export MAVEN\_HOME=/usr/local/apache-maven/apache-maven-3.5.0

export MAVEN=$MAVEN\_HOME/bin

video course

complete video-33

pending – see section 5- maven basics one more time- bec I didn’t took the notes

practice –run the phase and direct plugin to see diff

Reference urls

|  |  |
| --- | --- |
|  | <https://docs.oracle.com/middleware/12211/lcm/MAVEN/real_app.htm#A1017144> |
| maven home | https://maven.apache.org/ref/3.9.9/ |
| all plugins | <https://maven.apache.org/plugins/index.html>  <https://maven.apache.org/plugin-tools/maven-plugin-plugin/plugin-management.html> |
| project uses jaxb plugin to generate stubs | <https://github.com/springframeworkguru/mb2g-jaxb/blob/master/src/main/resources/jaxb.xsd> |
| project uses json schema to generate java stubs | <https://github.com/springframeworkguru/mb2g-json-schema/tree/master/src/main/resources/schema> |
|  |  |

About maven

We can also create custom maven plugins

Before maven:

Generally before maven if u want to compile a single class file, if that class is using any third party libraries, ,(first u should have downloaded them) u have to include those Libraries located path in the classpath as below

Ex:-

Import org.apaceh.lang3.StringUtils;

// the above class belongs to commonutils librarby , I have downloaded that jar and kept in lib folder

Class mani{

Main{ system.out.println(StringUtils.isnull(“mani”)); }

}

U have to include the jar path in classpath

java –classpath ./lib/\*: ./ Helloworld

java packaging

jar, - zip file contains only class files

war – zip file with class files+ web resources like img, scripts…

war, ear, -these needs to be deployed into web servers like tomcat/ glass fish

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
| fat jar/Uber jar | (source code jar + all dependencies) this fat jar is mostly used by spring boot  these are complete apps which contain even embedded application servers- can be deployed stand alone |

Terminologies

|  |  |
| --- | --- |
| Maven co-ordinates | Group id , artifact id ,version are called  group id = organization reverse domain is used  artifact id = project name  group id and version can be inherited from parent pom |
| snapshot | this means that is a development build not final build |
| maven repo | repo is where all our jars/artifacts stored  local,  remote – can be public or private- every company have remote private repo  central repo-public repo hosted by maven  generally maven will look in local repo, if not found then it will go to central repo |
|  |  |
|  |  |
|  |  |
|  |  |

Lifecycle

Maven is based on concept of build life cycle (LPG-lifecycle-phase-goal🡪each phase will have lot of goals)

Humans have many phases like our-childhood, young age, old age- and each phase have some tasks

In maven also a lifecycle=set of phases

Each phase =set of many plugin goals (like us in young age, we have goals called job, marriage, kids)

All the work is done by plugins,

Lifecycle and phases provide the framework to call plugin goals in a sequence

**There are three built-in lifecycles:**

* default: the main lifecycle, as it's responsible for project deployment
* clean: to clean the project and remove all files generated by the previous build
* site: to create the project's site documentation

**Each lifecycle consists of a sequence of phases.**

The default build lifecycle consists of 23 phases, as it's the main build lifecycle(this doesn’t have the clean phase at all , bec clean is a diff lifecycle) .

Ex:- a man lifecycle will have many phases- kid phase, student,business man, old age phase (if u want to exec any/last phase all the before phases will be executed automaticaly same like human life phase)

|  |  |
| --- | --- |
| mvn <phase name1> <phase name2> | mvn clean install  since clean phase is not in default lifecycle, we are mentioning clean phase too |

On the other hand, the *clean* life cycle consists of 3 phases, while the *site* lifecycle is made up of 4 phases.

Each of these build lifecycles is defined by a different list of build phases, wherein a build phase represents a stage in the lifecycle.

For example, the default lifecycle comprises of the following phases (for a complete list of the lifecycle phases, refer to the [Lifecycle Reference](https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference)):

Below are the phases of Default life cycle

|  |
| --- |
| 1. <phases> 2. <phase>validate</phase> 3. <phase>initialize</phase> 4. <phase>generate-sources</phase> 5. <phase>process-sources</phase> 6. <phase>generate-resources</phase> 7. <phase>process-resources</phase> 8. <phase>compile</phase> 9. <phase>process-classes</phase> 10. <phase>generate-test-sources</phase> 11. <phase>process-test-sources</phase> 12. <phase>generate-test-resources</phase> 13. <phase>process-test-resources</phase> 14. <phase>test-compile</phase> 15. <phase>process-test-classes</phase> 16. <phase>test</phase> 17. <phase>prepare-package</phase> 18. <phase>package</phase> 19. <phase>pre-integration-test</phase> 20. <phase>integration-test</phase> 21. <phase>post-integration-test</phase> 22. <phase>verify</phase> 23. <phase>install</phase> 24. <phase>deploy</phase> 25. </phases> |

|  |  |  |  |
| --- | --- | --- | --- |
| This is the primary lifecycle Maven uses to build and distribute your project. It consists of a sequence of well-defined phases. When you execute a specific phase, Maven will also execute all preceding phases in order.  Here are the main phases of the default lifecycle in Maven, in the order they are executed:   1. validate: Validates if the project is correctly set up and all necessary information is available in the pom.xml. 2. **initialize**: Initializes the build state, setting up properties and environment. 3. **generate-sources**: Generates any source code that needs to be compiled. 4. **process-sources**: Processes the source code, for example, by filtering values. 5. **generate-resources**: Generates resources that need to be included in the package. 6. **process-resources**: Copies and processes the resources into the output directory, preparing them for packaging. 7. compile: Compiles the project's source code. 8. **process-classes**: Post-processes the generated class files (e.g., for bytecode enhancement). 9. **generate-test-sources**: Generates any test source code. 10. **process-test-sources**: Processes the test source code. 11. **generate-test-resources**: Generates resources needed for testing. 12. **process-test-resources**: Copies and processes test resources. 13. **test-compile**: Compiles the test source code. 14. **process-test-classes**: Post-processes the generated test class files. 15. test: Executes the unit tests using a suitable testing framework.   (These tests should not require the code be packaged or deployed) 16. **prepare-package**: Performs any necessary actions to prepare the package before the actual packaging. 17. package: Takes the compiled code and resources and packages them into a distributable format (e.g., JAR, WAR, EAR). 18. **pre-integration-test**: Executes actions required before integration tests. 19. **integration-test**: Processes and deploys the package if necessary for running integration tests. 20. **post-integration-test**: Executes actions required after integration tests. 21. verify: Runs checks to ensure the package is valid and meets quality criteria. 22. install: Installs the packaged artifact into your local Maven repository (.m2 directory), making it available as a dependency for other local projects. 23. deploy: Copies the final packaged artifact to a remote repository for sharing with other developers and projects. This is typically done in a build server environment.   When you run a Maven command like mvn package, Maven will execute all the phases up to and including the package phase in the order listed above.  Similarly, mvn install will execute all phases up to install, and mvn deploy will execute the entire default lifecycle.  Understanding these phases is crucial for customizing your Maven build process by binding plugins and their goals to specific phases to perform various tasks  Note:- this default lifecycle doesn’t have clean phase at all , hence even if u execute install/ deploy phase (of default lifecycle) cleaning the target directory will not happen at all, so we have to manually include that clean plugin and we have to tell when to run that plugin/in which phase we should run the plugin  Note2:- these phases are very important, because every plugin will do some work,  when we configure plugin, we should only tell when to run the plugin (means we should tell in which phase of maven build lifecycle we should execute that plugin)   |  |  | | --- | --- | | <plugin>  <groupId>org.apache.cxf</groupId>  <artifactId>cxf-codegen-plugin</artifactId>  <version>3.3.0</version>  <executions>  <execution>  <id>generate-sources</id>  <phase>**generate-sources**</phase> | see here we configured that plugin and we only told when we should execute that plugin – like we should execute that plugin during compilation phase/ install phase or test phase or deploy phase? And here we told to execute that plugin during “**generate-sources” phase** | | here intellij is showing only the main phases of 3 lifecylces- default, clean,site |

These lifecycle phases (plus the other lifecycle phases not shown here) are executed sequentially to complete the default lifecycle. Given the lifecycle phases above, this means that when the default lifecycle is used, Maven will first validate the project, then will try to compile the sources, run those against the tests, package the binaries (e.g. jar), run integration tests against that package, verify the integration tests, install the verified package to the local repository, then deploy the installed package to a remote repository.

Plugins

1 plugin will do many tasks as 1 plugin will have many goals, like 1 man will do many tasks as he have many goals

|  |  |  |  |
| --- | --- | --- | --- |
| plugins | goals   |  |  | | --- | --- | |  | here inside lifecycle, we are seeing the phases as 1 lifecylce will have many phases  so here instead of running a phase from lifecycle, we can even run a goal from plugins tab  here we can execute the particular goal of a plugin  should we run direct plugin? Or should we run the lifecycle?   * If we run the phase like install/delpoy all the previous phases will be executed, and if u configure any plugin to run at any phase, even that plugin also will be executed (if that phase is before) * If we execute any goal of a plugin then only that goal will be executed   so always recommended to run phase instead of direct plugin | |
| clean | • Build Lifecycle - CLEAN  • Has only one goal - ‘clean’  • Purpose is to remove files generated during build process.  • By default removes /target directory project root and submodule root folders  has 1 gooal – clean to delete the target folder and it will delete all the generated jars  Note:- this clean plugin will not be executed automatically as and when we run “install” “package” lifecycle |
|  |  |

|  |  |  |
| --- | --- | --- |
| when u run this install / package phases , clean phase will not be executed automatically as this clean phase is from diff lifecycle & package,install phases are from diff lifecycle  to execute this clean, plugin automatically, then add this plugin | <build>  <plugins>  <plugin>  <artifactId>maven-clean-plugin</artifactId>  <version>3.4.1</version>  <executions>  <execution>  <id>auto-clean</id>  <phase>initialize</phase>  <goals>  <goal>clean</goal>  </goals>  </execution>  </executions>  </plugin>  </plugins>  </build> | **What does this achieve?**  **we are trying to execute the goal (clean) of plugin (maven-clean-plugin) during maven’s** initialize **phase of a default lifecycle (this initialise is one of 23 phases of maven default life cycle)**  By configuring the maven-clean-plugin to run during the initialize phase, you ensure that the project's target directory (which contains compiled classes, packaged artifacts, etc. from previous builds) is cleaned up very early in the build process, even before compilation begins   **<executions>:** This section allows you to configure when and how the plugin's goals are executed. A plugin can have multiple goals, and you can define multiple <execution> blocks to run these goals at different phases of the Maven lifecycle or with different configurations.   **<execution>:** This defines a single execution of the plugin's goals.   * **<id>auto-clean</id>:** This provides a unique identifier for this specific execution. This is useful if you have multiple executions of the same plugin with different configurations. * **<phase>initialize</phase>:** This specifies the Maven lifecycle phase during which the plugin's goals in this execution will be run. The initialize phase is one of the early phases in the build lifecycle, occurring before the compilation phase. * **<goals>:** This lists the specific goals of the plugin that should be executed during the specified phase.   + **<goal>clean</goal>:** This is the goal provided by the maven-clean-plugin that performs the actual cleaning of the project's build output (deleting the target directory). |

Below are the plugins of default lifecycle (every plugin will have many and different goals/work)

|  |  |
| --- | --- |
| plugins | info |
| compile | |  | | --- | | this compile plugin belongs to Build Lifecycle – DEFAULT  • Has two goals - compiler:compile, compiler:testCompile  • By Default uses the compiler ‘javax.tools.JavaCompiler  • Can be configured to use javac if needed  • Default source and target language levels are Java 1.6 |   • Apache team encourages these values to be set |

|  |  |  |  |
| --- | --- | --- | --- |
| resource | Build Lifecycle – DEFAULT  this resources plugin is needed especially when we want to move the files from 1 place to another place –example copying files from src/main/resources to target folder   |  |  | | --- | --- | | Purpose is to copy project resources to output directory (target dir)  • Has 3 goals - resources:resources, resources:testResources, resources:copy-resources  • Can be configured for encoding, source and target directories  • Rather versatile configuration options for copying files during build processing  by default it will copy only .class files to target folder & if u want to copy other files like cert, then we have to tell specifically saying copy this file also to target folder and include that file in jar creation  ex-in some cases, we need to include image files, certificates, jks , css, js files |  |   •  by default this plugin will look for resources in src/main/resources  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-resources-plugin</artifactId>  <version>${maven-resources-plugin.version}</version> </plugin>  u can even customize this plugin to change where it should copy from and where it should paste in the destination |
| surefire | |  |  | | --- | --- | | this is the real plugin which executes our test cases  Build Lifecycle - DEFAULT  • Has one goal: surefire:test  • The Surefire plugin is used to execute unit test of the project.  • By default supports JUnit 3, JUnit 4, JUnit 5, and TestNG  • Cucumber runs under JUnit, Spock compiles to JUnit byte code.  1 condition :- by default this plugin will look for /run the classes  whose names starting/ending with Test/Test  • \*\*/Test\*.java; \*\*/\*Test.java; \*\*/\*Tests.java; \*\*/\*TestCase.java | <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-surefire-plugin</artifactId>  <version>${maven-surefire-plugin.version}</version> </plugin> | |

#### Jar&deploy &install plugins

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| jar | •   |  |  | | --- | --- | | Build Lifecycle - DEFAULT  • Has two goals: jar:jar, jar:test-jar  • **Purpose is to build jars** from complied artifacts and project resources like files present in src/main/resources folder  • Can be configured for custom manifests, and to make executable jars.  this plugin reads the info present in manifest file before creating jar, |  | | Make The Jar Executable(to make jar executable, manifest file should have the main class name, else which class will be executed when we run the jar, because each jar will have lot of classes, so we only have to tell which cls to exec and we should keep that manifest file in the jar ) <build>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-jar-plugin</artifactId>  <configuration>  <archive>  <manifest>  <addClasspath>true</addClasspath>  **<mainClass>com.ampf.Child1App</mainClass>**  </manifest>  </archive>  </configuration>  </plugin>  </plugins>  </build>  The manifest produced using the above configuration would look like this: (the manifest file will be present inside the jar )   1. Manifest-Version: 1.0 2. Created-By: Apache Maven ${maven.version} 3. Build-Jdk: ${java.version} 4. Main-Class: **com.ampf.Child1App** //**here as this main class file name is present when we run the jar this cls will be executed** 5. Class-Path: plexus-utils-1.1.jar commons-lang-2.1.jar //Even these jars must be present in the same folder where our jar is- else jar wont run at all   now see this resultant manifest file have main class name, hence this jar is runnable , & when we run the jar it will run the main method present in the class  E:\study related\my git hub -new\Practice-projects\maven\ParentProj\child1\target>java -jar child1-1.0-SNAPSHOT.jar  Hello Bro i am a executable jar! |  |   if u are making thin jar then dependencies are not included (only source code will be present in that jar)all the other jars should be there in the classpath or the jars should be in same folder where our jar resides |
| Deploy | |  |  | | --- | --- | | Build Lifecycle - DEFAULT  • Has two goals - deploy:deploy, deploy:deploy-file  • Purpose is to deploy project artifacts to remote Maven repositories like jfrog  • Often done in CI  • Configuration is typically part of the Maven POM |  | |
| site | Has 7 goals:  • site:site - Generate site for project  • site:deploy - Deploy site via Wagon  • site:run - Run Site locally using Jetty as web server  • site:stage - generate site to a local staging directory  • site:stage-deploy - Deploy site to remote staging location  Site Plugin Goals (continued):  • site:attach-descriptor - adds site.xml (site map file used by search engines) to files for  deployment  • site:jar - bundles site into a jar for deployment to a repository  • site:effective-site - generates the site.xml file |

Jaxb plugin

This jaxb maven plugin will generate the stubs/java classes from the xsd file(it will take the xsd file from src/main/resuorces) it will generate and keep stubs in target folder

<https://github.com/springframeworkguru/mb2g-jaxb/blob/master/src/main/resources/jaxb.xsd>

|  |  |  |
| --- | --- | --- |
| <build>  <plugins>  <plugin>  <groupId>org.jvnet.jaxb2.maven2</groupId>  <artifactId>maven-jaxb2-plugin</artifactId>  <version>0.14.0</version>  <executions>  <execution>  <goals>  <goal>generate</goal>  </goals>  </execution>  </executions>  </plugin>  </plugins>  </build> | these 3 dependencies are required to generate stubs  <dependency>  <groupId>javax.xml.bind</groupId>  <artifactId>jaxb-api</artifactId>  <version>${jaxb.version}</version>  </dependency>  <dependency>  <groupId>com.sun.xml.bind</groupId>  <artifactId>jaxb-core</artifactId>  <version>${jaxb.version}</version>  </dependency>  <dependency>  <groupId>com.sun.xml.bind</groupId>  <artifactId>jaxb-impl</artifactId>  <version>${jaxb.version}</version>  </dependency> | here all the genereated java classes are kept in folder “generated-sources” |

Json schema to pojo plugin

jsonschema to pojo plugin to generate stubs/classes from json schema file

|  |  |  |
| --- | --- | --- |
| <plugin>  <groupId>org.jsonschema2pojo</groupId>  <artifactId>jsonschema2pojo-maven-plugin</artifactId>  <version>0.5.1</version>  <configuration>  <sourceDirectory>${basedir}/src/main/resources/jsonschema</sourceDirectory>  <targetPackage>guru.springframework.model</targetPackage>  <useCommonsLang3>true</useCommonsLang3>  </configuration>  <executions>  <execution>  <goals>  <goal>generate</goal>  </goals>  </execution>  </executions>  </plugin> | these 2 dependencies are required to generate stubs from json schema  <dependencies>  <dependency>  <groupId>org.apache.commons</groupId>  <artifactId>commons-lang3</artifactId>  <version>3.8.1</version>  </dependency>  <dependency>  <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-databind</artifactId>  <version>2.9.7</version>  </dependency>  </dependencies> | the generated classes are kept in target folder |
| here this plugin will take the schema files present in the source directory and It will use those dependecies and generate the stubs and it will keep those stubs in target folder |  |  |

Site life cycle

This site life cycle is used to create a web-site for ur project , defined with plugin bindings, this I least used

Commands

|  |  |
| --- | --- |
| To see the dependencies list | mvn dependency:tree |
|  | mvn dependency:go-offline |
|  | mvn dependency:purge-local-repository |
| To pull all the sources-jars  jar is nothing but a zip file | mvn dependency:sources |
| to run jar file | java –jar <location & jar file name>  generally even to execute any java file also we use java command, so here even to execute jars we used java command |
| run any maven phase | mvn <phase name1> <phase name 2>  mvn package //u can write any 1/23 phases of a default life cycle  mvn clean  mvn clean install 🡪 I gave here 2 phases clean, install – clean phase from clean lifecycle, install phase from default lifecycle |
| run any maven goal | mvn <plugin name> : <goal name belongs to a plugin>  mvn clean:clean  mvn install:install |
| to see all the dependencies | mvn dependency:tree |
| to see the final pom | mvn help:effective-pom  ex:- with this command, u will get the final pom after inheriting all the dependencies from parent pon |

Maven variables

<name>SOAP-2-StubsGenFromWSDLfile</name>

|  |  |
| --- | --- |
| <sourceRoot>  ${project.build.directory}/generated/cxf  </sourceRoot> | here project.build.directory means “target” folder |
| <wsdlOptions>  <wsdlOption>  <wsdl>${basedir}/src/main/resources/CustomerOrders.wsdl</wsdl> | In a Maven pom.xml file, ${basedir} is a **predefined property** that refers to the **absolute path of the directory containing the current pom.xml file**   * **here to refer some property files from src/main/resources folder we are using this**   **Source and Resource Directories:** Specifying the location of your Java source code (${basedir}/src/main/java), resources (${basedir}/src/main/resources), test source code (${basedir}/src/test/java), and test resources |
|  |  |

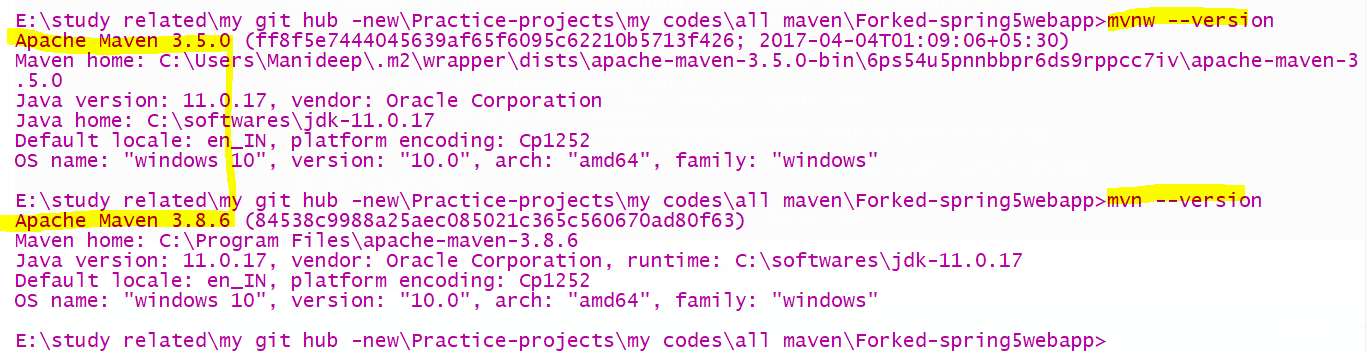
Maven wrapper

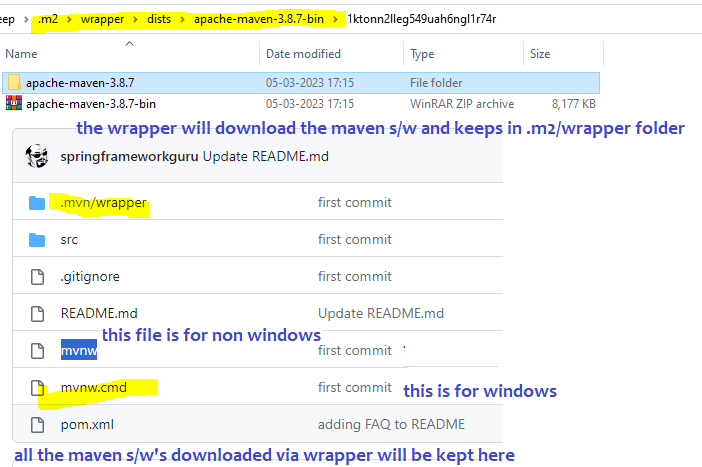
Mvnw == mavenwrapper

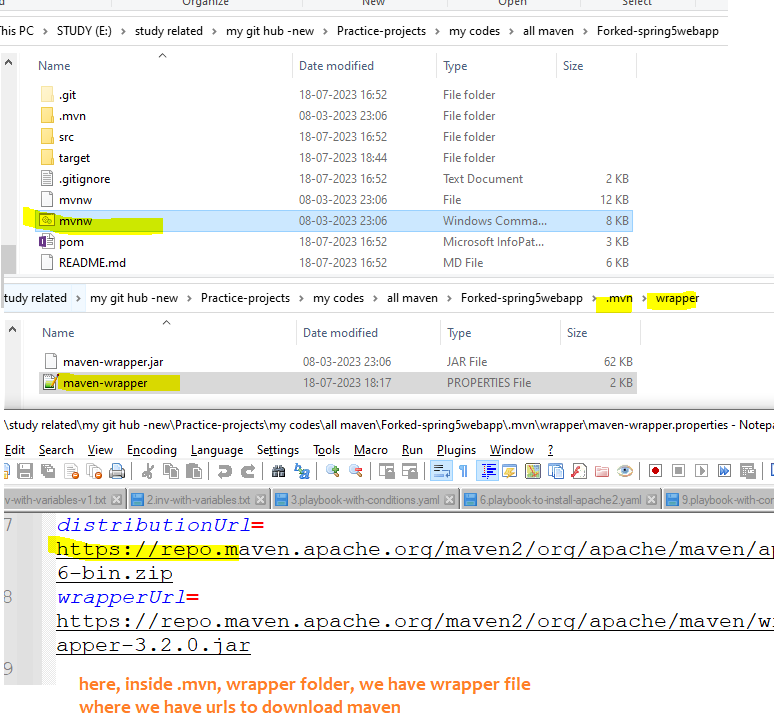
Maven wrapper is just a wrapper around maven software, this wrapper allows us to work with maven without having maven installed.

Because this wrapper itself will download and install maven

If u don’t have maven installed, then we can work with maven wrapper, which internally work with maven







Here if u see “mvnw” means it Is a tool called maven wrapper (just like javac), , this maven wrapper will download the original maven

When u download the project, then itself in the code itself u will get mvnw (maven wrapper ) file

Now U can do both mnv test as well as mvnw test

This wrapper is inspired from gradle wrapper,

Archetypes

He selected sitesimple archetype

## Excluding a dependency

1. <dependency>
2. <groupId>sample.ProjectA</groupId>
3. <artifactId>Project-A</artifactId>
4. <version>1.0</version>
5. <scope>compile</scope>
6. <exclusions>
7. <exclusion> <!-- declare the exclusion here -->
8. <groupId>sample.ProjectB</groupId>
9. <artifactId>Project-B</artifactId>
10. </exclusion>
11. </exclusions>
12. </dependency>

Plugins

1 plugin have many goals (Lifecycle 🡪 plugin🡪Goals)

Plugins are what perform the actual build tasks within those lifecycles. (plugins are like workers in a company, each plugin /person will have some goals/tasks)

For example,

the maven-compiler-plugin compiles your Java code,

the maven-jar-plugin packages it into a JAR file,

and the maven-war-plugin creates WAR files for web applications

**Task Execution:** Plugins execute specific tasks or goals. Each plugin can have multiple goals. For instance, the maven-compiler-plugin has compile and testCompile goals

1. clean plugin have goal called “clean”
2. default plugin have goals named “compile” , “testcompile””

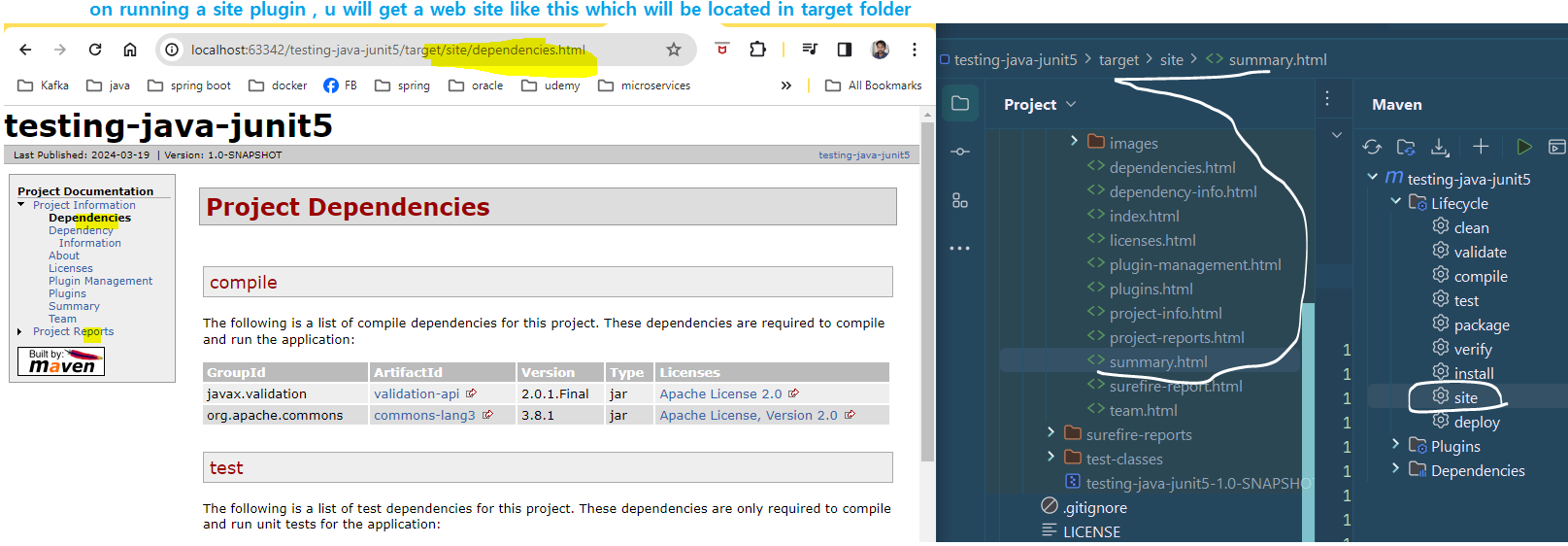
|  |  |
| --- | --- |
| Sure fire plugin | To run unit test |
| Sure fire plugin with tags | <build>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-surefire-plugin</artifactId>  <version>2.22.0</version>  <configuration>  <argLine>  --illegal-access=permit  </argLine>  <!--<groups>model</groups>--> <!This means only tests which are tagged with model will be executed @Tag(“model”)>  <!--<excludedGroups>controllers</excludedGroups>-->  </configuration>  </plugin> |
| Fail safe plugin | To handle integration tests |
| Site plugin | When u run “site“ then a clean big report will be generated about the tests how many ignored and coverage |
|  |  |
|  |  |

Site plugin : source code to add this plugin is found at

<https://github.com/springframeworkguru/testing-java-junit5/blob/surefire-report/pom.xml>

To generate like below website u need to add

|  |  |
| --- | --- |
| <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-site-plugin</artifactId>  <version>3.7.1</version>  </plugin>  </plugins>  </build> | <reporting>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-surefire-report-plugin</artifactId>  <version>2.22.0</version>  </plugin>  </plugins>  </reporting>  </project> |



Pom

POM - Project Object Model • The pom.xml is a XML document which describes a Maven Project • Must comply with the maven-4.0.0.xsd • XML Schema defining the ‘rules’ for the XML Document •

POMs can inherit properties from a parent POM • sub-modules also inherit •

‘Effective POM’ - is the final complete POM with inherited properties • mvn help:effective-pom

Or u can see effective pom as in intellij “right click on pom.xml”🡪 mave 🡪show effective pom



#### Few pom syntaxes

1. every pom.xml (even sub module should have a name ) – this name only will be appearing in the maven console

artifact name is the finar jar name

|  |  |
| --- | --- |
| <name>child1</name> <url>http://maven.apache.org</url>  in child1 pom.xml u should have above entry  this name only will appear in intellij maven | build console  [INFO] ParentProj ................................... SUCCESS [ 0.884 s]  [INFO] child1 ............................................. SUCCESS [ 3.730 s]  [INFO] child2 ............................................. SUCCESS [ 1.330 s] |
|  |  |

1. fg

Dependencies

Dependency Mediation - Determines what version to use when multiple versions of the same

dependency are encountered.

• ‘Nearest Definition’ in dependency tree is used

• Example: A -> B; A -> D 2.0; B -> D 1.5

• D version 2.0 would be included

• Excluded Dependencies - Ability to exclude specific dependencies

• Optional Dependencies - Ability to make dependencies optional. (excluded by default for downstream projects)

Ex:- spring parent pom will include hundreads of dependencies as optional like jdbc starter jar will declare optional jars such as – mysql,oracle,postgresql ….

When those are optional they wont come, but if downstream projects wants them they can declare in their pom

|  |  |
| --- | --- |
| To declare a dependency as optional in Maven, add the <optional>true</optional> tag within the dependency declaration in your POM file | <dependency>  <groupId>org.example</groupId>  <artifactId>optional-lib</artifactId>  <version>1.0.0</version>  <optional>true</optional>  </dependency> |

Scopes

There are 3 classpaths available ,

Compile, runtime, test classpath

Maven defines 6 scopes: **compile**, **runtime**, **provided**, **system**, **test**, and **import**. Maven defines the behavior for each scope as following (copied verbatim from the [dependency management](https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html) page)

* **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** (means those jars will be provided by servlet container or server- these jars are needed at compile time we don’t need to include them at runtime as server is already providing )

Dependencies are available during compile and test but are expected to be provided by the runtime environment (e.g., a servlet container).

So these dependencies need not to be present at runtime, because these are already given by server ,

so if u make that jar scope as provided then it will be present only at compile, test time not during runtime

so any jar if u feel server is already giving, then u can mark those jars as provided

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. This scope is only available on the compilation and test classpath, and is not transitive.

* **runtime** This scope indicates that the dependency is not required for compilation, but is required only at runtime. It is in the runtime and test classpaths, but not the compile classpath. (e.g., database drivers)

**ex:**- jpa is enough at compile time, but at run time we need mysql driver, so here we can mark mysql as runtime

* **test** Dependencies are only needed for compiling and executing tests (e.g., JUnit, Mockito).. This scope is not transitive.

**The test scoped dependency is available only for test compilation and test execution**

* **system** This scope is similar to **provided scope** 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. (rarely used)

**Ex:- some jars may not be provided by server, but in server linux vm we would have had these jars in some specific folder , while declaring we should tell where this jar is present**

we provide a path **to the dependency on that  machine instead of pulling it from a maven repository**

<dependency>

<groupId>com.sun</groupId>

<artifactId>tools</artifactId>

<version>1.8</version>

<scope>system</scope>

<systemPath>${java.home}/../lib/tools.jar</systemPath>

</dependency>

* **import** (only available in Maven 2.0.9 or later) this scope is to import all the dependencies declared in their pom

This scope is only supported on a dependency of type pom in the section. It indicates the dependency to be replaced with the effective list of dependencies in the specified POM's section. Since they are replaced, dependencies with a scope of import do not actually participate in limiting the transitivity of a dependency.

 This scope is specifically used within the <dependencyManagement> section of a POM file.

 Essentially, it imports the <dependencyManagement> section of the specified POM, enabling you to reuse dependency versions and configurations

|  |  |
| --- | --- |
| <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-dependencies</artifactId>  <version>2.7.17</version>  <type>pom</type>  <scope>import</scope>  </dependency> | this import scope means all the dependencies  It allows you to import dependency configurations from other POM files.  ex:- spring people would have declared all the best suited dependencies In their pom file  if we use <scope>import</scope> then all the dep declared in that pom also will be downloaded to us   By importing a POM, you can centralize dependency version management, ensuring consistency across your project.   This is very useful for things like Spring Boot dependency management, where spring provides a pom file that contains all of the versioning information. |

• Dependencies are managed by the Maven Dependency Plugin

• Important Goals:

• dependency:tree - shows the dependency tree. Useful for resolving conflicts ex:- mvn dependency:tree

• dependency:go-offline - Resolve all all, prepare to go offline

• dependency:purge-local-repository - Clear artifacts from local repository

• dependency:sources - get sources for all dependencies.

Maven multi module projects

|  |  |
| --- | --- |
| 1. Create a pom in parent folder and that parent project packaging type should be pom(not jar/ear/war) as below   Sample parent pom here  <groupId>com.bharath.product</groupId>  <artifactId>productparent</artifactId>  <version>1.0</version>  <packaging>**pom**</packaging> | 1. Add modules in parent pom – means parent should have child project names   and child also should have parent pom details  <modules>  <module>productservices</module>  <module>productweb</module>  </modules>  note:- if u build parent then all child projects also will be build in the above declared order  so to build parent- goto parent pom and type “mvn clean install” |
| 1. In every child pom mention parent pom details then only child will inherit the properties declared in the parent pom   <parent>  <groupId>com.bharath.product</groupId>  <artifactId>productparent</artifactId>  <version>1.0</version>  </parent>  <artifactId>productservices</artifactId>  <packaging>jar</packaging>  and here for child we didn’t mention the version & group id , (we mentioned only artifact id)  child will inherit the parent jar’s version, so productservices version also will be 1.0 same as parent | inter module dependency  if 1 module depends on another module – use that group id, artifact id ,version  **every child will have different set of dependencies,** u can see them here |

|  |  |  |
| --- | --- | --- |
| parent- here we have to tell modules to define childs | child 1 – here for child we should tell who is the parent – hence we defined parent details in <parent> tag | child 2  keep all the child projects inside parent proj folder as per image |
| <groupId>com.ampf</groupId> <artifactId>ParentProj</artifactId> <version>1.0-SNAPSHOT</version> <packaging>pom</packaging>  <name>ParentProj</name> <url>http://maven.apache.org</url>  **<modules>  <module>child1</module>  <module>child2</module> </modules>** | <parent>  <groupId>com.ampf</groupId>  <artifactId>ParentProj</artifactId>  <version>1.0-SNAPSHOT</version> </parent>  <artifactId>child1</artifactId> <version>1.0-SNAPSHOT</version> <packaging>jar</packaging>  <name>child1</name> | |  |  | | --- | --- | | <parent>  <groupId>com.ampf</groupId>  <artifactId>ParentProj</artifactId>  <version>1.0-SNAPSHOT</version> </parent>  <artifactId>child2</artifactId> <version>1.0-SNAPSHOT</version> <packaging>jar</packaging>   <name>child2</name> |  |   if child 2 wants some classes from child1 then add it as a dependency  child2/pom.xml – here in child2 pom we added child1 – now in child 2 we can access child 1 classes  <dependencies>  <dependency>  <groupId>com.ampf</groupId>  <artifactId>child1</artifactId>  <version>1.0-SNAPSHOT</version>  </dependency> |

Advantage of declaring parent in child pom

We can inherit the dependencies, plugins from parent pom – just define the dependencies, plugins in parent pom with versions,

In which ever child we want, we can just add the dependency/ plugin without version name – so that version number will be inherited

|  |  |  |  |
| --- | --- | --- | --- |
| advantages with <dependencyManagement> tag   1. Centralised dependency management   The parent POM's <dependencyManagement> section allows you to define versions for your project's dependencies. Child modules can then declare dependencies without specifying versions, ensuring consistency across the entire project. This prevents version conflicts and makes it easier to update dependencies in one place.   |  |  | | --- | --- | | sample parent pom declaring all dependencies in <dependencyManagement> tag  <groupId>com.ampf</groupId>  <artifactId>ParentProj</artifactId>  <version>1.0-SNAPSHOT</version>  <packaging>pom</packaging>   <name>ParentProj</name>  <url>http://maven.apache.org</url>   <modules>  <module>child1</module>  <module>child2</module>  </modules> <**dependencyManagement**>  <dependencies>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>3.8.1</version>  <scope>test</scope>  </dependency>  </dependencies> </**dependencyManagement**>  By default dep declared In that <**dependencyManagement**>/ plugins declared in <pluginManagement>  tag will not be inherited to all child’s, which ever child wants, that has to declare this dep again in that child pom, without need to specify version number  sample child pom  <parent>  <groupId>com.ampf</groupId>  <artifactId>ParentProj</artifactId>  <version>1.0-SNAPSHOT</version> </parent>  <artifactId>child2</artifactId> <version>1.0-SNAPSHOT</version> <packaging>jar</packaging> <name>child2</name>  <dependencies>  *<!-- here we used junit dep which is already declared in parent  without need of mentioning version -->*  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <scope>test</scope>  </dependency> | here spring people declared 464 dependencies in “spring-boot-dependencies” pom , since we are defining this as our parent pom, all these 464 dep are available to us, if we want any of these depe, we just need to include dep without mentioning the version in child pom  lly, they defined all the plugins, which ever we want we can just include without mentioning the version  why spring people didn’t kept in <dependencies> tag,  if they kept in that tag, then all childs will get all 464 dependencies |  * See here in child pom we are not mentioning the version this is the advantage   advantage is – all child will have same version which is declared in parent  so even when we want to update the version in all child poms, instead of updating in all childs, just we can update in parent alone | 1. **Consistent Plugin Configuration:**   Similar to dependencies, the <build> and <pluginManagement> sections in the parent POM allow you to configure Maven plugins and their versions consistently across all child modules. This ensures that all modules use the same build processes and plugin versions, leading to more predictable builds  <build>  <pluginManagement>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <version>3.12.1</version>  <configuration>  <source>17</source>  <target>17</target>  </configuration>  </plugin>  </plugins>  </pluginManagement>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-surefire-plugin</artifactId>  </plugin>  </plugins>  </build>  use like below In child poms  <build>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  </plugin>  </plugins>  </build>  so here in child pom we are included plugin without version |
| default dependecies inheritance  <dependencies>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>3.8.1</version>  <scope>test</scope>  </dependency> </dependencies>  unlike declaring in <**dependencyManagement**>  when u declare dependencies in <dependencies> tag , then all the dependencies wil be inherited to all childs by default |  |

 **Simplified Child POMs:** By inheriting configuration from the parent, child POMs become much leaner and easier to read and manage. They only need to define the specific configurations and dependencies unique to that particular module. This reduces redundancy and clutter.

 **Project-Wide Settings:** You can define project-wide settings in the parent POM, such as repository locations, build profiles, and reporting configurations. These settings will automatically be applied to all child modules, ensuring consistency in the project's infrastructure.

That’s why spring people defined 464 dependencies and many plugins in parent pom </**dependencyManagement**> and <pluginManagement> tags, so that which ever child wants any dep, they can just mention artifact id without need to specify the tag

 **Improved Maintainability:** When you need to update a dependency version or plugin configuration, you only need to make the change in the parent POM. This change will automatically propagate to all child modules, saving you time and effort and reducing the risk of errors.

 **Clear Project Structure:** Defining a parent POM clearly establishes the hierarchical relationship between the modules in your project, making it easier to understand the project's organization and dependencies.

Renaming EJB module

//generally all jars will be included inside an EAR

Before including an jar inside an EAR, that jar will be renamed inside ear and kept inside an EAR file, instead of keeping the jar with original name inside an EAR

To customize module filenames in a Maven EAR project, you need to modify your pom.xml file. Within the <configuration> section of the maven-ear-plugin, add a <modules> section. For each module (e.g., <ejbModule>), specify the <groupId>, <artifactId>, and then use the <bundleFileName> tag to set the desired new filename (including the extension).

For example, to rename a module to anotherName-1.2.3.jar, you would configure it like this within your pom.xml:

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-ear-plugin</artifactId>

<configuration>

<modules>

<ejbModule>

<groupId>com.example</groupId>

<artifactId>my-ejb</artifactId>

<bundleFileName>anotherName-1.2.3.jar</bundleFileName> //this means an existing jar will be renamed and kept inside an EAR

</ejbModule>

</modules>

</configuration>

</plugin>