Maven

Contents

Introduction

What is Maven

- The developers call it a build automation tool but it is much more than that it is a **project management tool**.
- Building a project means compiling the source code, running the tests which could be unit tests as well as integration tests, packaging the compiled code into jar files, bundling these jar files into a web archive or a war file, deploying these wars on to the servers and several other tasks.
- All these tasks can be automated using tools such as maven and etc. If you have worked with tools like ant apache ant then you know that we have to come up with a lot of XML configuration to perform these tasks or to automate these tasks for our application whereas **maven uses convention over configuration** that is if we follow a certain project structure when we create our projects, we can simply execute a maven command such as mvn install and maven will compile the source code under the src/main/java, run the unit tests under src/test/java and if the tests pass it will bundle or package the compiled classes into a jar file if it is a standalone java application, if it is a web application it will bundle it into a war file and it can also deploy the war file onto a web application.
- This folder structure will be slightly different for a standalone Java project, for a web application and for different types of projects but the beauty is that we need not create these folders manually for each type of project. Maven provides several **archetypes,** think about these archetypes as templates.
- We can execute a command with an archetype and it will create the standard folder structure required by maven for us.
- There are different types of archetypes like standalone, webapp, EAR etc.
- Another additional advantage is that all the popular ides such as Eclipse Intelli J come with inbuilt support these archetypes. We can create different types of maven projects from within an IDE and also run and execute our build from within eclipse or intelliJ.

Why Maven

- In addition to its convention over configuration maven offers several other advantages such as a **common interface** for the developers.
- Before maven, if we as developers had to work on an open source project or even another project within our company or enterprise, we had first understand how to build that project because each team might have their own way of configuring the build when they use other tools other than maven which is both time consuming and hectic that is where maven simplifies things by simply grabbing the project from the source control.

We can execute maven clean install and the project will be built for us because all the developers now will follow **one standard project structure** which is specified by maven.

- Secondly maven is not just a build management tool it can also **manage dependencies** that is if our project depends on other projects from within the organization or even in the open source world. Maven can grab those projects, download those dependencies and it will use them to compile our source code, run the test and even bundle these dependencies into war files and EAR files.
- Maven has a concept called **repository** where it puts all the artifacts and its plugins and it will download them on the fly so if a project needs a particular dependency in the open source world or even within our firm those dependencies will be pulled from this repository and when we build our projects our projects can also be pushed into these repositories.
- These repositories can be on the Internet. There is a public maven repository where all the open-source projects are available as jars. Also many companies maintain their own internal repository which runs on their servers that way they can have control on what jar files or what projects who open source projects the teams within that company can use and also they can share their artifacts within the company by pushing them to this repository.
- Maven is very **lightweight.** When we download and install it, it grabs whatever it needs as **plugins,** it uses a **plug-in model** for example a compiler plugin, surefire plugin that can run unit tests as well as the wsimport plugin which can be used to generate stubs from a web services WSDL file and many other plugins are there.
- So maven when uses a plug-in model through which these can be **reused** across projects and even if a upgrade happens to where the compiler or the Surefire plugin or any other plug-in maven will automatically download the latest plug-in for example this surefire plug-in will be downloaded by maven only when we run our very first unit test after that it will not download it until this plugin changes that is the beauty of using a plug-in model reuse as well as if an upgrade happens it can pull it on the fly.
- With all these advantages maven of course is one of the **best build management, dependency management and completely a project management tool** that should be used.

Maven Key Concepts

Archetype

- `mvn archetype:generate -DgroupId=com.bharath -DartifactId=hellomaven -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false`
- `archetype:generate` is a maven **goal**. A goal describes a unit of work in maven.
- -D – parameters passed to the goal.
- `-DgroupId` – is the package Id.
- `-DartifactId` - is the name of the project.
- `-DarchetypeArtifactId` – is what type of project we are creating.
  E.g. `maven-archetype-quickstart` is for standalone Java project.
- When you run above command, it will create necessary project structure (folders and files) based on given `archetypeArtifactId.`

POM

- Project Object Model
- When we execute a maven command to build a project the maven command looks for the pom.xml under the project for information
- packaging - defines what type of project it is. E.g. jar, war
- name – user friendly name
- url – URL of the company.
- groupId, artifactId, packaging and version are together called **maven coordinates**.
- dependency – this section is where we add all the libraries that our project needs in order to function.

Build

- `mvn build` – to buildyour maven project.

Plugins and Goals

- A maven **plugin** is a collection of one or more **goals**.
  E.g.
  `archetype:generate` - generate goal from the archetype plugin
  `install:install` – install goal from the install plugin.
- A **goal** can be a specific task which we usually run independently or it can be a part of a bigger build.
  E.g. compile, test, package
- The goals can take parameters (used with -D). The goals can assume some default values for those parameters.

- We refer to a goal using pluginId:goalId. E.g. `archetype:generate` - generate goal from the archetype plugin.
- By itself maven does not know how to create a project, compile it, package it, etc. It uses the plugins like compiler, jar to get the work done.
- Every maven project gets a set of these default plugins through the parent settings but we can override them by defining them in our pom.xml. E.g. we can override the default java version which the compiler uses.

Life Cycle Phases

- When you run the maven install command, we ask maven to execute a life cycle **phase**.
- Maven has multiple lifecycle phases starting from process-resources, compile, test, package and many more.
- When we execute a particular phase, all its previous phases are also executed. E.g. when we execute package, it will execute `process-resources, compile, test` then `package`.
- Each life cycle phase is associated with one or more goals for example the process-resources is associated with a resources plugin's resources goal.
- The phases are something maven internally knows but they don't do any work on their own. They are simply a part of the life cycle actual work is done by the plugin **goals** associated with those phases.
- E.g.
  `compile` phase is associated with `compiler:compiler` goal.
  `test` phase is associated with `surefire:test` goal.

`package` phase is associated with `jar:jar` goal.

- We can have multiple goals associated with a phase but usually there is only one goal.
- You can execute a particular goal instead of a phase. E.g. `mvn surefire:test`. Maven will then execute all the phases before the phase with which this goal is associated.
- Depending on the type of the project, the association might change. If you are building a stand-alone java project, the package phase can be associated with jar:jar and if it is a web application, project maven dynamically associate it with a war:war goal.

Coordinates

- `groupId, artifactId, packaging` and `version` are together called **maven coordinates**.
- These coordinates are internally represented by maven as `groupId:artifactId:packaging:version` and it will decide where our project will be located in the maven repository and the name of the final output that is the jar file or war file or ear file that comes out of a maven build.
- The packaging decides what type of project is being built whether it's a jar project or a warproject or a EAR project.
- So using all these coordinates a particular project can be pushed into a maven repository and also later on when we need to add dependencies so if one project depends on another project, using these four coordinates we will define that dependency in our pom.xml

Repositories

- Maven is very small when we install it and it downloads what is required to do the build from the central repository on the fly.
- The default maven **central repository** location is http://repo.maven.apache.org/maven2/ (When you build, this repository is outputted in the console)
- The artifacts are stored under a certain folder structure (`/groupId/artifactId/version`) on this server. The folder structure is derived from the maven coordinates in the pom.xml of a project.
- If we want certain libraries that are not available in the **central repository** then we'll have to use a **enterprise level repository**.
- Maven also creates a **local repository** on a developer's machine. It's not going to fetch the plugins and jars from central repo every time we run maven install, it will pull these jars the very first time and then it will locate or put them into our local repository on our machine. The location of local repository is C:\Users\<user>\.m2

Customization

- Everything in maven works as plugins and when we download maven it has only the core part of it but if you want to customize its behavior or need any additional functionality we get it through plugins. E.g. maven compiler plugin.

Some commands

- To compile, mvn install
- To skip tests, mvn install **–DskipTests**

Dependency Management

- Maven will download the dependencies mentioned in the pom file to the local repository under your user folder and it will add them to the classpath during compile time, runtime and it can even bundle them into a war file when you work on a web application project.
- Another beauty of maven is it can pull the dependencies **transitively** for example when you add the spring context, it also downloads all the dependencies that the spring context depends on spring aop, spring bean, spring core etc.

Multi module project

- Parent pom's packaging is always pom.
- Define <modules> in the parent pom.
- Include <parent> tag in the child modules so establish the parent-child mapping.
- We need not define version in the child modules. The child modules derive the version from the parent module.
- When we build a multi module project using mvn clean install, maven puts the declared modules in a **reactor**. The reactor decides the **build order** if one project depends on another project that project needs to be built first so maven will do all the dependency checks and places the projects in the build order accordingly. Then maven **builds** a project and **pushes** it to the local repository so that if any project/module needs this dependency can include it in their pom.xml.

Scopes

- Maven allows us to specify the visibility of our project dependencies using a **scope** element.
- Which scope we use will impact on when those dependencies are available in the maven lifecycle for our project.
- There are total of six dependencies starting with **compile**, **runtime**, **provided**, **test**, **system** and **import** and depending on what scope you mark a dependency with, it will affect the availability of that dependency during the maven lifecycle for our projects.

Compile

- If we use the **compile** time scope then that those dependencies will be available during the project **build** that is when our class are compiled when our tests are compiled and the **test** are run and then our application is **run**.
- By **default** if we don't specify any scope compile is the scope that will be used by Maven.

Provided

- These are the dependencies that are required during the build, test and run but they are not required to be exported meaning they need not be a part of our application when they are deployed.
- For example if we are building a web application then the servlet-api is required locally in our project when we compile our classes when we run our test but it need not go into the war file or the web application when it is deployed to a server because servers or application containers or web containers have the servlet-api jar that is the reason will mark a servlet-api dependency as provided.

Runtime

- These dependencies are available only for running the tests (test) as well as running our application (run). They will NOT be available for compilation.

Test

- These are available to compile our test and run the tests.
- E.g. jUnit. We don't need jUnit to compile our application source classes, we need it for compiling our test as well as running the tests.

System

- System scope is not that popular but still system scope is similar to that offer a provided scope but instead of an application container or a web container providing that dependency or that is not even pulled from a maven repository. We will copy that dependency onto a subdirectory of our project and we will point to it when we use the

system scope will also provide a path to that particular dependency which is relative to our project usually. So this is very rarely used.

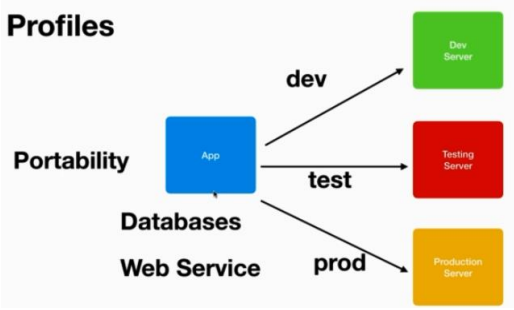
- But remember that if you want to point to a library which is not a part of a maven dependency and not even provided by a web container then you will use the system scope and point to that external dependency.

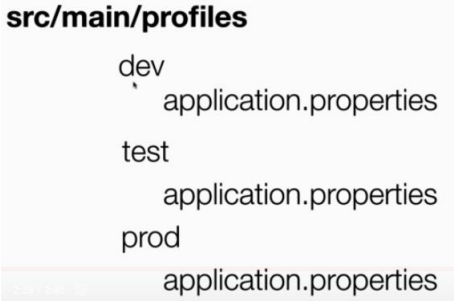
Import

- As the name itself says this is used on **Pom based projects,** not jar projects, not war projects.
- Used along with `dependencyManagement`.

Profiles

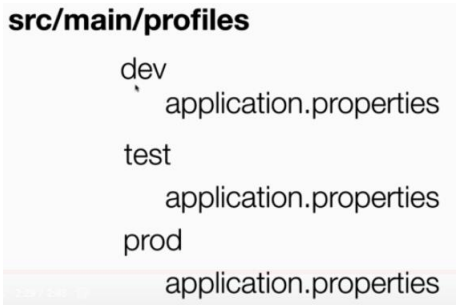
- **Build portability** means the deployment engineer or a DevOps engineer or even a developer should be able to build the application across different environments without changing any configuration or by changing a little configuration.
- Maven allows build portability through its **profiles**.
- We create different profiles and based on which profile is currently activated and when our application is built, maven will pick that information and make it a part of our application so that it can be deployed.
- So when we are building our application on DEV we will activate the dev profile and maven will pick the configuration for the dev profile so only those files that has the database information on the web service connection information for dev. Similarly for other environment like IT, UAT, PROD.





Create Configuration

- Create environment specific configuration files as mentioned in below folder structures.
- The subfolders under profiles folder will be as per your requirements.



Configure Profiles

- Details – https://maven.apache.org/guides/introduction/introduction-to-profiles.html
- What are the different types of profile? Where is each defined?
    - Per Project
        - - Defined in the POM itself (pom.xml).
    - Per User
        - - Defined in the Maven-settings (%USER_HOME%/.m2/settings.xml).
    - Global
        - - Defined in the global Maven-settings (${maven.home}/conf/settings.xml).
- Add <profiles> element in your pom.xml. This is where our profiles will live.
- Under profiles element, create <profile> tag for each environment.
- Inside profile element, you can properties, resources sections.
- There are many parameters available for different customizations.
- E.g.

```
<profile>
      <id>dev</id>
      <properties>
        <!- should match folder name under profiles -->
        <build.profile.id>dev</build.profile.id>
        ..
      </properties>
      <build>
        ...
      </build>

  </profile>
```

- Now build your project based on profile. E.g.
  `mvn install –pdev`
  This will activate the dev profile, projectwill be built and configuration file from dev will be picked up.
- E.g.2
  https://mkyong.com/maven/maven-profiles-example/

Tips and Tricks

- Spring tool suite (STS) is a very powerful IDE that helps us build spring based projects very easily. It is very similar to eclipse with support for special support for spring based projects.
- spring-boot-starter-parent is a **BOM**, which is a special type of pom. BOM stand for **Bill of Materials** within which all the versions of various libraries required for our projects are defined.
- @RunWith(SpringRunner.class) – with this we tell Spring Boot to use SpringRunner instead of default JUnitRunner class to run our Test classes.
- @SpringBootTest - This annotation tells spring boot to search for a class that is marked with spring Boot application and use that class to create a spring application context, a container with all the beans in that application so that we can start testing those beans in that application.