Maven as a DevOps tool

What is apache maven?

Apache Maven is an build tool mainly for Java applications to help the developer at the whole process of a software project.

Maven is a powerful project management tool that is based on POM (project object model), used for projects build, dependency and documentation.

It is a tool that can be used for building and managing any Java-based project.

Maven makes the day-to-day work of Java developers easier and helps with the building and running of any Java-based project.

What Maven does ?

Compilation of Source Code

Running Tests (unit tests and functional tests)

Packaging the results into JAR’s,WAR’s,RPM’s,etc..

Upload the packages to remote repo’s (Nexus,Artifactory)

Maven Build lifecycle

Maven defines and follows conventions. Right from the project structure to building steps, Maven provides conventions to follow. If we follow those conventions, with minimal configuration we can easily get the build job done.

A life cycle has multiple phases. For example, ‘default’ lifecycle has following phases (listed only the important phases),

compile — compiles the source code

test — executes unit test cases

package — bundles the compiled code (Ex: war / jar)

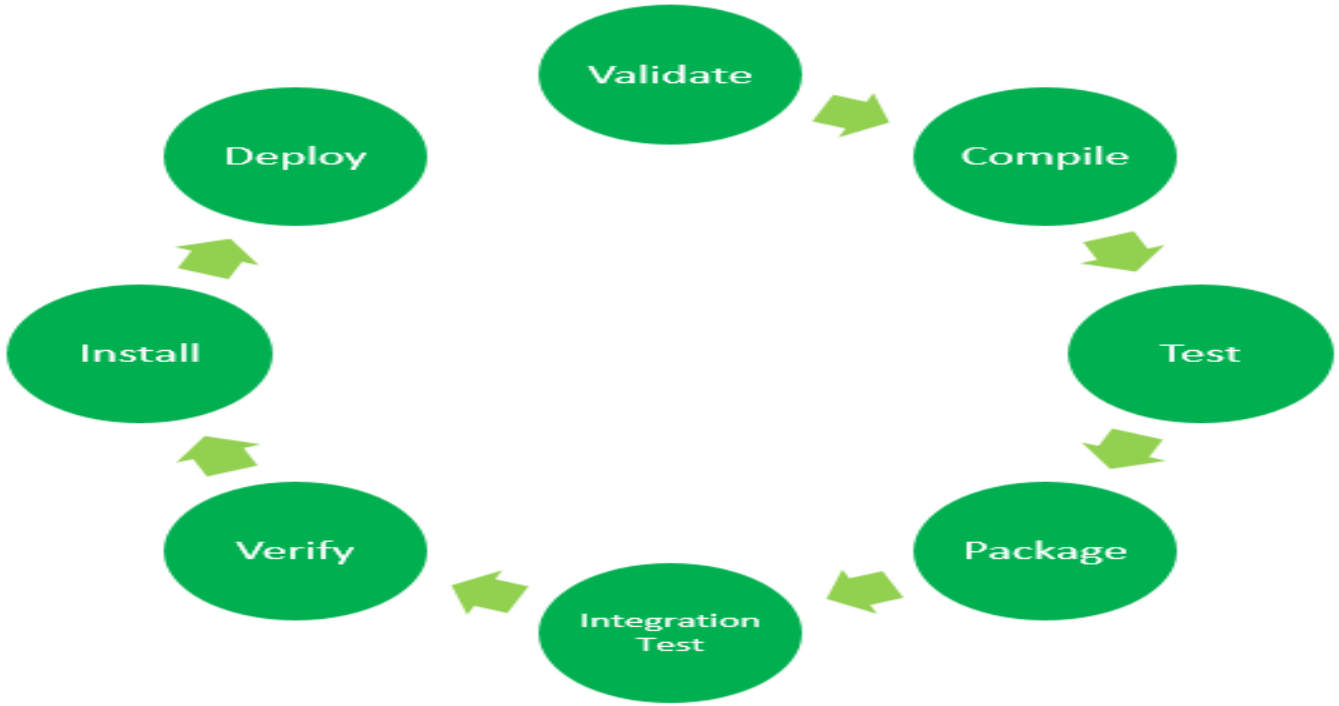
install — stores the built package in local Maven repository

deploy — store in remote repository for sharing

So to go through the above phases, we just have to call one command mvn install, all the phases are executed sequentially till the ‘install’ phase.

More on maven lifecycle

Maven Lifecycle: Below is a representation of the default Maven lifecycle and its 8 steps: Validate, Compile, Test, Package, Integration test, Verify, Install and Deploy.



More on Life-cycle of Maven

Validate: This step validates if the project structure is correct. For example – It checks if all the dependencies have been downloaded and are available in the local repository.

Compile: It compiles the source code, converts the .java files to .class and stores the classes in target/classes folder.

Test: It runs unit tests for the project.

Package: This step packages the compiled code in distributable format like JAR or WAR.

Integration test: It runs the integration tests for the project.

Verify: This step runs checks to verify that the project is valid and meets the quality standards.

Install: This step installs the packaged code to the local Maven repository.

Deploy: It copies the packaged code to the remote repository for sharing it with other developers.

More.....

Maven follows a sequential order to execute the commands where if you run step *n*, all steps preceding it (Step 1 to *n-1*) are also executed.

For example – if we run the Installation step (Step 7), it will validate, compile, package and verify the project along with running unit and integration tests (Step 1 to 6) before installing the built package to the local repository.

Some of the basic maven commands

mvn clean: Cleans the project and removes all files generated by the previous build.

mvn compile: Compiles source code of the project.

mvn test-compile: Compiles the test source code.

mvn test: Runs tests for the project.

mvn package: Creates JAR or WAR file for the project to convert it into a distributable format.

mvn install: Deploys the packaged JAR/ WAR file to the local repository.

mvn deploy: Copies the packaged JAR/ WAR file to the remote repository after compiling, running tests and building the project.

More on maven commands

Generally when we run any of the above commands, we add the mvn clean step so that the target folder generated from the previous build is removed before running a newer build. This is how the command would look on integrating the clean step with install phase: mvn clean install

Similarly, if we want to run the step in debug mode for more detailed build information and logs, we will add -X to the actual command. Hence, the install step with debug mode on will have the following command: mvn -X install

Consider a scenario where we do not want to run the tests while packaging or installing the Java project. In this case, we use -DskipTests along with the actual command. If we need to run the install step by skipping the tests associated with the project, the command would be: mvn install -DskipTests

Maven Repository

Repository is where the build artifacts are stored.

Build artifacts means, the dependent files (Ex: dependent jar files) and the build outcome (the package we build out of a project).

There are two types of repositories, local and remote.

Local maven repository(.m2) is in the user's system.

It stores the copy of the dependent files that we use in our project as dependencies.

Remote maven repository is setup by a third party(nexus) to provide access and distribute dependent files. Ex: repo.maven.apache.org from internet.

Maven central

Maven Central, a.k.a. the Central Repository, is the default repository for Maven, SBT, Leiningen, and many other JVM based build tools.

It has been around since 2002, and serves many terabytes of assets every year.

Maven Central can be accessed from https://repo.maven.apache.org/maven2/.

Artifactory and artifact

As a Maven repository, Artifactory is both a source for artifacts needed for a build, and a target to deploy artifacts generated in the build process.

An artifact is a file, usually a JAR, that gets deployed to a Maven repository.

Artifactory is a Repository Manager that functions as a single access point organizing all of your binary resources including proprietary libraries, remote artifacts and other 3rd party resources.

POM

POM stands for Project Object Model.

It is fundamental unit of work in Maven. It is an XML file that resides in the base directory of the project as pom.xml And has all the configuration settings for the project build.

Generally we define the project dependencies (Ex: dependent jar files for a project), maven plugins to execute and project description /version etc.

Simplest pom.xml should have 4 important information.

Imp functions

modelVersion-4.0.0 (POM version for Maven 2 and is always required)

groupId —will identify your project uniquely across all projects, ex:ebs.obill.webs, com.companyname.project

artifactId — is the name of the jar without version(keeping in mind that it should be jar-name friendly)

version — if you distribute it then you can choose any typical version with numbers and dots (1.0, 1.1, 1.0.1, …)

```
<project>
<modelVersion>4.0.0</modelVersion>
<groupId>com.intuit.jsapp</groupId>
<artifactId>js-app</artifactId>
<version>1</version>
</project>
```

Maven Dependencies

There is an element available for declaring dependencies in project pom.xml This is used to define the dependencies that will be used by the project.

Maven will look for these dependencies when executing in the local maven repository. If not found, then Maven will download those dependencies from the remote repository and store it in the local maven repository.

Mvn dependencies

Example declaring junit and log4j as project dependencies, here scope describes under which context this dependency will be used.

```
<dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.12</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
```

Maven plugins

All the execution in Maven is done by plugins.

A plugin is mapped to a phase and executed as part of it.

A phase is mapped to multiple goals.

A goal represents a specific task which contributes to the building and managing of a project.

Those goals are executed by a plugin. We can directly invoke a specific goal while Maven execution.

A plugin configuration can be modified using the plugin declaration.

```
<build>
 <finalName>springexcelexport</finalName>
        <plugins>
         <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-war-plugin</artifactId>
                <version>2.2</version>
        </plugin>
        <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.16</version>
                <configuration>
                        <skipTests>true</skipTests>
                </configuration>
        </plugin>
</build>
```

Maven project structure

Maven uses a convention for project folder structure.

If we follow that, we need not describe in our configuration setting, what is located where.

Maven knows from where to pick the source files, test cases etc.

```
BANL13dd26e76:jaisriram agv$ tree
.
├── pom.xml
└── src
    ├── main
    │   └── java
    │       └── com
    │           └── intuit
    │               └── App.java
    └── test
        └── java
            └── com
                └── intuit
                    └── AppTest.java

9 directories, 3 files
```