MAVEN

DRAWBACKS:-

We have many problems with project development like

1. We need to add set of jar files in each project, these jar files must also include dependencies
2. If we don’t create right project structure, the project will not be executed
3. We need to write scripts for any tasks

WHAT IS BUILD TOOL:-

Source code = the actual program/code written by programmer in human readable format ex: - .java files

Byte code = the binary file which is understood by machine language ex: - .class files

Build tool is a program that takes source code and compile and then finally creates an executable file

Source code (.java files) will be compiled then we get .class files. These .class files are packaged into jar/war files

.jar is like tar file in Linux, which is collection of .class files

.war is used for web applications & open source which is collection of jar files, html & xml files

.ear is licensed

Libraries =

Dependencies = the external libraries that a project uses in order to compile/build/test/run

Plugins = plugin is used to carry out a task. We have 2 types of plugins i.e.., build plugins (executed during build process and they should be configured in <build> in pom.xml) & reporting plugins. Some of the popular plugins are clean, compiler, surefire, jar, war, javadoc, antrun

Goals = phases have goals, goals in maven are like tasks. Ex:-compile can be a goal, install can be a goal. Each goal is bounded to each phase ex: - phase----------🡪plugin: goal, ex: - mvn [plugin-name]:[goal:name]

Compile phase is bounded to compile: compile, in package phase we have jar: jar similarly we have install: install in install phase

Instead of executing phase (#mvn compile) we can also execute goals (mvn compile: compile)

MAVEN INTRODUCTION:-

Maven is a powerful **build automation tool** which is basically used for java projects. (Building means taking your source code then compiling and archiving into single executable file)

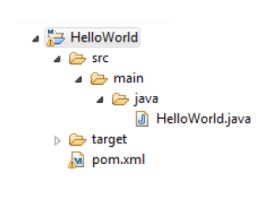
1. **Project management tool** = the name of the project, who works on it, what scm is configured, etc…
2. **Artifactory repository** = a centralized server where dependencies, build artifacts, plugins are stored

Maven helps in code building, dependency management, document creation, site publication

Maven will download jar, plugin dynamically

Maven contains lifecycle

Mavens supports 1700 directory structures



MAVEN WORK FLOW:-

We install & configure maven and select a directory structure. We configure pom.xml now when the code is done, we build the source code using maven. During that time maven will generate target folder which has .class file for all source code and download all dependencies, libraries, jar…. Finally we will get build artifact (war/jar/ear) as an outcome

MAVEN REPOSITORIES:-

A maven repository is a place where all the java related jars and plugins are maintained

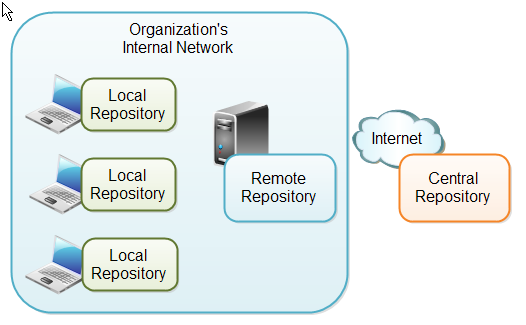
There are 3 types of maven repositories

1. Local repo = this repository is created whenever you run mvn command for the first time. The local repo present in local system of developer. maven by default downloads the libraries in (.m2) which is in users home directory

Ex: - c:\user\raju\.m2\repository

1. Public/Central repo = it is the centralized repository maintained by maven community. Maven knows the central repo location
2. Remote repo = A repository which is maintained within the organization (your own companies libraries) ex: - jfrog/nexus. You have to manually configure remote repository location in pom.xml

Maven first searches in local repository for dependencies, if not found then searches in central repository, if maven didn’t find then it searches in remote repository as how you mention in pom.xml

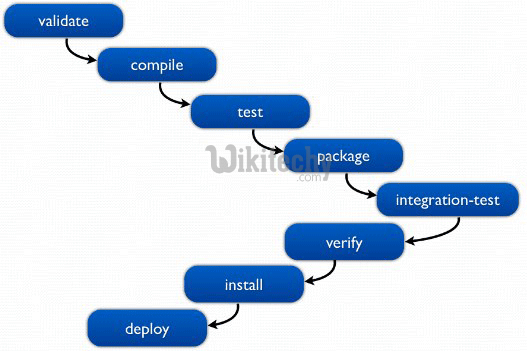


MAVEN BUILD LIFECYCLES:-

There are 3 built in lifecycles in maven. Maven build follows specific lifecycle to deploy and distribute the target project

1. default:-

Default is the main lifecycle in this default lifecycle. We have 6 phases in default life cycle



1. *Validate* = checks if pom.xml exists

We know that dependencies are not mentioned in source code. So we write all dependencies in pom.xml. When you run **#mvn validate** then it will check dependencies download it and store in /home/user/.m2/repository/Dependency

1. *Compile* = it will compile the source code

When you run **#mvn compile,** it will compile the source code and generate a .class file under target directory

Ex: - if there is add.java file corresponding to that there will be a add.class file will be generated

1. *Test* = tests the compiled source code using unit testing framework like junit

Ex: -for every add.java file there will be corresponding testadd.java file when test phase is executed, it will generate testadd.class file

1. *Package* = takes the compiled code and package into “jar/war/ear” as how you defined in pom.xml
2. *Verify* = run any checks to verify the package is valid and meets quality criteria
3. *Install* = install phase will push artifact(jar/ear/war) to your local repository (.m2)
4. *Deploy* = this phase will push artifacts from local repo(.m2) to remote repository

\*To use maven phases

#mvn phase (or) #mvn clean phase

\*To execute any mvn commands you need to be present in pom.xml location

1. Clean: - to clean the project and remove all files generated by previous builds. In clean lifecycle we have 3 phases
2. Pre-clean
3. Clean
4. Post-clean
5. Site: - to create fresh documentation to create reports, deploy site, etc…

MAVEN POM.XML:-

POM means Project Object Model. The pom.xml contains 4 sections

1. **project** = it is all about project information
2. ***model version*** = which version of pom.xml are you using
3. ***group id*** = grouping structure which is unique for each project
4. ***artifact id*** = name of the artifact ex:- helloworld.war
5. ***version*** = version of the artifact, it maintains version of each artifact
6. ***packaging*** = the outcome of the build weather jar/war/ear/ejb

<project>

<modelVersion>4.0.0</modelVersion>

<groupId>com.infosys.project1</groupId>

<artifactId>HelloWorld</artifactId>

<version>1.0</version>

<packaging>jar</packaging>

</project>

1. **dependencies** = in this section you can have n dependency which are used for application
2. ***group id*** = grouping structure of dependency found in their site
3. ***artifact id*** = name of the dependency
4. ***version*** = specific version of dependency

Ex: - junit\_1.0.jar

<dependencies>

<dependency>

<groupId>org.apache.junit</groupId>

<artifactId>junit</artifactId>

<version>1.0</version>

</dependency>

</dependencies>

1. **Build** = this allows you to customize the behavior of maven build. Here we add plugins.

* In this build section we have plugins section in which we add one or more plugin
* Each plugin section contains groupid, artifactid, version & executions
* Inside executions we have many execution
* Plugin is used to do some task

Ex: - there is apache plugin, with this we can start tomcat (this is one type of execution), stop tomcat (this is one type of execution) deploy file to other server (this one type of execution). We are doing all this execution using same plugin

mvn <goals>------->plugin--->tasks------>{this goal will call default plugin and that plugin will do some task}

Ex: - mvn compile ------->compile\_plugin-------->run------>plugin---------> {for compile goal maven uses default compile plugin after this maven will invoke run goal and it will call the plugin which is associated}

<!—Build Settings -->

<build>

<plugins>

<plugin>

#what is the plugin

<groupid>value</groupid>

<artifactid>value</artifactid>

<version>value</version>

<executions>

<execution>

#when to use the plugin

<phase>test</phase>

#what exactly the plugin has to do

<goals>

<goal>run</goal>

</goals>

</execution>

<execution>

</execution>

</executions>

</plugin>

<plugin>

#plugin2

</plugin>

</plugins>

</build>

* Maven has plugin called “exec” which runs any OS commands

Ex: - mvn package exec:exec

1. **distribution management** = the remote repository location need to be specified here/ where we download/push artifacts to
2. ***snapshot*** = partial completed jar files is called snapshot (or) it indicates that project is in development
3. ***release/final*** = official release of jar is called release/indicating that project is in production

In case of version maven downloads only once and never try to download newer version but in case of snapshot maven will automatically fetch the latest snapshot every time when a project is built

Ex: - there is a developer "Adam" who is working from January to June. "Adam" can officially release the code in June. Say there is other developer "john" who is working from February to September. Now imagine if john wants a jar file from Adam then he has to wait till June where john work will delayed. So john asks Adams partial jar and then works on it

\* In our remote repositories we maintain two types of repositories

In pom.xml under project you have something like this <version>1.0snapshot</version>. This tells maven to push it to snapshot repository. If nothing is mentioned in version then it is telling maven to push to release version

<repositories>

<repository>

<id>my-internal-site</id>

<name>sonarnexus</name>

<url>https://host:8081/repositories/releases</url>

<snapshots>

<enabled>false</enabled>

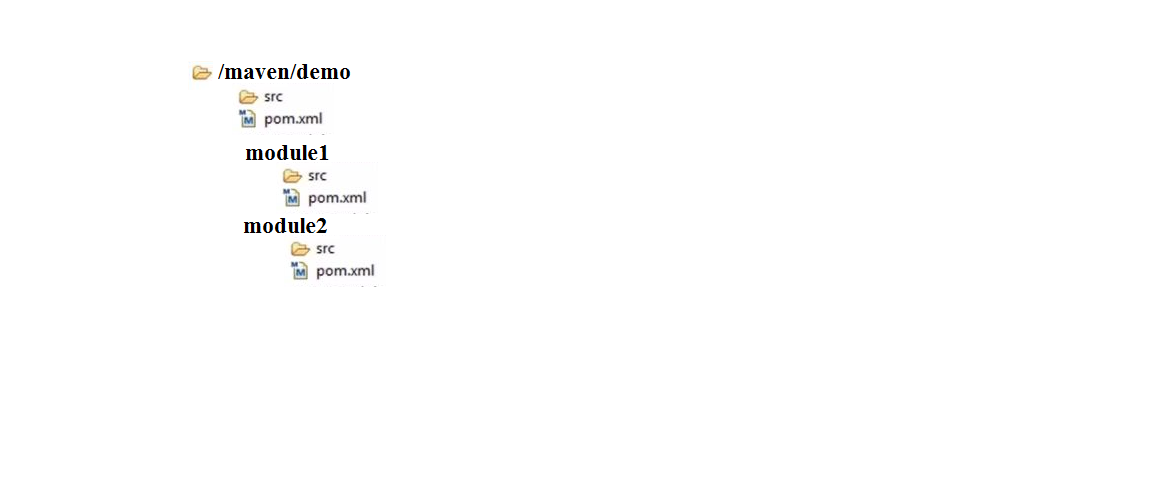
</snapshots>

</repository>

</repositories>

Enable = false means which means maven will not look for updates for this repository

1. **Multi module** =splitting complete project into smaller components, where each of them is treated as separate project(instead of writing all the things in single pom.xml file which looks messy we divide therefore)

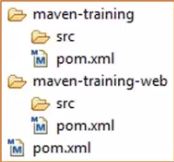
****

Generally we put all java files under src/main, but if there is a project with several components. Then each developer will be working on single component/module. Imagine if all java files of all components are put into src/main then

1. The developer cannot do build for his module alone
2. When there is a build failure it will fail all the modules

So we segregate this by creating each project for each module, thus we have many modules called as multi module.

* For each module we have separate folder structure and we mention all this module details in super pom.xml



* To run particular module alone

#mvn clean -pl ModuleName

Imagine if module2 is dependent on module1, then we add mod2 dependencies in in mod1 pom.xml under dependency section. So if there are several modules in a project, where in each module is dependent on each other, then we cannot run single modules individually. So we go for ***SUPER-POM*** (or)***PARENT-POM.*** The packaging type of parent pom.xml is “**pom**”

A parent pom.xml will pass the values to all child pom.xml

Ex: - go to parent pom.xml location and run compile, then parent will pass the value to child pom.xml of every module

Now understand the dependencies or the flow of modules which one is first which one is next according to that add the modules in parent pom.xml

<modules>

<module>mod1<module>

<module>mod2<module>

</modules>

* Also copy the parent “g,a,v” and add it into child pom.xml of every module under parent section

<parent>

<groupId>EBU</groupId>

<artifactId>ParentModule</artifactId>

<version>1.0-SNAPSHOT</version>

</parent>

<groupId>EBU</groupId>

<artifactId>ChildtModule</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>jar</packaging>

MAVEN SETTINGS.XML:-

This setting.xml file provides configuration for either single user [${user.home}/.m2/settings.xml] (or) for all maven users [${maven.home}/conf/settings.xml]

It consists of

1. License
2. Proxies
3. Servers
4. Mirrors
5. Profiles
6. Local repository, interactive mode, plugin groups

EX: - There is a calculator application which has 4 modules addition, subtraction, multiplication & division

Now on each module each developer works and push it to source code repository also each developer tests their code that is called unit testing

Ex:-the developer who worked on addition module will check weather his code is working correct

* after every developer push the code maven will build the source code and generate a build artifact and that is deployed on test servers now testers test the application which is called integration testing
* dev build = building one component (building only addition module)
* incremental build = integrating and building latest code
* qa build = whole component build