Maven

**Introduction**

Maven is a project management and comprehension tool that provides developers a complete build lifecycle framework. Development team can automate the project's build infrastructure in almost no time as Maven uses a standard directory layout and a default build lifecycle. Maven provides developers ways to manage the following –

* Builds
* Documentation
* Reporting
* Dependencies
* SCMs
* Releases
* Distribution
* Mailing list

**Convention over Configuration**

Maven uses **Convention** over **Configuration**, which means developers are not required to create build process themselves and also do not have to mention each and every configuration detail.

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

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

**Maven POM: (Project Object Model)**

* It is fundamental unit of work in Maven.
* It is an XML file that resides in the base directory of the project as pom.xml
* The POM contains information about the project and various configuration detail used by Maven to build the project(s). The POM contains information about the project and various configuration detail used by Maven to build the project(s).
* POM also contains the goals and plugins. While executing a task or goal, Maven looks for the POM in the current directory. It reads the POM, gets the needed configuration information, and then executes the goal
* Some of the configuration that can be specified in the POM are following –
  + project dependencies
  + plugins
  + goals
  + build profiles
  + project version
  + developers
  + mailing list
* Before creating a POM, we should first decide the project group (groupId), its name (artifactId) and its version as these attributes help in uniquely identifying the project in repository.

*Note:* There should be a single POM file for each project

**POM Example**

<project xmlns = "http://maven.apache.org/POM/4.0.0"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.companyname.project-group</groupId>

<artifactId>project</artifactId>

<version>1.0</version>

</project>

* All POM files require the project element and three mandatory fields: groupId, artifactId, version.
* Projects notation in repository is groupId:artifactId:version.
* Minimal requirements for a POM –

|  |  |
| --- | --- |
| **NODE** | **DESCRIPTION** |
| Project root | This is project root tag. You need to specify the basic schema settings such as apache schema and w3.org specification. |
| Model version | Model version should be 4.0.0 |
| groupId | This is an Id of project's group. This is generally unique amongst an organization or a project. For example, a banking group *com.company.bank* has all bank related projects. |
| artifactId | This is an Id of the project. This is generally name of the project. For example, consumer-banking. Along with the groupId, the artifactId defines the artifact's location within the repository. |
| version | This is the version of the project. Along with the groupId, It is used within an artifact's repository to separate versions from each other. For example −  *com.company.bank:consumer-banking:1.0*  *com.company.bank:consumer-banking:1.1.* |

**Build Lifecycle:**

A Build Lifecycle is a well-defined sequence of phases, which define the order in which the goals are to be executed. Here phase represents a stage in life cycle. As an example, a typical Maven Build Lifecycle consists of the following sequence of phases.

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

* There are always pre and post phases to register goals, which must run prior to, or after a particular phase.
* When Maven starts building a project, it steps through a defined sequence of phases and executes goals, which are registered with each phase.
* Maven has the following three standard lifecycles –
  1. clean
  2. default (or build)
  3. site
* A goal represents a specific task which contributes to the building and managing of a project.
* The order of execution depends on the order in which the goal(s) and the build phase(s) are invoked.
* For example, consider the command below. The clean and package arguments are build phases while the dependency:copy-dependencies is a goal.

mvn clean dependency:copy-dependencies package

* Here the clean phase will be executed first, followed by the dependency:copy-dependencies goal, and finally package phase will be executed.

**1. Clean Lifecycle**

* When we execute *mvn post-clean* command, Maven invokes the clean lifecycle consisting of the following phases.
  + pre-clean
  + clean
  + post-clean
* when *mvn clean* command executes, Maven deletes the build directory.
* We can customize this behavior by mentioning goals in any of the above phases of clean life cycle.

**2.Default (or Build) Lifecycle**

* This is the primary life cycle of Maven and is used to build the application. It has 21 phases
* When a phase is called via Maven command, for example mvn compile, only phases up to and including that phase will execute.
* Different maven goals will be bound to different phases of Maven lifecycle depending upon the type of packaging (JAR / WAR / EAR).

**3. Site Lifecycle**

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

**Build Profile:**

* A Build profile is a set of configuration values, which can be used to set or override default values of Maven build.
* Using a build profile, you can customize build for different environments such as Production v/s Development environments.
* Profiles are specified in pom.xml file using its activeProfiles/profiles elements and are triggered in variety of ways.
* Build profiles are majorly of three types.

|  |  |  |
| --- | --- | --- |
| **S.No** | **Type** | **Where it is defined** |
| 1 | Pre Project | Defined in the project POM file, pom.xml |
| 2 | Pre User | Defined in Maven settings xml file (%USER\_HOME%/.m2/settings.xml) |
| 3 | Global | Defined in Maven global settings xml file (%M2\_HOME%/conf/settings.xml) |

* *Profile Activation:* A Maven Build Profile can be activated in various ways:
  + Explicitly using command console input.
  + Through maven settings.
  + Based on environment variables (User/System variables).
  + OS Settings (for example, Windows family).
  + Present/missing files.

**Maven Repository**

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

1. **Local:** Maven local repository is a folder location on your machine. It gets created when you run any maven command for the first time. Maven local repository keeps your project's all dependencies (library jars, plugin jars etc.). When you run a Maven build, then Maven automatically downloads all the dependency jars into the local repository. Maven local repository by default get created by Maven in %USER\_HOME% directory. To override the default location, mention another path in Maven settings.xml file available at %M2\_HOME%\conf directory. When you run Maven command, Maven will download dependencies to your custom path.
2. **Central:** Maven central repository is repository provided by Maven community. It contains a large number of commonly used libraries. When Maven does not find any dependency in local repository, it starts searching in central repository using following URL −  <https://repo1.maven.org/maven2/>

Key concepts of Central repository are as follows −

* + This repository is managed by Maven community.
  + It is not required to be configured.
  + It requires internet access to be searched.

To browse the content of central maven repository, maven community has provided a URL −  <https://search.maven.org/#browse>.

Using this library, a developer can search all the available libraries in central repository.

1. **Remote:** Sometimes, Maven does not find a mentioned dependency in central repository as well. It then stops the build process and output error message to console. To prevent such situation, Maven provides concept of Remote Repository, which is developer's own custom repository containing required libraries or other project jars.

When we execute Maven build commands, Maven starts looking for dependency libraries in the following sequence −

* Step 1 − Search dependency in local repository, if not found, move to step 2 else perform the further processing.
* Step 2 − Search dependency in central repository, if not found and remote repository/repositories is/are mentioned then move to step 4. Else it is downloaded to local repository for future reference.
* Step 3 − If a remote repository has not been mentioned, Maven simply stops the processing and throws error (Unable to find dependency).
* Step 4 − Search dependency in remote repository or repositories, if found then it is downloaded to local repository for future reference. Otherwise, Maven stops processing and throws error (Unable to find dependency).

**Maven Plugin:**

Maven is actually a plugin execution framework where every task is actually done by plugins. Maven Plugins are generally used to –

* create jar file
* create war file
* compile code files
* unit testing of code
* create project documentation
* create project reports

A plugin generally provides a set of goals, which can be executed using the following syntax –

*Syntax:*  mvn [plugin-name]:[goal-name]

*Example:* mvn compiler:compile

Maven provided the following two types of Plugins –

1. **Build Plugins:** They execute during the build process and should be configured in the <build/> element of pom.xml.
2. **Reporting Plugins:** They execute during the site generation process and they should be configured in the <reporting/> element of the pom.xml.

Following is the list of few common plugins –

|  |  |
| --- | --- |
| **Plugin** | **Description** |
| clean | Cleans up target after the build. Deletes the target directory. |
| compiler | Compiles Java source files. |
| surefire | Runs the JUnit unit tests. Creates test reports. |
| jar | Builds a JAR file from the current project. |
| war | Builds a WAR file from the current project. |
| javadoc | Generates Javadoc for the project. |
| antrun | Runs a set of ant tasks from any phase mentioned of the build. |

**Creating Project:**

Maven uses archetype plugins to create projects. To create a simple java application, we'll use maven-archetype-quickstart plugin. In example below, we'll create a maven-based java application project.

* Let's open the command console and execute the following mvn command.

*mvn archetype:generate*

* Maven will start processing and will create the complete java application project structure.
* Now go to C:/MVN directory. You'll see a java application project created, named consumer Banking (as specified in artifactId).
* Using the above example, we can understand the following key concepts –

|  |  |
| --- | --- |
| **Folder Structure** | **Description** |
| consumerBanking | contains src folder and pom.xml |
| src/main/java | contains java code files under the package structure (com/companyName/bank). |
| src/main/test | contains test java code files under the package structure (com/companyName/bank). |
| src/main/resources | it contains images/properties files (In above example, we need to create this structure manually). |

**Build & Test Project:**

Go to C:/MVN directory where you've created your java application. Open consumerBanking folder. You will see the POM.xml file with the following contents.

<project xmlns = "http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.companyname.projectgroup</groupId>

<artifactId>project</artifactId>

<version>1.0</version>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

</dependency>

</dependencies>

</project>

Here you can see, Maven already added Junit as test framework. By default, Maven adds a source file App.java and a test file AppTest.java in its default directory structure.

Let's open the command console, go the C:\MVN\consumerBanking directory and execute the following mvn command.

C:\MVN\consumerBanking>mvn clean package

Maven will start building the project.

You've built your project and created final jar file, following are the key learning concepts −

* We give maven two goals, first to clean the target directory (clean) and then package the project build output as jar (package).
* Packaged jar is available in consumerBanking\target folder as consumerBanking-1.0-SNAPSHOT.jar.
* Test reports are available in consumerBanking\target\surefire-reports folder.
* Maven compiles the source code file(s) and then tests the source code file(s).
* Then Maven runs the test cases.
* Finally, Maven creates the package.

Now open the command console, go the C:\MVN\consumerBanking\target\classes directory and execute the following java command.

>java com.companyname.bank.App

You will see the result as follows −

Hello World!

*Adding Java Source Files:*

Open C:\MVN\consumerBanking\src\main\java\com\companyname\bank folder, create Util class in it as Util.java.

package com.companyname.bank;

public class Util {

public static void printMessage(String message){

System.out.println(message);

}

}

Update the App class to use Util class.

package com.companyname.bank;

/\*\*

\* Hello world!

\*

\*/

public class App {

public static void main( String[] args ){

Util.printMessage("Hello World!");

}

}

Now open the command console, go the C:\MVN\consumerBankingdirectory and execute the following mvn command.

>mvn clean compile

After Maven build is successful, go to the C:\MVN\consumerBanking\target\classes directory and execute the following java command.

>java -cp com.companyname.bank.App

You will see the result as follows −

Hello World!

**External Dependencies**

Maven does the dependency management using the concept of Repositories. But what happens if dependency is not available in any of remote repositories and central repository? Maven provides answer for such scenario using concept of External Dependency.

For example, let us do the following changes to the project created:

* Add lib folder to the src folder.
* Copy any jar into the lib folder. We've used ldapjdk.jar, which is a helper library for LDAP operations.

Here you are having your own library, specific to the project, which is an usual case and it contains jars, which may not be available in any repository for maven to download from. If your code is using this library with Maven, then Maven build will fail as it cannot download or refer to this library during compilation phase.

To handle the situation, let's add this external dependency to maven pom.xml using the following way.

<project xmlns = "http://maven.apache.org/POM/4.0.0"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.companyname.bank</groupId>

<artifactId>consumerBanking</artifactId>

<packaging>jar</packaging>

<version>1.0-SNAPSHOT</version>

<name>consumerBanking</name>

<url>http://maven.apache.org</url>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>ldapjdk</groupId>

<artifactId>ldapjdk</artifactId>

<scope>system</scope>

<version>1.0</version>

<systemPath>${basedir}\src\lib\ldapjdk.jar</systemPath>

</dependency>

</dependencies>

</project>

Look at the second dependency element under dependencies in the above example, which clears the following key concepts about External Dependency.

* External dependencies (library jar location) can be configured in pom.xml in same way as other dependencies.
* Specify groupId same as the name of the library.
* Specify artifactId same as the name of the library.
* Specify scope as system.
* Specify system path relative to the project location.

**Project Documents**

It is possible to create documentation of the application in one go.

* Go to C:/MVN/ consumerBanking
* Execute the following mvn command

mvn site

* Maven will start building the project.
* Your project documentation is now ready. Maven has created a site within the target directory.
* Open C:\MVN\consumerBanking\target\site folder. Click on index.html to see the documentation.
* Maven creates the documentation using a documentation-processing engine called [Doxia](https://maven.apache.org/doxia/index.html) which reads multiple source formats into a common document model.

**Project Templates:**

Maven provides users, a very large list of different types of project templates using the concept of Archetype. Maven helps users to quickly start a new java project using the following command.

mvn archetype:generate

*Archetype:* Archetype is a Maven plugin whose task is to create a project structure as per its template.

You can create any kind of project using single command in maven and can kick-start your development.

|  |  |
| --- | --- |
| **Archetype ArtifactIds** | **Description** |
| maven-archetype-archetype | An archetype, which contains a sample archetype. |
| maven-archetype-j2ee-simple | which contains a simplified sample J2EE application. |
| maven-archetype-mojo | which contains a sample a sample Maven plugin |
| maven-archetype-plugin | which contains a sample Maven plugin |
| maven-archetype-quickstart | which contains a sample Maven project |
| maven-archetype-site-simple | which contains a sample Maven site. |
| maven-archetype-webapp | which contains a sample Maven Webapp project. |

**SNAPSHOTS:**

A large software application generally consists of multiple modules and it is common scenario where multiple teams are working on different modules of same application. For example, consider a team is working on the front end of the application as app-ui project (app-ui.jar:1.0) and they are using data-service project (data-service.jar:1.0).

Now it may happen that team working on data-service is undergoing bug fixing or enhancements at rapid pace and they are releasing the library to remote repository almost every other day.

Now if data-service team uploads a new version every other day, then following problems will arise −

* data-service team should tell app-ui team every time when they have released an updated code.
* app-ui team required to update their pom.xml regularly to get the updated version.

To handle such kind of situation, SNAPSHOT concept comes into play.

* SNAPSHOT is a special version that indicates a current development copy.
* Unlike regular versions, Maven checks for a new SNAPSHOT version in a remote repository (JFrog) for every build.
* Now data-service team will release SNAPSHOT of its updated code every time to repository, say data-service: 1.0-SNAPSHOT, replacing an older SNAPSHOT jar.

*Snapshot vs Version:*

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

Although, in case of SNAPSHOT, Maven automatically fetches the latest SNAPSHOT on daily basis, you can force maven to download latest snapshot build using -U switch to any maven command.

mvn clean package -U

**Build Automation**

Build Automation defines the scenario where dependent project(s) build process gets started once the project build is successfully completed, in order to ensure that dependent project(s) is/are stable.

*Example*

Consider a team is developing a project **bus-core-api** on which two other projects **app-web-ui** and **app-desktop-ui** are dependent.

**app-web-ui**project is using 1.0-SNAPSHOT of bus-core-api project.

<project xmlns = "http://maven.apache.org/POM/4.0.0"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>app-web-ui</groupId>

<artifactId>app-web-ui</artifactId>

<version>1.0</version>

<packaging>jar</packaging>

<dependencies>

<dependency>

<groupId>bus-core-api</groupId>

<artifactId>bus-core-api</artifactId>

<version>1.0-SNAPSHOT</version>

</dependency>

</dependencies>

</project>

**app-desktop-ui** project is using 1.0-SNAPSHOT of bus-core-api project.

<project xmlns = "http://maven.apache.org/POM/4.0.0"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>app\_desktop\_ui</groupId>

<artifactId>app\_desktop\_ui</artifactId>

<version>1.0</version>

<packaging>jar</packaging>

<name>app\_desktop\_ui</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>bus\_core\_api</groupId>

<artifactId>bus\_core\_api</artifactId>

<version>1.0-SNAPSHOT</version>

<scope>system</scope>

<systemPath>C:\MVN\bus\_core\_api\target\bus\_core\_api-1.0-SNAPSHOT.jar</systemPath>

</dependency>

</dependencies>

</project>

**bus-core-api** project −

<project xmlns = "http://maven.apache.org/POM/4.0.0"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>bus\_core\_api</groupId>

<artifactId>bus\_core\_api</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>jar</packaging>

</project>

Now, teams of app-web-ui and app-desktop-ui projects require that their build process should kick off whenever bus-core-api project changes.

Using snapshot, ensures that the latest bus-core-api project should be used but to meet the above requirement we need to do something extra.

We can proceed with the following two ways −

* Add a post-build goal in bus-core-api pom to kick-off app-web-ui and app-desktop-ui builds.
* Use a Continuous Integration (CI) Server like Hudson/Jenkins to manage build automation automatically.

*Using Maven*:

Update bus-core-api project pom.xml.

<project xmlns = "http://maven.apache.org/POM/4.0.0"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>bus-core-api</groupId>

<artifactId>bus-core-api</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>jar</packaging>

<build>

<plugins>

<plugin>

<artifactId>maven-invoker-plugin</artifactId>

<version>1.6</version>

<configuration>

<debug>true</debug>

<pomIncludes>

<pomInclude>app-web-ui/pom.xml</pomInclude>

<pomInclude>app-desktop-ui/pom.xml</pomInclude>

</pomIncludes>

</configuration>

<executions>

<execution>

<id>build</id>

<goals>

<goal>run</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

<build>

</project>

Now, execute the following mvn command.

>mvn clean package -U

Maven will start building the project bus-core-api.

[INFO] Scanning for projects...

[INFO] ------------------------------------------------------------------

[INFO] Building bus-core-api

[INFO] task-segment: [clean, package]

[INFO] ------------------------------------------------------------------

...

[INFO] [jar: jar {execution: default-jar}]

[INFO] Building jar: C:\MVN\bus-core-ui\target\

bus-core-ui-1.0-SNAPSHOT.jar

[INFO] ------------------------------------------------------------------

[INFO] BUILD SUCCESSFUL

[INFO] ------------------------------------------------------------------

Once bus-core-api build is successful, Maven will start building the app-web-ui project.

[INFO] ------------------------------------------------------------------

[INFO] Building app-web-ui

[INFO] task-segment: [package]

[INFO] ------------------------------------------------------------------

...

[INFO] [jar:jar {execution: default-jar}]

[INFO] Building jar: C:\MVN\app-web-ui\target\

app-web-ui-1.0-SNAPSHOT.jar

[INFO] ------------------------------------------------------------------

[INFO] BUILD SUCCESSFUL

[INFO] ------------------------------------------------------------------

Once app-web-ui build is successful, Maven will start building the app-desktop-ui project.

[INFO] ------------------------------------------------------------------

[INFO] Building app-desktop-ui

[INFO] task-segment: [package]

[INFO] ------------------------------------------------------------------

...

[INFO] [jar:jar {execution: default-jar}]

[INFO] Building jar: C:\MVN\app-desktop-ui\target\

app-desktop-ui-1.0-SNAPSHOT.jar

[INFO] -------------------------------------------------------------------

[INFO] BUILD SUCCESSFUL

[INFO] -------------------------------------------------------------------

*Using Continuous Integration Service with Maven*

Using a CI Server is more preferable to developers. It is not required to update the bus-core-api project, every time a new project (for example, app-mobile-ui) is added, as dependent project on bus-core-api project. Hudson is a continuous integration tool written in java, which in a servlet container, such as, Apache tomcat and glassfish application server. Hudson automatically manages build automation using Maven dependency management.

Hudson considers each project build as job. Once a project code is checked-in to SVN (or any Source Management Tool mapped to Hudson), Hudson starts its build job and once this job gets completed, it starts other dependent jobs (other dependent projects) automatically.

In the above example, when bus-core-ui source code is updated in SVN, Hudson starts its build. Once build is successful, Hudson looks for dependent projects automatically, and starts building app-web-ui and app-desktop-ui projects.

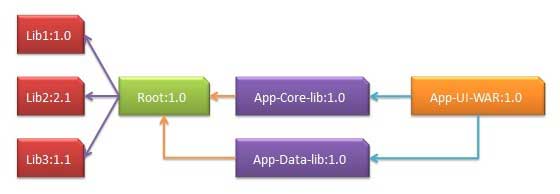
**Dependency Management:**

 when a library, say A, depends upon other library, say B. In case another project C wants to use A, then that project requires to use library B too.

Maven helps to avoid such requirements to discover all the libraries required. Maven does so by reading project files (pom.xml) of dependencies, figure out their dependencies and so on.

We only need to define direct dependency in each project pom. Maven handles the rest automatically.

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



Following are the detail of the above dependency graph −

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

Now when we build App-UI-WAR project, Maven will discover all the dependencies by traversing the dependency graph and build the application.

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

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

**Deployment Automation**

In project development, normally a deployment process consists of the following steps −

* Check-in the code from all project in progress into the SVN (version control system) or source code repository and tag it.
* Download the complete source code from SVN.
* Build the application.
* Store the build output either WAR or EAR file to a common network location.
* Get the file from network and deploy the file to the production site.
* Updated the documentation with date and updated version number of the application.

*Problem Statement*

There are normally multiple people involved in the above-mentioned deployment process. One team may handle check-in of code, other may handle build and so on. It is very likely that any step may be missed out due to manual efforts involved and owing to multi-team environment. For example, older build may not be replaced on network machine and deployment team deployed the older build again.

*Solution*

Automate the deployment process by combining the following −

* Maven, to build and release projects.
* Subversion, source code repository, to manage source code.
* Remote Repository Manager (Jfrog/Nexus) to manage project binaries.

*Update Project POM.xml*

We will be using Maven Release plug-in to create an automated release process.

For Example: bus-core-api project POM.xml.

<project xmlns = "http://maven.apache.org/POM/4.0.0"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>bus-core-api</groupId>

<artifactId>bus-core-api</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>jar</packaging>

<scm>

<url>http://www.svn.com</url>

<connection>scm:svn:http://localhost:8080/svn/jrepo/trunk/

Framework</connection>

<developerConnection>scm:svn:${username}/${password}@localhost:8080:

common\_core\_api:1101:code</developerConnection>

</scm>

<distributionManagement>

<repository>

<id>Core-API-Java-Release</id>

<name>Release repository</name>

<url>http://localhost:8081/nexus/content/repositories/

Core-Api-Release</url>

</repository>

</distributionManagement>

<build>

<plugins>

<plugin>

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

<artifactId>maven-release-plugin</artifactId>

<version>2.0-beta-9</version>

<configuration>

<useReleaseProfile>false</useReleaseProfile>

<goals>deploy</goals>

<scmCommentPrefix>[bus-core-api-release-checkin]-<

/scmCommentPrefix>

</configuration>

</plugin>

</plugins>

</build>

</project>

In Pom.xml, following are the important elements we have used –

|  |  |
| --- | --- |
| **Element** | **Description** |
| SCM | Configures the SVN location from where Maven will check out the source code. |
| Repositories | Location where built WAR/EAR/JAR or any other artifact will be stored after code build is successful. |
| Plugin | maven-release-plugin is configured to automate the deployment process |

*Maven Release Plug-in*

The Maven does the following useful tasks using maven-release-plugin.

mvn release:clean

It cleans the workspace in case the last release process was not successful.

mvn release:rollback

Rollback the changes done to workspace code and configuration in case the last release process was not successful.

mvn release:prepare

Performs multiple number of operations, such as −

* Checks whether there are any uncommitted local changes or not.
* Ensures that there are no SNAPSHOT dependencies.
* Changes the version of the application and removes SNAPSHOT from the version to make release.
* Update pom files to SVN.
* Run test cases.
* Commit the modified POM files.
* Tag the code in subversion
* Increment the version number and append SNAPSHOT for future release.
* Commit the modified POM files to SVN.

mvn release:perform

Checks out the code using the previously defined tag and run the Maven deploy goal, to deploy the war or built artifact to repository.

Execute the following mvn command.

>mvn release:prepare

Maven will start building the project. Once build is successful run the following mvn command.

>mvn release:perform

Once build is successful you can verify the uploaded JAR file in your repository.