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> |
| project uses plugin to generate java stubs | <https://github.com/springframeworkguru/mb2g-mapstruct> |
| parent & child pom – inheritance demo | <https://github.com/springframeworkguru/mb2g-mm-maven/tree/flatten-plugin> |
| spring maven multi mod proj demo | <https://github.com/springframeworkguru/mb2g-spring-boot-mm/> |
|  |  |

Important points &

|  |  |
| --- | --- |
| **including dep**   1. 464 dep are already defined in spring boot paren pom, so next time if u want any dep, instead of copying artifact name from maven central, copy it from spring boot parent pom |  |
| **running the junits**   1. When u run verify phase “mvn verify” then automatically all the integration tests will be executed   because before verify phase we have phases like – test, integration test phases and all so..   1. Or u can run “mvn test”   best way “mvn verify” is because u can configure “spring-boot-maven-plugin“ to execute at pre,post integration phase | **Building the app**  Since clean is a diff life cycle and as it doesnot belongs to default life cycle when u run mvn install then project will not be cleaned  Hence we have to specifically add clean phase if we want to clean the project “**mvn clean install**” |
| to generate a pom with values instead of placeholders (${version}) then use a plugin called  <artifactId>flatten-maven-plugin</artifactId>  the generated pom will be kept inside jar/META-INF | **Running the spring boot app**  here after u can start spring boot app with  mvn clean install spring-boot:run  no need to depend on intellij as this spring-boot plugin can run the jar  this install will create the jar and keeps that jar in local repo |
| in intellij **after adding any dep** in pom we have to do below,   1. Execute command – mvn clean install 2. Click maven reload – only then this the dep will be added to current proj – so always do reload 4. If u are working with any plugin which should generate classes –it will generate but if intellij is not picking this click 2nd option “Generate sources” |  |

Best practices

While creating a multi module project, better to define BOM (means declare all the dependencies with versions in <dependencyManagement> tag)

So that any child, if they want that dep, they will just mention group id ,artifact id without need to mention the version

Setting to single java version for entire proj

|  |  |
| --- | --- |
| define java version in parent pom  use the same variable in all childs  so that all parent proj, child module all will be using same java version  similarly setting intellij proj version also to same version | parent pom properties  <properties>  <java.version>11</java.version>  <revision>0.0.1-SNAPSHOT</revision>  <maven.compiler.source>${java.version}</maven.compiler.source>  <maven.compiler.target>${java.version}</maven.compiler.target> </properties>  Child pom properties – here use same java version  <properties>  <maven.compiler.source>${java.version}</maven.compiler.source>  <maven.compiler.target>${java.version}</maven.compiler.target>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <spring-boot.repackage.skip>true</spring-boot.repackage.skip> </properties> |
| <properties>  <java.version>11</java.version>  <maven.compiler.source>${java.version}</maven.compiler.source>  <maven.compiler.target>${java.version}</maven.compiler.target>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <spring-booxt.repackage.skip>true</spring-boot.repackage.skip> </properties> | note only with these properties current build path will be changed from java 17 to 11    here if u want intellij or if u want ur maven to use any specific java version then we have to set specifically |

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 |

Alternate jvm languages

Most alternate jvm languages (groovy, kotlin, scala) can be compiled into java byte code with maven

Typically done via plugins hooking into compile phase

Ex:- in gradle we will write gradle code In groovy language, even if u write in groovy language finally it wil be compiled into java byte code only

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, and PG- 1 plugin will have lot of goals

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 (**default build lifecycle consists of 23 phases**)
* clean: to clean the project and remove all files generated by the previous build – means it will delte the target folder
* site: to create the project's site documentation -

**default 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)

**Why we have to mention clean phase ,why this phase wont execute automatically on mvn install ?**

|  |  |
| --- | --- |
| 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.  ex2:- Since verify phase it is at last, when u run verify it will run integration tests also  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=1 person  1 person = many tasks /goals | 1 person will have many goals when he is growing up |

|  |  |  |
| --- | --- | --- |
| to gen the stubs from wsdl file | to gen the stubs from xsd file | to gen the stubs from json schema file |
| *use “****cxf-codegen-plugin****” plugin* | <plugin>  <groupId>org.jvnet.jaxb2.maven2</groupId>  <artifactId>maven-jaxb2-plugin</artifactId> | <plugin>  <groupId>org.jsonschema2pojo</groupId>  <artifactId>jsonschema2pojo-maven-plugin</artifactId> |

|  |  |  |  |
| --- | --- | --- | --- |
| 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  mvn <plugin name> :<goal name>  ex:- mvn clean:clean  mvn install:install | |
| 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>  means here this clean plugin will be executed during maven default initialize phase itself | **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). |

Maven compiler plugin(cfg java version)

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

|  |  |  |
| --- | --- | --- |
| 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 | <properties>  <revision>0.0.1-SNAPSHOT</revision>  <java.version>11</java.version>  <maven.compiler.source>${java.version}</maven.compiler.source>  <maven.compiler.target>${java.version}</maven.compiler.target>  </properties> |
|  | <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <version>3.8.1</version>  <configuration>  <source>21</source>  <target>21</target>  <compilerArgs>--enable-preview</compilerArgs>  </configuration> </plugin> | if u want to use the preivew features, then we have to add this flag  <compilerArgs>--enable-preview</compilerArgs> |

|  |  |
| --- | --- |
| <properties>  <java.version>11</java.version>  <maven.compiler.source>${java.version}</maven.compiler.source>  <maven.compiler.target>${java.version}</maven.compiler.target>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <spring-booxt.repackage.skip>true</spring-boot.repackage.skip> </properties> | note only with these properties current build path will be changed from java 17 to 11    here if u want intellij or if u want ur maven to use any specific java version then we have to set specifically |

|  |  |  |  |
| --- | --- | --- | --- |
| 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 all our junit 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 |

Fat jar

|  |  |
| --- | --- |
| note- spring boot maven plugin is the one that is creating fat jar/deployable jar | we cant directly run source code because , our source code is dep on many spring libraries,  hence if u want to run our source code, we should pack our source code along with dep libraries  note:- here this manifest file will be created by “spring-boot-maven-plugin” |
| Bec of spring-boot-maven plugin- if we just annotate the class with @SpringBootApplication that class name will be considered as main class  & that class name will be auto added to manifest.mf file | |

By default spring boot creates 2 jars –

**Fat jar** – means a jar will be created with all the libraries/dependent jars like spring,jpa,hib jars +our source code – hence it is 29 MB (this fat jar is called deployable jar/uber jar)

This fat jar is an executable jar for spring boot (this jar has a manifest file which has a main class name) - “java –jar <jar file name>” will run that jar by seeing the manifest file content

Since our source code is dep on spring lib & To make the jar as executable jar then along with source code we must need all dep jar like spring, hib then that exec jar is made like fat jar ()- only that jar is runnable – we cant run our source code simply with out libraries

**Original jar** contains only our source code hence it is only 4KB

|  |  |  |  |
| --- | --- | --- | --- |
| 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” |
| by default this plugin will look files only that are present in src/main/resources  it will look only for direct files, if u keep that xsd file inside some abcd folder inside “src/main/resources/abcd/jax.xsd” then that xsd file will be skipped | these jars are needed because, the generated stub classes must be annotated with some anno & these anno belongs to these jars only |  |

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>    this file should be kept in src/main/resources | 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 | here dependencies are needed because generated classes must be annotated with anno from these jars only |  |

Map struct generate impl class

* Here compiler plugin during the compile time itself scans for the interfaces annotated with @Mapper
* it will generate an Impl class with all methods that are declared in that interface which has the logic to map the data from 1 bean to other bean

Those method’s contains the logic to map the data of 1 bean to another bean if both classes have same getters and setter variable names

|  |  |  |  |
| --- | --- | --- | --- |
| <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <version>3.8.0</version>  <configuration>  <annotationProcessorPaths>  <path>  <groupId>org.mapstruct</groupId>  <artifactId>mapstruct-processor</artifactId>  <version>${mapstruct.version}</version>  </path>  </annotationProcessorPaths>  </configuration>  </plugin> | | import org.mapstruct.Mapper;  @Mapper public interface *StudentMapper* {   Student getStudentFromProfessor(Student *employee*);  Professor getProfessorFromStudent(Professor *professor* ); }    note:- alteast 1 class must have all getters/ setters instead of lombok @Data |  |
|  |  | |  |

Maven-enforcer-plugin

|  |  |
| --- | --- |
| this plugin will enforce by checking the pre-requisites  here since we mentioned java 11, maven 3.5 if we didn’t meet those min versions then build will fail | <plugin>  <artifactId>maven-enforcer-plugin</artifactId>  <executions>  <execution>  <id>enforce-env-requirements</id>  <goals>  <goal>enforce</goal>  </goals>  <configuration>  <rules>  <requireJavaVersion>  <version>[11,)</version>  </requireJavaVersion>  <requireMavenVersion>  <version>[3.5,)</version>  </requireMavenVersion>  </rules>  </configuration>  </execution>  </executions>  </plugin> |

flatten-maven-plugin

the answer is written in next pages

to generate a pom with values instead of placeholders (${version}) then use a plugin called

<artifactId>flatten-maven-plugin</artifactId>

the generated pom will be kept inside jar/META-INF

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

Note after changing java version by editing in systm variables

1. u must restart the windows command prompt
2. u must restart intellij idea if u want to run any maven command- if u don’t re-start intellij cmnd prompt will still point to old java version

|  |  |
| --- | --- |
| 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 |
| you can execute both phase and plugin in single command | mvn <phase> <plugin:goal>  mvn clean spring-boot:run  mvn clean install spring-boot:run |
| easy way to start spr boot app- without exec main class | mvn <plugin name>:<goal>  mvn spring-boot:run  Note:- this will internally takes system java home as base jdk so because with that  this command wont take maven pom.xml jdk version |

Maven variables

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

|  |  |
| --- | --- |
| what is ${project}  its like “this” keyword in java  ${project} refers to current pom  ex:-  <parent>  <artifactId>mb2g-mm-maven-fixme</artifactId>  <groupId>guru.springframework</groupId>  <version>1.0-SNAPSHOT</version> *<!-- <version>${revision}</version>-->* </parent>   <artifactId>web-app</artifactId>   <dependencies>  <dependency>  <groupId>guru.springframework</groupId>  <artifactId>jpa-entities</artifactId>  <version>${project.version}</version> //this will use above 1.0-SNAPSHOT as its using current project version  </dependency> | In a Maven pom.xml file, ${project} is another **predefined property**, but it represents the **entire Maven Project Object Model (POM)** as an object.  to access all the information defined within your pom.xml file programmatically within the POM itself. You can then use dot notation to access specific elements and their values within this project object.  how it relates to other properties like ${project.version}:   * ${project.version} is actually accessing the version field of the project object. Similarly:   + ${project.groupId} would access the <groupId> element.   + ${project.artifactId} would access the <artifactId> element   + ${project.name} would access the <name> element (if defined).   + ${project.build.directory} would access the <build><directory> element.   + And so on...   **In essence, ${project} is the root object that holds all the project-related information defined in your POM.**  **Why use it?**   * **Self-referencing:** It allows the POM to refer to its own defined values   Note:- instead of using current version , better declare all those variables in parent pom & simply use those variables in child pom using “${version}” |
| Using current proj variables with ${project} | <artifactId>web-app</artifactId>  <properties>  <maven.compiler.source>11</maven.compiler.source>  <maven.compiler.target>11</maven.compiler.target>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding> </properties> <dependencies>  <dependency>  <groupId>${project.groupId}</groupId> //see here we are using current proj group id  <artifactId>web-model</artifactId>  <version>${project.version}</version>  </dependency> </dependencies>   * See here we are using then and there itself   <properties>  <revision>0.0.1-SNAPSHOT</revision>  <java.version>11</java.version>  <maven.compiler.source>${java.version}</maven.compiler.source>  <maven.compiler.target>${java.version}</maven.compiler.target>  </properties> |
| <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 |
| definining and fetching from same pom.xml – just use ${} | defining in same pom  <properties>  <java.version>11</java.version>  <mani.custom.property>ayurvedic jar</mani.custom.property>  </properties>  using in same pom  <additionalProperties>  <projecter.javaversion>${java.version}</projecter.javaversion>  here we defined in properties tag  and we are using with >${java.version} in same pom |

Defining variables in parent and using those variables in child

The variables defined in parent pom <properties> tag will be inherited to childs & we can directly use those variables using $ symbol

This variable inheritance is only available since 3.5

|  |  |
| --- | --- |
| properties defined in parent pom | entries in child pom – to demo inheritance |
| <properties>  <parentPomVersion> 2.0-SNAPSHOT</parentPomVersion>  <child2Version> 2.0-SNAPSHOT</child2Version>  </properties> | <groupId>com.ampf</groupId>  <artifactId>child3Hib</artifactId>  <version>${parentPomVersion}</version>  // Now in child 3 we used a variable defined in parent poms <properties> tag |
|  | Demo for using variables defined in current pom with ${project} keyword  <groupId>com.ampf</groupId> <artifactId>child3Hib</artifactId> <version>${parentPomVersion}</version>  <dependency>  <groupId>com.ampf</groupId>  <artifactId>child2</artifactId>  <version>${project.version}</version> // here this uses the version of current poms version tags value /version </dependency> |

Problem of creating jar with placeholder poms

If u use place holder (<version>${project.version}</version>) in parent pom/ in any pom then even the same place holders will be present In the pom present in final jar

|  |  |
| --- | --- |
| sample generated pom | this generated pom will have placeholders  generated pom will be in below location inside jar  “child2-2.0-SNAPSHOT.jar\META-INF\maven\com.ampf\child2” |
| So to avoid above problem, use a plugin called | |  |  | | --- | --- | | <build>  <plugins>  <plugin>  <groupId>org.codehaus.mojo</groupId>  <artifactId>flatten-maven-plugin</artifactId>  <configuration>  <flattenMode>bom</flattenMode>  </configuration>  <executions>  <!-- enable flattening -->  <execution>  <id>flatten</id>  <phase>process-resources</phase>  <goals>  <goal>flatten</goal>  </goals>  </execution>  <!-- ensure proper cleanup -->  <execution>  <id>flatten.clean</id>  <phase>clean</phase>  <goals>  <goal>clean</goal>  </goals>  </execution>  </executions>  </plugin>  </plugins>  </build> | when u add this plugin in parent pom or child pom  then the generated pom inside jars/META-INF folder will not have the place holders like this    along with parent pom, add this plugin in child pom  & this plugin must be run in <phase>process-resources | |

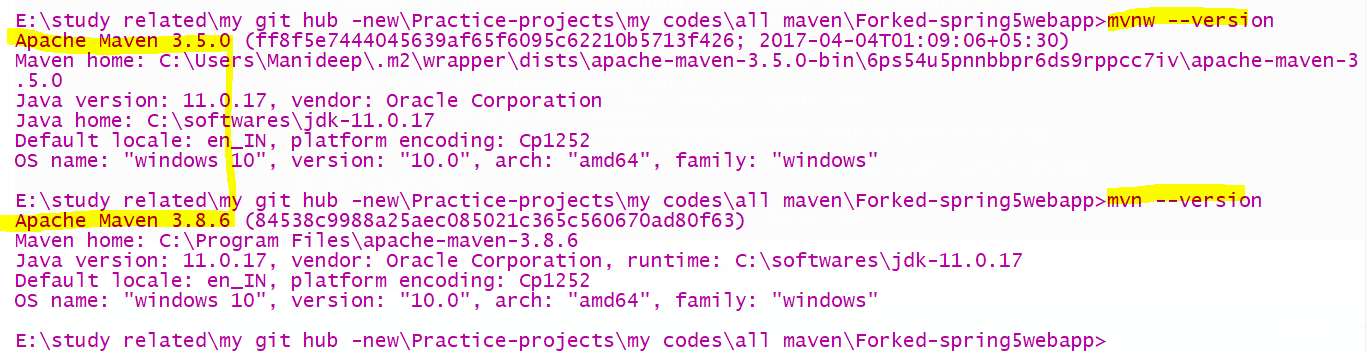
Maven wrapper

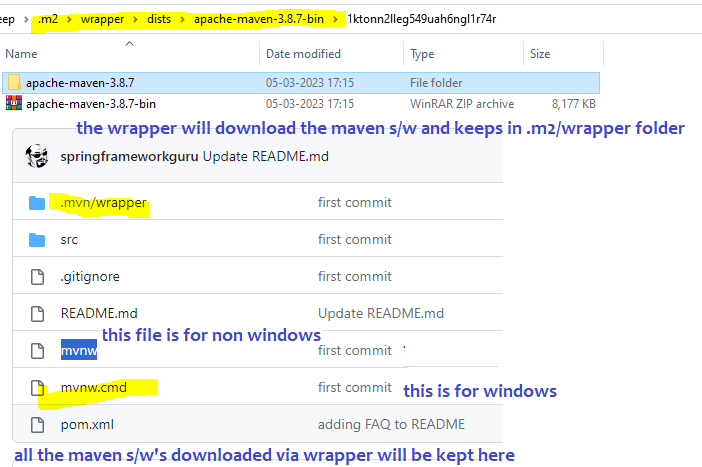
Mvnw == mavenwrapper

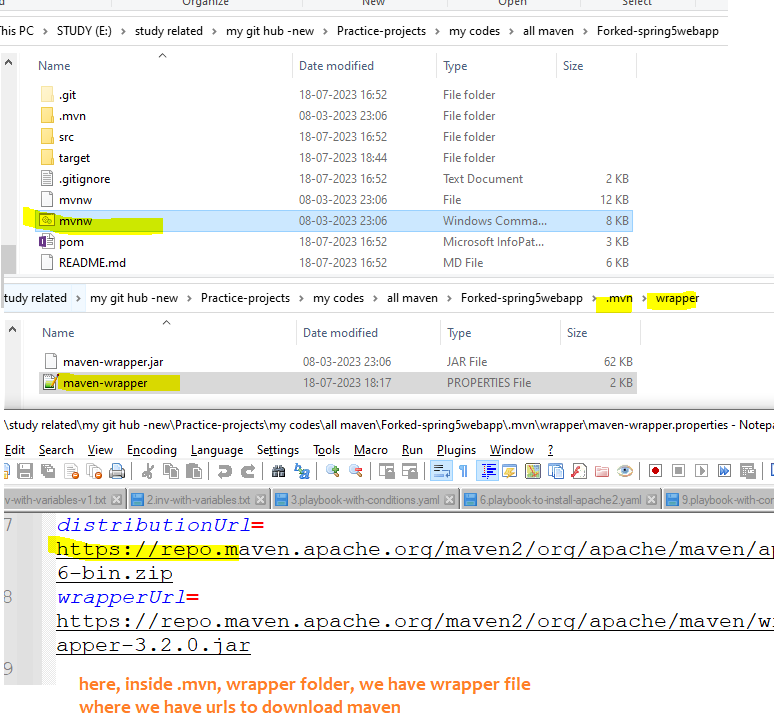
Maven wrapper is just a wrapper around maven software, this wrapper allows us to work with maven without having maven installed.

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””

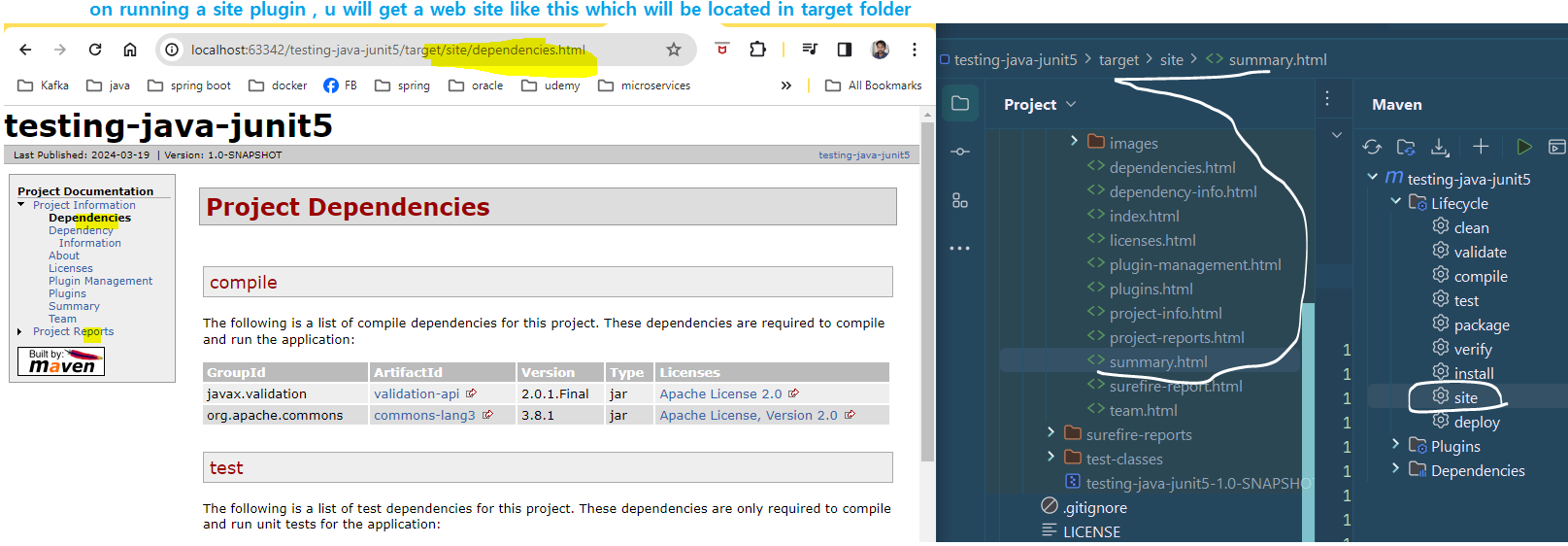
|  |  |
| --- | --- |
| Sure fire plugin | To run unit test |
| Sure fire plugin with tags | <build>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-surefire-plugin</artifactId>  <version>2.22.0</version>  <configuration>  <argLine>  --illegal-access=permit  </argLine>  <!--<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> |
| how to pick right dep version in realtime?  pick the versions which are released and near same time  for spring-boot-starter-parent 3.2.2 (released in jan 2024) select  cxf-spring-boot-starter-jaxws – 4.0.4 which is also released in jan 2024  cxf-codegen-plugin 4.0.4 released in *Mar 12, 2024*  *so use all dep and plugins which were released in similar time* |  |

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

Note:- 1 child cant see other child dependencies (like in real world 1 child cant see other child assets) ,child can inherit parent jars

Always its better to create new module using intellij,

All the common properties can be kept In parent project (ex:- if all childs wants to use java 8 then keep this prop in parent, if all childs wants to use same compiler plugin then keep that plugin in parent)

|  |  |
| --- | --- |
| <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <log4j.version>2.17.1</log4j.version>  <maven.compiler.source>8</maven.compiler.source>  <maven.compiler.target>8</maven.compiler.target>  </properties> | bote:- if u define all these properties in parent  then all these variables will be inherited to all childs, we can just use ${}  <maven.compiler.source>${java.version}</maven.compiler.source> <maven.compiler.target>${java.version}</maven.compiler.target> |

Note:- keep only common dep in parent module (Generally no dep will be in parent module- bec no lib will be common for all modules)

|  |  |
| --- | --- |
| Note each module is effectvely a maven project & each parent module can have child projects/modules  Note each child can even be parent and that can have another childs  usually parent project is of pom type and children will be of jar/war/ear type | 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>  advantages of declaring childs in parent pom   1. If u clean parent, then all childs target folder will be deleted   where to see the dependencies of parent    in parent pom, if u declare any dependencies outside of </dependencyManagement>  then only those are considered and depe and those dep will be included for all childs by default  sample parent pom |

‘

|  |  |
| --- | --- |
| Maven Reactor’ is the one like team lead who will build all the child sub modules  • The Reactor is the what builds each module of the Maven project  • The Reactor collects the modules to build  • The Reactor will then run selected build lifecycle against each module  • The Reactor will determine the build order of the modules  • By default The Reactor will build modules sequentially  • Optionally can use threads to build modules in parallel | How to create a new child module In intellij   1. Just right click on parent module🡪 New🡪module   [INFO] Reactor Build Order:  [INFO]  [INFO] ParentProj [pom]  [INFO] child1 [jar]  [INFO] child2 [jar]  [INFO] child3Hib [jar]  as per above , we can see reactor is the one who builds all the child modules |

Factors determining the build order used by Reactor

|  |  |
| --- | --- |
| 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:- bec of above module declaration only, if u build parent then all child projects /sub modules also will be build   * By default it will be build in the above declared order * But if each module is dependent on other, then re-usable dep will be built 1st (modules used by other modules in the project will get built first)   so to build parent- goto parent pom and type “mvn clean install” | ex:- in below code, even though I placed child3Hib as 1st, since child2 is the dependent module, reactor built child2 first than child3hib  <modules>  <module>child3Hib</module>  <module>child1</module>  <module>child2</module> </modules>  Reactor Build Order:  [INFO]  [INFO] ParentProj [pom]  [INFO] child1 [jar]  [INFO] child2 [jar]  [INFO] child3Hib [jar]  [INFO] |
| 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    if child module is having dependecies (either its own or inherited from parent)  then we can see that module specific dependecies in this tab  ex:- if this child /any sub module/even parent if they don’t have any dependecy those this tab itself will be hidden |

Over modules creation

Don’t just simply create a new module , / Do not over-use modules

because each module is a separate maven project and all lifecycles must be executed , Don’t just create a module by assuming some one will use in future

Too many modules == lot of build time == too many jar creations,

Worst approach 🡪 One class or interface in a module

• Many small modules which could be combined in real time

If someone need it in future we can create a module by refactoring and extracting that code

When to create a submodule

**Reusability Across Projects:**

* If a specific part of your application (like the domain model or a utility library) might be reused in other future projects, isolating it in its own module makes it easy to include as a dependency in those projects.

**. Deployment Considerations:**

* In some deployment scenarios (e.g., microservices), you might want to deploy different parts of your application independently. Separating them into modules is the first step towards achieving this

|  |  |  |
| --- | --- | --- |
| 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 using tags (<dependencyManagement>,<dependencies> ) – 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.  All these dep in this depMgmt tag will not be directly inheriting to all childs , those will be available to all childs only if we re-declare in child  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 | **Example pom in spring boot**  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.4.0</version>  <relativePath/> <!-- lookup parent from repository -->  </parent>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web-services</artifactId>  </dependency>  <dependency>  <groupId>org.apache.cxf</groupId>  <artifactId>cxf-spring-boot-starter-jaxws</artifactId>  <version>3.4.0</version>  </dependency>  </dependencies>  here after declaring parent in pom, while defining dependencies we didn’t mention jar versions as those are already defined in parent pom <**dependencyManagement**> tag |

If u use multimodule project, childs wil inherit parent dependencies, and if u clean parent then all childs will be cleaned , if u build parent, then all childs will be build

 **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>

Spring boot related

Bill of materials

BOM means dependencies declared within the dependencyManagement section of the POM

• Fully qualified dependencies are listed under the dependencyManagement section of the POM

• Typically used to standardized versions

|  |  |
| --- | --- |
| Dependencies declared under the dependencies section of the POM will inherit the  version / packaging information from dependencyManagement | ex:- in parent pom we define all the dependencies under <dependencyManagement> tag  by default dep declared in parent pom <dependencyManagement> tag  will not be inherited to child  we can inherit only version info  so while in child, we just need to mention groupid, artifact id ,we don’t need to mention version as the version, package info will be inherited from <dm> tag |

|  |  |
| --- | --- |
| defining dependencies in parent pom  <dependencyManagement>  <dependencies>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>3.8.1</version>  <scope>test</scope>  </dependency>  </dependencies> </dependencyManagement> | inheriting version from parent to child  <dependencies>  *<!-- here we used junit dep which is already declared in parent pom dm tag  without need of mentioning version as version info will be inherited from parent-->* <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <scope>test</scope>  </dependency> |

Any dependencies declared in the dependencyManagment section of the POM DO NOT become transitive dependencies for the artifact

• Dependencies declared under the <dependencies> section of the POM DO become transitive dependencies for the artifact

• Dependencies declared under the <dependencies> section will inherit attributes such as version from dependencyManagement

Spring boot maven plugin

|  |  |  |
| --- | --- | --- |
| to run spr boot application   * 1. here after u can start spring boot app with   **mvn clean install spring-boot:run**  no need to depend on intellij as this spring-boot plugin can run the jar  this install will create the jar and keeps that jar in local repo | here spring boot is a plugin & it have lot of goals  mvn <plugin:goal name>  mvn spring-boot:run | you can execute both phase and plugin in single command  mvn <phase> <phase> <plugin name:goal>  mvn clean install spring-boot:run |

|  |  |
| --- | --- |
| **spring-boot:repackage** | this repackage goal is the one which will creates the deployable jar /fat jar  this needs the main class name- so that it will add the main class name in manifest file  which is mandatory to create the self deployable jar  This is most useful , even in core java project if u want to create a fat jar  then add this plugin and executea repackage goal  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId> </plugin>  and run this command  mvn spring-boot:repackage    so this command will create me both the fat jar and normal jar |

|  |  |
| --- | --- |
| easy way to start spr boot app- with mvn command  & without exec main class & without using intellij | mvn <plugin name>:<goal>  mvn spring-boot:run |

Configure the plugin to start the container before running any integration test and stop the container only after all test cases are executed

|  |
| --- |
| 1st of all integration test cases are diff and normal test cases which annotated with below are different  @ExtendWith(SpringExtension.class) @ContextConfiguration(classes = Soap1JaxbSprDemoApplication.class) public class SOAPControllerTest { |

|  |  |
| --- | --- |
| **Integration test –**  these will require container to be readily available before execution starts – for these test cases full container will be started with all the beans present in the project, hence for these integration test cases when spring container starts we can see banner | junit test cases – here bec of @ContextConfiguration(classes = Soap1JaxbSprDemoApplication.class) anno container will be started with limited beans- fully container will not be started – only required beans will be created  hence spring banner will not be visible when we run these kind of test cases |
| @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM\_PORT)  public class GreetingIntegrationTest {  @LocalServerPort  private int port;  @Autowired  private TestRestTemplate restTemplate;  @Test  public void testGreetingEndpoint() {  String url = "http://localhost:" + port + "/greeting";  ResponseEntity<Greeting> response = restTemplate.getForEntity(url, Greeting.class);  assertEquals(HttpStatus.OK, response.getStatusCode());  assertNotNull(response.getBody());  assertEquals("Hello, World!", response.getBody().getMessage());  } | @ExtendWith(SpringExtension.class) @ContextConfiguration(classes = Soap1JaxbSprDemoApplication.class) public class SOAPControllerCopyTest {   @Autowired  StudentSOAPController **studentSOAPController**;   @Test  public void test1(){  System.***out***.println("Testing class-->"+this.getClass().getSimpleName());  System.***out***.println(**studentSOAPController**.hashCode());  } |
|  |  |

|  |  |
| --- | --- |
| <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  <executions>  <execution>  <id>pre-integration-test</id>  <goals>  <goal>start</goal>  </goals>  </execution>  <execution>  <id>post-integration-test</id>  <goals>  <goal>stop</goal>  </goals>  </execution>  </executions>  </plugin>  </plugins>  </build> | <phase>integration-test</phase>  <phase>post-integration-test</phase>  these 2 are the phases of maven default life cycle which 23 phases  here we mentioned 2 phases , means this plugin will be executed in 2 phases of a default life cycle  when pre-integration-test phase arrived then this plugin will execute start goal – to start container  when post-integration-test phase arrived this plugin will be execute stop goal – to stop container  While you don't see the <phase> tag directly, the magic happens because Maven implicitly binds these <execution> blocks to specific phases of its lifecycle based on the plugin's configuration and the goal being executed.   The spring-boot-maven-plugin is configured to have its start goal typically bound to the pre-integration-test phase. This means that when Maven reaches the pre-integration-test phase, it will execute the start goal of the spring-boot-maven-plugin (as defined in your first <execution>).   Similarly, the stop goal is typically bound to the post-integration-test phase. When Maven reaches this phase, it will execute the stop goal (as defined in your second <execution>) |
| when pre-integration-test starts this plugin will be execute start goal – to start container as below  [INFO] --- spring-boot-maven-plugin:2.4.0:start (**pre-integration-test**) @ SOAP-2-StubsGenFromWSDLfile ---  [INFO] Attaching agents: []  . \_\_\_\_ \_ \_\_ \_ \_  /\\ / \_\_\_'\_ \_\_ \_ \_(\_)\_ \_\_ \_\_ \_ \ \ \ \  ( ( )\\_\_\_ | '\_ | '\_| | '\_ \/ \_` | \ \ \ \  \\/ \_\_\_)| |\_)| | | | | || (\_| | ) ) ) )  ' |\_\_\_\_| .\_\_|\_| |\_|\_| |\_\\_\_, | / / / /  =========|\_|==============|\_\_\_/=/\_/\_/\_/  :: Spring Boot :: (v2.4.0)  if u see this banner then only officially container is started | when post-integration-test phase triggers then this plugin will execute stop goal – to stop container  2025-05-08 13:24:00.370 INFO 15088 --- [ main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''  2025-05-08 13:24:00.410 INFO 15088 --- [ main] com.ampf.Soap1JaxbSprDemoApplication : Started Soap1JaxbSprDemoApplication in 7.616 seconds (JVM running for 9.669)  [INFO]  [INFO] --- spring-boot-maven-plugin:2.4.0:stop (**post-integration-test**) @ SOAP-2-StubsGenFromWSDLfile ---  [INFO] Stopping application...  2025-05-08 13:24:00.496 INFO 15088 --- [on(2)-127.0.0.1] inMXBeanRegistrar$SpringApplicationAdmin : Application shutdown requested.  2025-05-08 13:24:00.691 INFO 15088 --- [on(2)-127.0.0.1] o.s.s.concurrent.ThreadPoolTaskExecutor : Shutting down ExecutorService 'applicationTaskExecutor' |
| Hence this plugin is doing this job to start and stop the container during **pre-integration-test** & post-integration-test phases | |

|  |  |
| --- | --- |
| Note –**without declaring these phases** – pre,post integ tests also  while executing junit test cases  @ContextConfiguration(classes = Soap1JaxbSprDemoApplication.class) public class SOAPControllerCopyTest {  container will be starting only with limited beans( – not full container )before test cases exec and stopping only when all test cases are executed (this is the behaviour of junit test cases annotated with spring annotation )  where as the above configuration is for integration test not for junit test annotated with spr annot | Note while exec junit test cases – it will start the container but it wont print the banner |

Seeing All prop in actuator endpoint via generating build-info.properties file

Steps

1. actuator will show the all prop present in file named “build-info.properties”
2. “mvn spring-boot:**build-info**”this goal will generate the “build-info.properties”file (we have to configure this plugin to put what prop in the file)

|  |  |
| --- | --- |
| <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId> </plugin>   1. using spring-boot-maven-plugin generate a file called “build-info.properties”(this file will be present in META-INF folder)   to generate this prop file, execute a goal called “**build-info”** present in the plugin named **“spring-boot”**  mvn spring-boot:build-info   1. only when u exec this goal, this “**build-info.properties**” file will be generated (by default this file will not be generated) |  |
| 1. all the properties present in this file will be displayed in “/actuator/info”, once u hit this url actuator will pick the data from this file and shows to us     sample content  *#Properties #Mon May 12 13:11:55 IST 2025* build.version=0.0.1-SNAPSHOT build.group=com.ampf build.name=sprBootMvnPluginDemo build.artifact=sprBootMvnPluginDemo build.time=2025-05-12T07\:41\:55.848568400Z | now we are able to see all the props defined in the file |
| 1. customizing the spring-boot-maven plugin to show our custom properties   now if we are able to generate the with more properties all that file content will be displayed in above actuator  <properties>  <java.version>11</java.version>  <mani.custom.property>ayurvedic jar</mani.custom.property> </properties>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  <executions>  <execution>  <!-- here this means execute goal named build-info should  be executed during the compilation phase/build-info-goal phase-->  <id>build-info-goal</id>  <goals>  <goal>build-info</goal>  </goals>  <configuration>  <additionalProperties>  <projecter.javaversion>${java.version}</projecter.javaversion>  <projecter.customProperty>${mani.custom.property}</projecter.customProperty>  </additionalProperties>  </configuration>  </execution>  </executions>  </plugin>  here not just configuring the plugin - we only should tell when to run that plugin  here we are trying to execute that plugin during compile phase | now on mvn command “mvn spring-boot:build-info”    now all below file content have been displayed |

Plugin which Adds git info

1. add this plugin – this plugin is already present in parent pom <pluginManagement> tag, we just need to inherit that plugin into child pom
2. <plugin>  
    <groupId>pl.project13.maven</groupId>  
    <artifactId>git-commit-id-plugin</artifactId>
3. Add prop 🡪 management.info.git.mode=full

That plugin will generate a new file called (git.properties) and all the contents of this file will be displayed by “/actuator/info” tag

Creating thin jar

Thin jar means we should create a jar only with our source code – we should not include any library (if u include all the libraries then it is called fat jar)

|  |  |
| --- | --- |
| <spring-boot.repackage.skip>true</spring-boot.repackage.skip>  well, after keeping this prop only   1. Child module build is success 2. It then genereated a thin jar – see the size here (both are 4KB only) means it gen jar only with source code |  |

Sometimes child sub modules will un-necessarily attempts to create a fat jar – so to avoid that we should add this prop

[ERROR] Failed to execute goal org.springframework.boot:spring-boot-maven-plugin:2.1.1.RELEASE:repackage (repackage) on project web-app: Execution repackage of goal org.springframework.boot:spring-boot-maven-plugin:2.1.1.RELEASE:repackage failed: Unable to find main class -> [Help 1]

When child modules will create a far jar?

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

If u keep this plugin in parent pom, then all the childs by defaults attempts to create a fat jar- but all the child modules will not have the main class(as main class will only present in 1 module)so without main class fat jar creation is not possible –

and more over child modules should create only thin jars which is possible with above prop, only \*\* should create 1 fat jar

pending:- plugin will generate the stubs and it will keep in target folder, if u didn’t manually copy to src/main/java then will those still be part of jar???????????????