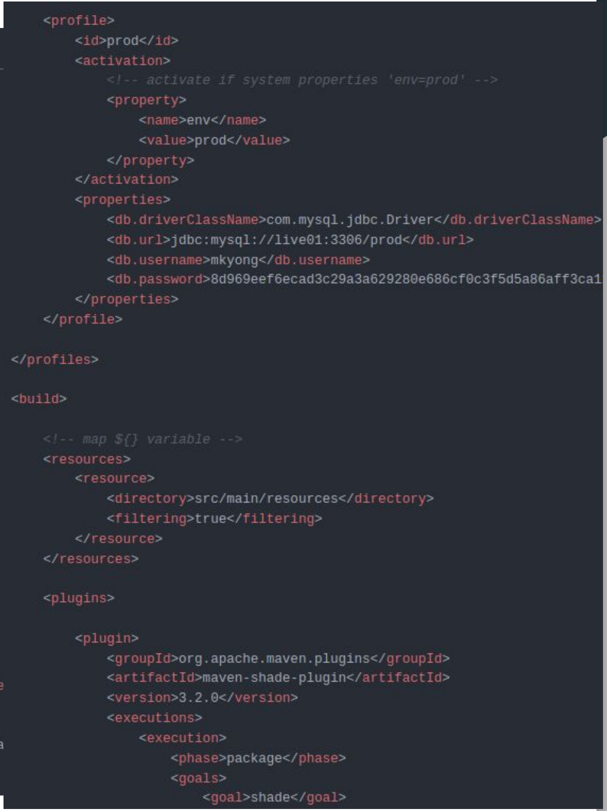


```
pom.xml

    <!-- map ${} variable -->
    <resources>
        <resource>
            <directory>src/main/resources</directory>
            <filtering>true</filtering>
        </resource>
    </resources>
```

2.3 Create two profiles ids (dev and prod) with different properties values.



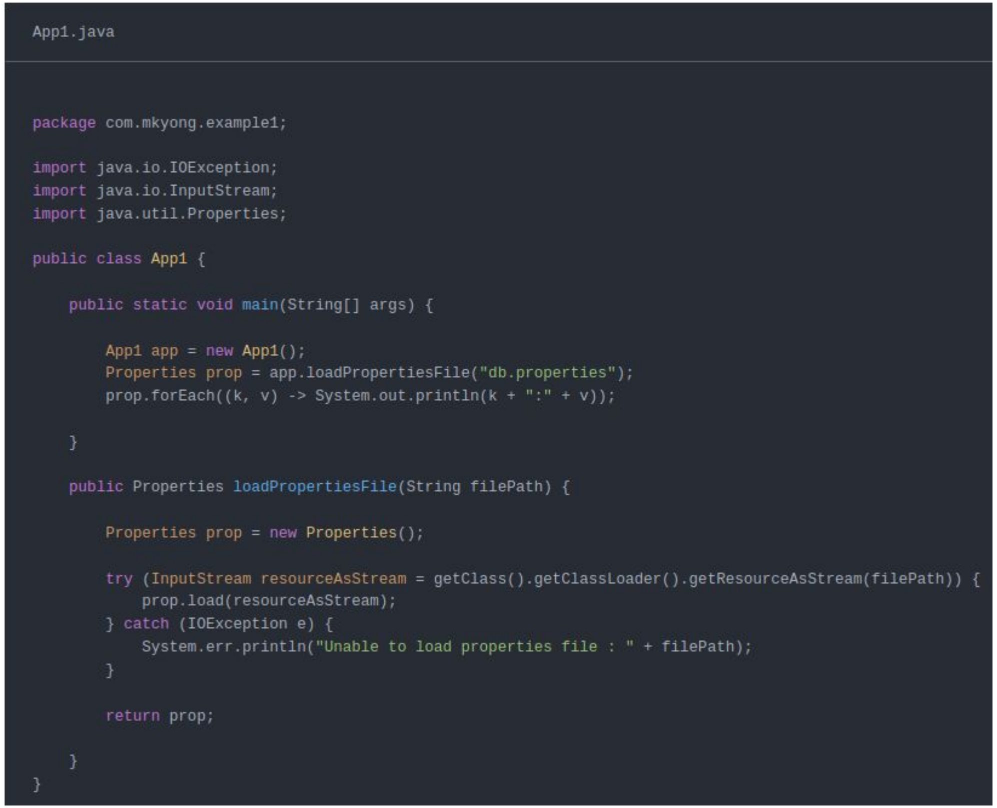
```
pom.xml

<?xml version="1.0" encoding="UTF-8"?>
<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">

    <parent>
        <artifactId>maven-profiles</artifactId>
        <groupId>com.mkyong</groupId>
        <version>1.0</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>example1</artifactId>

    <profiles>

        <profile>
            <id>dev</id>
            <activation>
                <!-- this profile is active by default -->
                <activeByDefault>true</activeByDefault>
                <!-- activate if system properties 'env=dev' -->
                <property>
                    <name>env</name>
                    <value>dev</value>
                </property>
            </activation>
            <properties>
                <db.driverClassName>com.mysql.jdbc.Driver</db.driverClassName
                <db.url>jdbc:mysql://localhost:3306/dev</db.url>
                <db.username>mkyong</db.username>
                <db.password>8d969eef6ecad3c29a3a629280e686cf0c3f5d5a86aff3ca
            </properties>
        </profile>
```



```
        <profile>
            <id>prod</id>
            <activation>
                <!-- activate if system properties 'env=prod' -->
                <property>
                    <name>env</name>
                    <value>prod</value>
                </property>
            </activation>
            <properties>
                <db.driverClassName>com.mysql.jdbc.Driver</db.driverClassName>
                <db.url>jdbc:mysql://live01:3306/prod</db.url>
                <db.username>mkyong</db.username>
                <db.password>8d969eef6ecad3c29a3a629280e686cf0c3f5d5a86aff3ca1
            </properties>
        </profile>

    </profiles>

    <build>

        <!-- map ${} variable -->
        <resources>
            <resource>
                <directory>src/main/resources</directory>
                <filtering>true</filtering>
            </resource>
        </resources>

        <plugins>

            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>3.2.0</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
```

Build Profiles: Examples

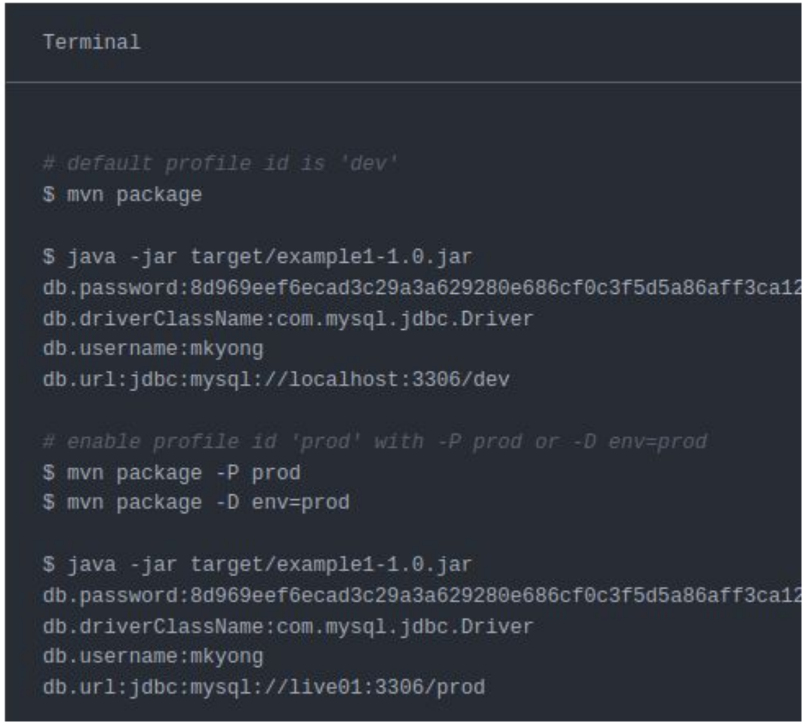
2.4 Loads the properties file and print it out.



App1.java

```java
package com.mkyong.example1;

import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;

public class App1 {

    public static void main(String[] args) {

        App1 app = new App1();
        Properties prop = app.loadPropertiesFile("db.properties");
        prop.forEach((k, v) -> System.out.println(k + ":" + v));

    }

    public Properties loadPropertiesFile(String filePath) {

        Properties prop = new Properties();

        try (InputStream resourceAsStream = getClass().getClassLoader().getResourceAsStream(filePath)) {
            prop.load(resourceAsStream);
        } catch (IOException e) {
            System.err.println("Unable to load properties file : " + filePath);
        }

        return prop;

    }
}
```

2.5 Test it.



Terminal

```
# default profile id is 'dev'
$ mvn package

$ java -jar target/example1-1.0.jar
db.password:8d969eef6ecad3c29a3a629280e686cf0c3f5d5a86aff3ca12
db.driverClassName:com.mysql.jdbc.Driver
db.username:mkyong
db.url:jdbc:mysql://localhost:3306/dev

# enable profile id 'prod' with -P prod or -D env=prod
$ mvn package -P prod
$ mvn package -D env=prod

$ java -jar target/example1-1.0.jar
db.password:8d969eef6ecad3c29a3a629280e686cf0c3f5d5a86aff3ca12
db.driverClassName:com.mysql.jdbc.Driver
db.username:mkyong
db.url:jdbc:mysql://live01:3306/prod
```

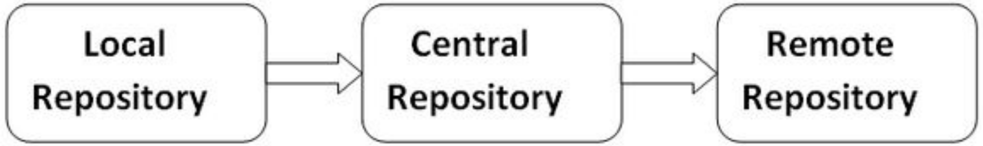
Repositories

https://maven.apache.org/guides/introduction/introduction-to-repositories.html

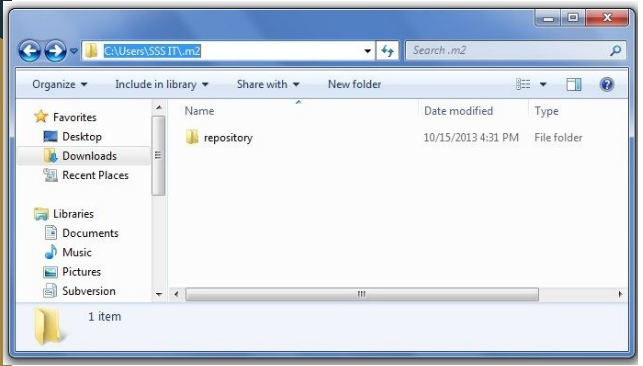
In Maven terminology, a repository is a directory where all the project jars, library jar, plugins or any other project specific artifacts are stored and can be used by Maven easily.

3 type: local, central, remote

Repository Search Order



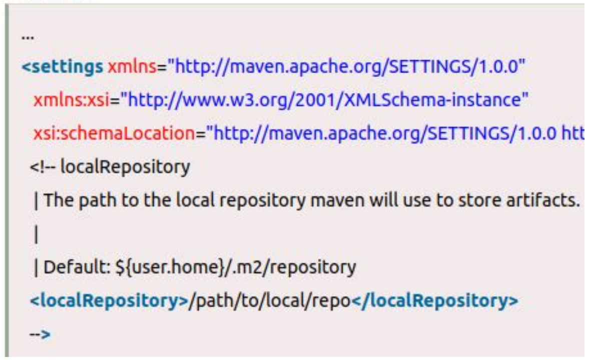
Local Repository: /home/root1/.m2



Name	Date modified	Type
repository	10/15/2013 4:31 PM	File folder

We can change the location of maven local repository by changing the **settings.xml** file. It is located in **MAVEN_HOME/conf/settings.xml**

Let's see the default code of settings.xml file.



settings.xml

```
...
<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 htt
 <!-- localRepository
 | The path to the local repository maven will use to store artifacts.
 |
 | Default: ${user.home}/.m2/repository
 <localRepository>/path/to/local/repo</localRepository>
 -->
```

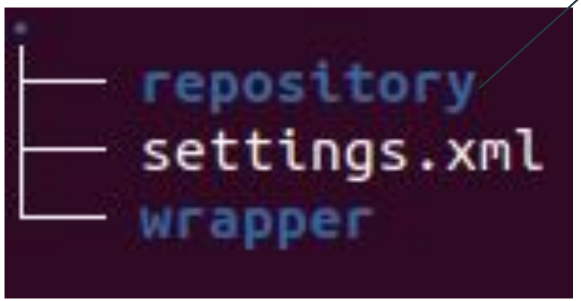
Maven Central Repository: https://mvnrepository.com/repos/central

By default, Maven will download from the central repository.

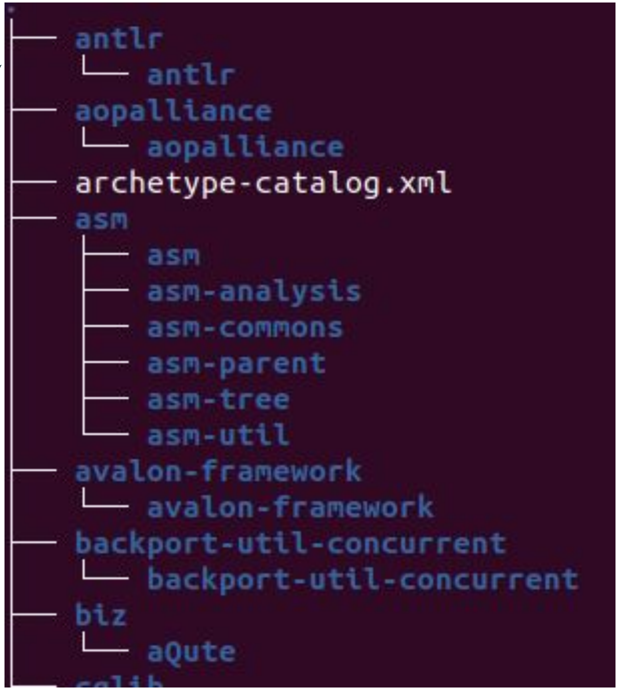
To override this, you need to specify a mirror as shown in Using Mirrors for Repositories.

You can set this in your settings.xml file to globally use a certain mirror.

.m2 repository



```
.
├── repository
├── settings.xml
└── wrapper
```



```
.
├── antlr
│   └── antlr
├── aopalliance
│   └── aopalliance
├── archetype-catalog.xml
├── asm
│   ├── asm
│   ├── asm-analysis
│   ├── asm-commons
│   ├── asm-parent
│   ├── asm-tree
│   └── asm-util
├── avalon-framework
│   └── avalon-framework
├── backport-util-concurrent
│   └── backport-util-concurrent
├── biz
│   └── aQute
```

Dependency Management

https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html

To check dependency tree - mvn dependency:tree

Dependency management is a core feature of Maven. Managing dependencies is a difficult task once we've to deal with multi-module projects (consisting of hundreds of modules/sub-projects). Maven provides a high degree of control to manage such scenarios

Transitive Dependencies: Maven avoids the need to discover and specify the libraries that your own dependencies require by including transitive dependencies automatically. With transitive dependencies, the graph of included libraries can quickly grow quite large. For this reason, there are additional features that limit which dependencies are included:

1. *Dependency mediation* - this determines what version of an artifact will be chosen when multiple versions are encountered as dependencies. Maven picks the "nearest definition". That is, it uses the version of the closest dependency to your project in the tree of dependencies. You can always guarantee a version by declaring it explicitly in your project's POM (dependency management feature). Note that if two dependency versions are at the same depth in the dependency tree, the first declaration wins.
2. *Dependency management* - this allows project authors to directly specify the versions of artifacts to be used when they are encountered in transitive dependencies or in dependencies where no version has been specified.
3. *Dependency scope* - this allows you to only include dependencies appropriate for the current stage of the build. This is described in more detail below.
4. *Excluded dependencies* - If project X depends on project Y, and project Y depends on project Z, the owner of project X can explicitly exclude project Z as a dependency, using the "exclusion" element.
5. *Optional dependencies* - If project Y depends on project Z, the owner of project Y can mark project Z as an optional dependency, using the "optional" element. When project X depends on project Y, X will depend only on Y and not on Y's optional dependency Z. The owner of project X may then explicitly add a dependency on Z, at her option. (It may be helpful to think of optional dependencies as "excluded by default.")

Dependency Management

https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html

Dependency Scope: Dependency scope is used to limit the transitivity of a dependency and to determine when a dependency is included in a classpath.

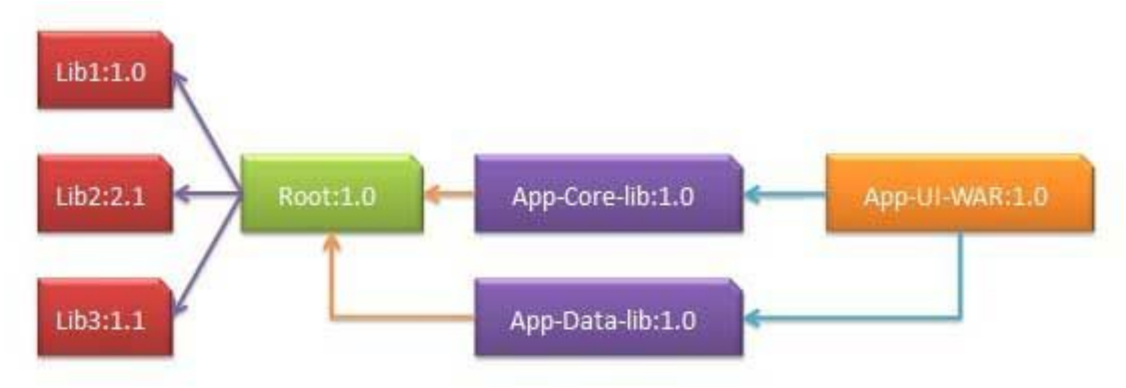
There are 6 scopes:

- **compile**
  This is the default scope, used if none is specified. Compile dependencies are available in all classpaths of a project. Furthermore, those dependencies are propagated to dependent projects.
- **provided**
  This is much like compile, but indicates you expect the JDK or a container to provide the dependency at runtime. For example, when building a web application for the Java Enterprise Edition, you would set the dependency on the Servlet API and related Java EE APIs to scope provided because the web container provides those classes. A dependency with this scope is added to the classpath used for compilation and test, but not the runtime classpath. It is not transitive.
- **runtime**
  This scope indicates that the dependency is not required for compilation, but is for execution. Maven includes a dependency with this scope in the runtime and test classpaths, but not the compile classpath.
- **test**
  This scope indicates that the dependency is not required for normal use of the application, and is only available for the test compilation and execution phases. This scope is not transitive. Typically this scope is used for test libraries such as JUnit and Mockito. It is also used for non-test libraries such as Apache Commons IO if those libraries are used in unit tests (src/test/java) but not in the model code (src/main/java).
- **system**
  This scope is similar to provided except that you have to provide the JAR which contains it explicitly. The artifact is always available and is not looked up in a repository.
- **import**
  This scope is only supported on a dependency of type pom in the <dependencyManagement> section. It indicates the dependency is to be replaced with the effective list of dependencies in the specified POM's <dependencyManagement> section. Since they are replaced, dependencies with a scope of import do not actually participate in limiting the transitivity of a dependency.

https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html

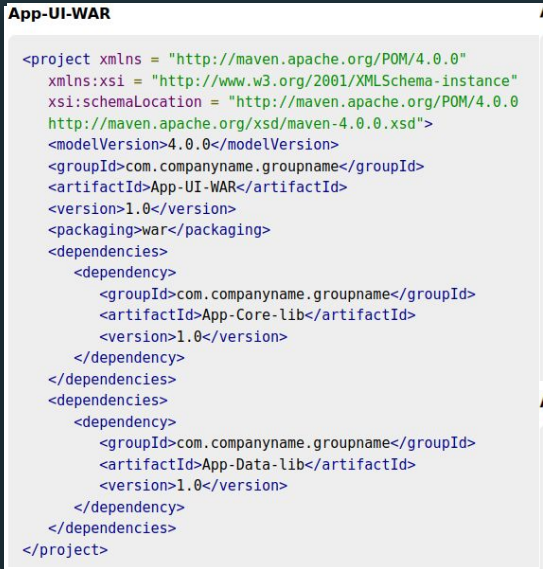
Dependency Management - Example

Parent POM: Usually, we have a set of project under a common project. In such case, we can create a common pom having all the common dependencies and then make this pom, the parent of sub-project's poms. Following example will help you understand this concept.



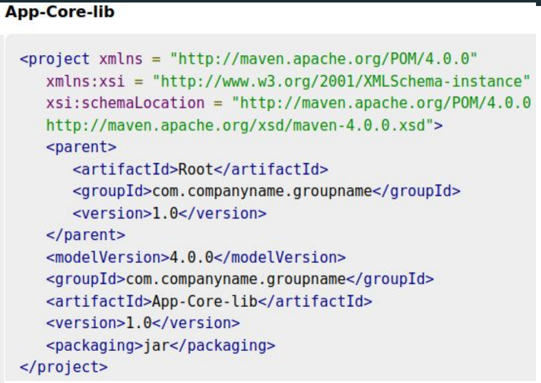
Following are the detail of the above dependency graph –

- App-UI-WAR depends upon App-Core-lib and App-Data-lib.
- Root is parent of App-Core-lib and App-Data-lib.
- Root defines Lib1, lib2, Lib3 as dependencies in its dependency section.



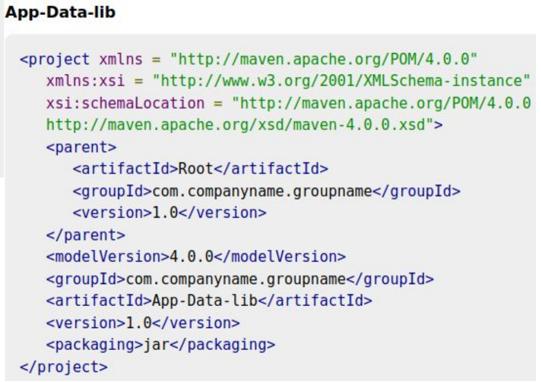
App-UI-WAR

```
<project xmlns = "http://maven.apache.org/POM/4.0.0"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.companyname.groupname</groupId>
   <artifactId>App-UI-WAR</artifactId>
   <version>1.0</version>
   <packaging>war</packaging>
   <dependencies>
      <dependency>
         <groupId>com.companyname.groupname</groupId>
         <artifactId>App-Core-lib</artifactId>
         <version>1.0</version>
      </dependency>
   </dependencies>
   <dependencies>
      <dependency>
         <groupId>com.companyname.groupname</groupId>
         <artifactId>App-Data-lib</artifactId>
         <version>1.0</version>
      </dependency>
   </dependencies>
</project>
```



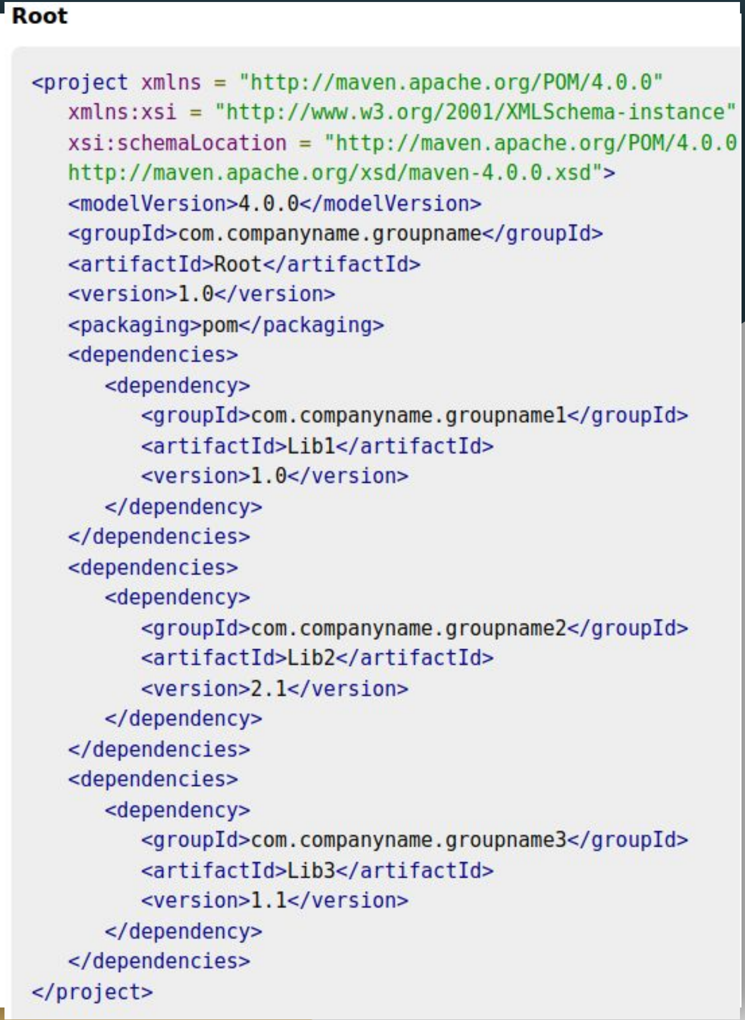
App-Core-lib

```
<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">
   <parent>
      <artifactId>Root</artifactId>
      <groupId>com.companyname.groupname</groupId>
      <version>1.0</version>
   </parent>
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.companyname.groupname</groupId>
   <artifactId>App-Core-lib</artifactId>
   <version>1.0</version>
   <packaging>jar</packaging>
</project>
```



App-Data-lib

```
<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">
   <parent>
      <artifactId>Root</artifactId>
      <groupId>com.companyname.groupname</groupId>
      <version>1.0</version>
   </parent>
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.companyname.groupname</groupId>
   <artifactId>App-Data-lib</artifactId>
   <version>1.0</version>
   <packaging>jar</packaging>
</project>
```



Root

```
<project xmlns = "http://maven.apache.org/POM/4.0.0"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>com.companyname.groupname</groupId>
   <artifactId>Root</artifactId>
   <version>1.0</version>
   <packaging>pom</packaging>
   <dependencies>
      <dependency>
         <groupId>com.companyname.groupname1</groupId>
         <artifactId>Lib1</artifactId>
         <version>1.0</version>
      </dependency>
   </dependencies>
   <dependencies>
      <dependency>
         <groupId>com.companyname.groupname2</groupId>
         <artifactId>Lib2</artifactId>
         <version>2.1</version>
      </dependency>
   </dependencies>
   <dependencies>
      <dependency>
         <groupId>com.companyname.groupname3</groupId>
         <artifactId>Lib3</artifactId>
         <version>1.1</version>
      </dependency>
   </dependencies>
</project>
```

From above example, we can learn the following key concepts −

- Common dependencies can be placed at single place using concept of parent pom. Dependencies of **App-Data-lib** and **App-Core-lib** project are listed in *Root* project (See the packaging type of Root. It is POM).
- There is no need to specify Lib1, lib2, Lib3 as dependency in App-UI-WAR. Maven use the **Transitive Dependency Mechanism** to manage such detail.

https://maven.apache.org/archetype/maven-archetype-plugin/usage.html

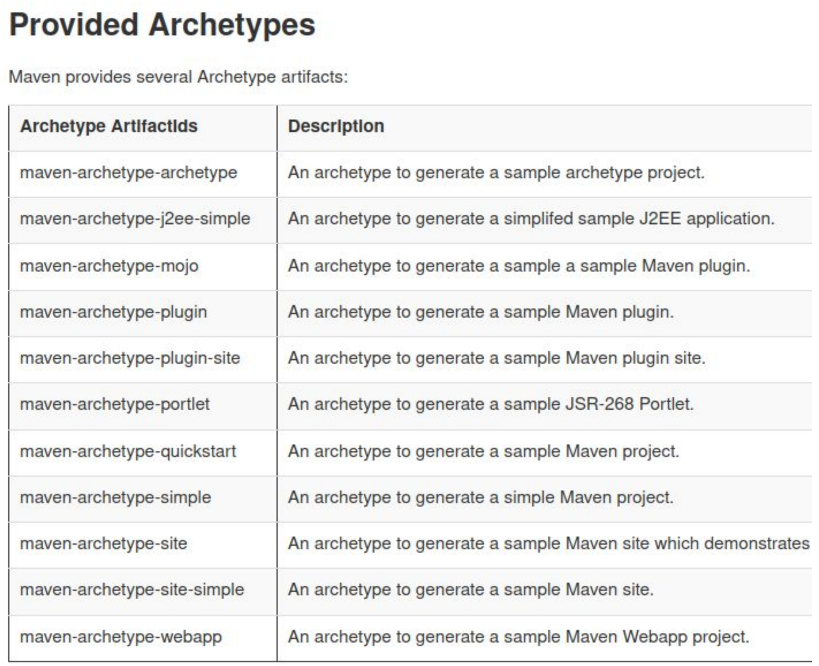
Generate maven project using Archetype

Archetype is a Maven project **templating toolkit**.
Maven archetype enables developer to quickly get sample project up & running by providing a project skeleton with base POM file for specific type of archetype

Using Archetype: **mvn archetype:generate**
Its an interactive tool which will ask set of question to user and based on answers will generate project skeleton.

What makes up an Archetype?

Archetypes are packaged up in a JAR and they consist of the archetype metadata which describes the contents of archetype, and a set of Velocity templates which make up the prototype project. If you would like to know how to make your own archetypes, please refer to our Guide to creating archetypes.

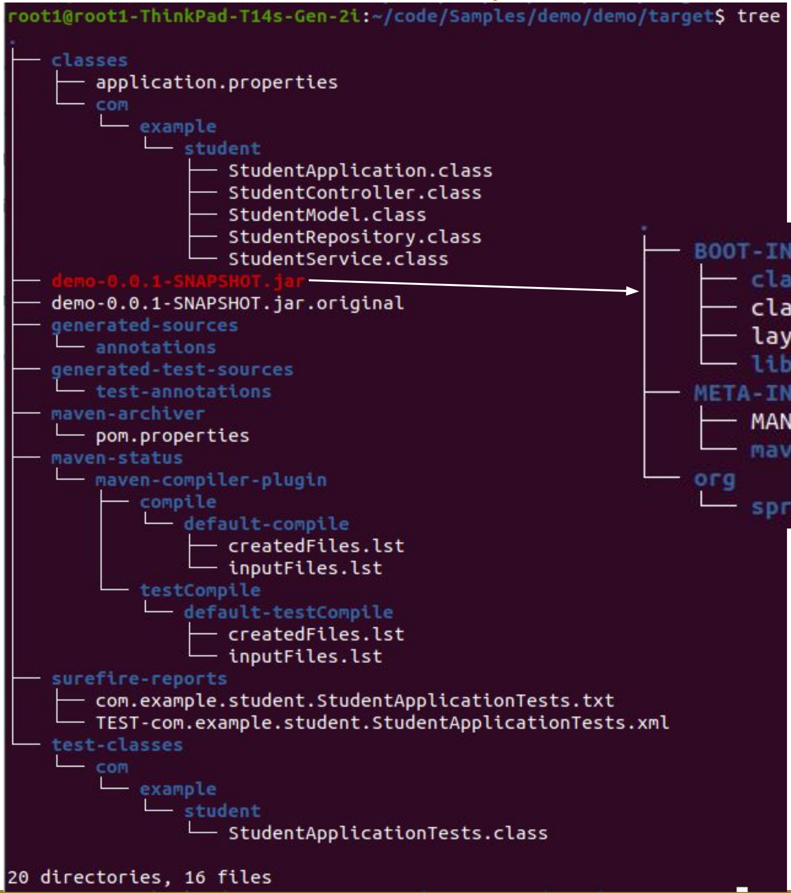


Provided Archetypes

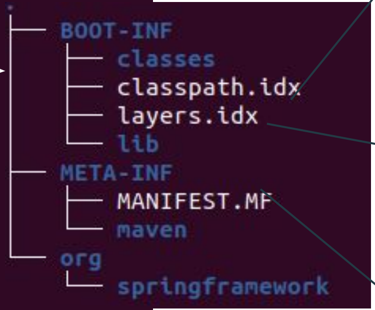
Maven provides several Archetype artifacts:

Archetype ArtifactIds	Description
maven-archetype-archetype	An archetype to generate a sample archetype project.
maven-archetype-j2ee-simple	An archetype to generate a simplifed sample J2EE application.
maven-archetype-mojo	An archetype to generate a sample a sample Maven plugin.
maven-archetype-plugin	An archetype to generate a sample Maven plugin.
maven-archetype-plugin-site	An archetype to generate a sample Maven plugin site.
maven-archetype-portlet	An archetype to generate a sample JSR-268 Portlet.
maven-archetype-quickstart	An archetype to generate a sample Maven project.
maven-archetype-simple	An archetype to generate a simple Maven project.
maven-archetype-site	An archetype to generate a sample Maven site which demonstrates
maven-archetype-site-simple	An archetype to generate a sample Maven site.
maven-archetype-webapp	An archetype to generate a sample Maven Webapp project.

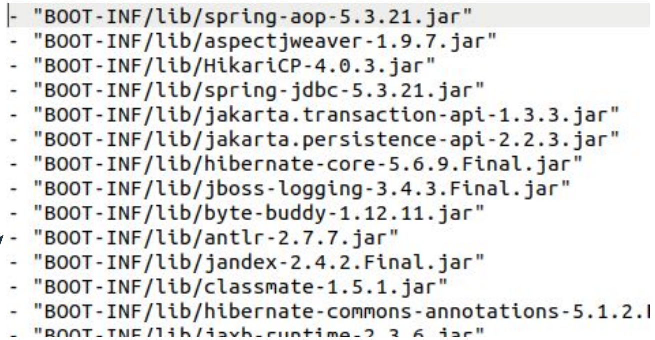
Generated Output



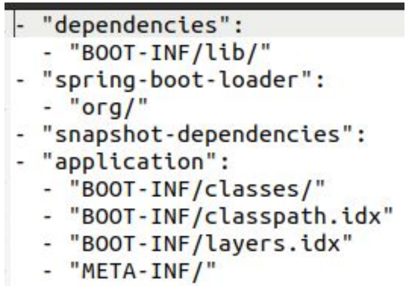
```
root1@root1-ThinkPad-T14s-Gen-2i:~/code/Samples/demo/demo/target$ tree
.
├── classes
│   ├── application.properties
│   └── com
│       └── example
│           └── student
│               ├── StudentApplication.class
│               ├── StudentController.class
│               ├── StudentModel.class
│               ├── StudentRepository.class
│               └── StudentService.class
├── demo-0.0.1-SNAPSHOT.jar
├── demo-0.0.1-SNAPSHOT.jar.original
├── generated-sources
│   └── annotations
├── generated-test-sources
│   └── test-annotations
├── maven-archiver
│   └── pom.properties
├── maven-status
│   └── maven-compiler-plugin
│       ├── compile
│       │   └── default-compile
│       │       ├── createdFiles.lst
│       │       └── inputFiles.lst
│       └── testCompile
│           └── default-testCompile
│               ├── createdFiles.lst
│               └── inputFiles.lst
├── surefire-reports
│   ├── com.example.student.StudentApplicationTests.txt
│   └── TEST-com.example.student.StudentApplicationTests.xml
└── test-classes
    └── com
        └── example
            └── student
                └── StudentApplicationTests.class

20 directories, 16 files
```



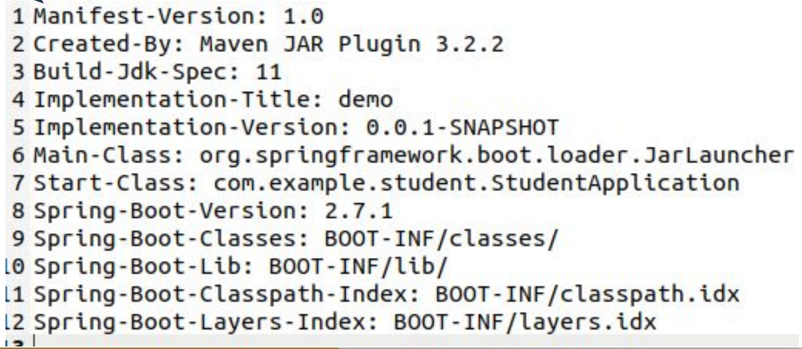
```
.
├── BOOT-INF
│   ├── classes
│   ├── classpath.idx
│   ├── layers.idx
│   └── lib
├── META-INF
│   ├── MANIFEST.MF
│   └── maven
└── org
    └── springframework
```



```
- "BOOT-INF/lib/spring-aop-5.3.21.jar"
- "BOOT-INF/lib/aspectjweaver-1.9.7.jar"
- "BOOT-INF/lib/HikariCP-4.0.3.jar"
- "BOOT-INF/lib/spring-jdbc-5.3.21.jar"
- "BOOT-INF/lib/jakarta.transaction-api-1.3.3.jar"
- "BOOT-INF/lib/jakarta.persistence-api-2.2.3.jar"
- "BOOT-INF/lib/hibernate-core-5.6.9.Final.jar"
- "BOOT-INF/lib/jboss-logging-3.4.3.Final.jar"
- "BOOT-INF/lib/byte-buddy-1.12.11.jar"
- "BOOT-INF/lib/antlr-2.7.7.jar"
- "BOOT-INF/lib/jandex-2.4.2.Final.jar"
- "BOOT-INF/lib/classmate-1.5.1.jar"
- "BOOT-INF/lib/hibernate-commons-annotations-5.1.2.
- "BOOT-INF/lib/jaxb-runtime-2.3.6.jar"
```



```
- "dependencies":
  - "BOOT-INF/lib/"
- "spring-boot-loader":
  - "org/"
- "snapshot-dependencies":
- "application":
  - "BOOT-INF/classes/"
  - "BOOT-INF/classpath.idx"
  - "BOOT-INF/layers.idx"
  - "META-INF/"
```

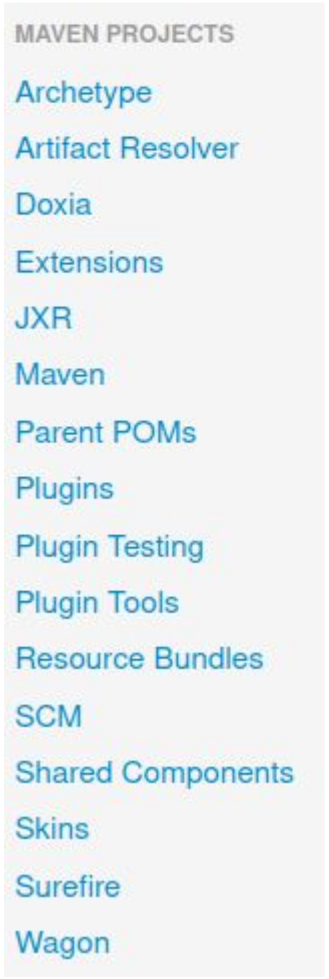


```
Manifest-Version: 1.0
Created-By: Maven JAR Plugin 3.2.2
Build-Jdk-Spec: 11
Implementation-Title: demo
Implementation-Version: 0.0.1-SNAPSHOT
Main-Class: org.springframework.boot.loader.JarLauncher
Start-Class: com.example.student.StudentApplication
Spring-Boot-Version: 2.7.1
Spring-Boot-Classes: BOOT-INF/classes/
Spring-Boot-Lib: BOOT-INF/lib/
Spring-Boot-Classpath-Index: BOOT-INF/classpath.idx
Spring-Boot-Layers-Index: BOOT-INF/layers.idx
```

Thanks

Maven: Features

- **Simple & predefined project setup:** that follows best practices with consistent usage across all projects.
- **Dependency management & Versioning:** including automatic updating using versions.
- **Large Central Repository:** A large and growing repository of libraries.
- **Extensible**, with the ability to easily write plugins in Java or scripting languages
- **Model-based builds** − Maven is able to build any number of projects into predefined output types such as jar, war, metadata.
- **Documentation** − Using the same metadata as per the build process, maven is able to generate a html and a PDF including complete documentation.
- **Release management and distribution publication** − Without additional configuration, maven will integrate with your source control system such as CVS and manages the release of a project.
- **Backward Compatibility** − You can easily port the multiple modules of a project into Maven 3 from older versions of Maven. It can support the older versions also.
- **Inheritance -** Parent POM usage in sub modules.
- **Parallel builds** − It analyzes the project dependency graph and enables you to build schedule modules in parallel. Using this, you can achieve the performance improvements of 20-50%.
- **Better Error and Integrity Reporting** − Maven improved error reporting, and it provides you with a link to the Maven wiki page where you will get full description of the error.



MAVEN PROJECTS

Archetype
Artifact Resolver
Doxia
Extensions
JXR
Maven
Parent POMs
Plugins
Plugin Testing
Plugin Tools
Resource Bundles
SCM
Shared Components
Skins
Surefire
Wagon

<proj-dir>/target vs ~/.m2

https://stackoverflow.com/questions/40979487/target-folder-vs-local-repository

- target represents the build directory. This is to say, every temporary file that is generated during the build from the sources ends up there. Quite notably, you'll find the compiled classes of the main and test Java sources, but you'll also find lots of things in there (generated source files, filtered files, etc.). What matters, is that everything that is contained in this folder is inherently temporary. You can delete it at any time, running mvn clean, and be assured that the next build will (or at least *should*) work just fine. All the files and folders generated under target serve a single purpose: create the artifacts of the project. A Maven project, for example with jar packaging, will have a single main artifact, which is composed of its final name with a jar extension, and will contain the compiled Java classes. The final name can be a custom name, set within the POM, or the default one derived from the Maven coordinates of the project. Such a project can also have additional attached artifacts, like a test JAR, or a sources JAR.
- The local repository only contains the artifacts. There are no temporary files in there. What is installed when running mvn install is strictly the generated artifacts of the Maven project, i.e. the end products, plus the POM file of the project. Everything that served to create them isn't put in the local repository, and the build of a project must never put temporary things in there. Keep in mind that the local repository is a Maven repository, and, as such, follows a strict naming scheme: a project with a group id of my.groupid, an artifact id of my-artifactid and a version of 1.0 will get installed in the folder my/groupid/my-artifactid/1.0; in which you'll find the POM file, and all the other artifacts. The name of the artifacts themselves cannot be overridden: it will be my-artifactid-1.0.jar for a JAR project (perhaps with a classifier added).

This is generally a source of confusion: the name of the main artifact file that is generated under the target folder is completely distinct from the name that it will have in the local repository when installed, or in remote repositories when deployed. The first can be controlled, but the latter is defined by the naming scheme of the repository, which is calculated from the coordinates.

To recap: target contains all the gory temporary details during the build which creates the artifacts of a project (main JAR, sources, Javadoc... i.e. everything that is supposed to be deployed and released by that project), while the local repository (and remote repositories) will contain only the artifacts themselves.