MAVEN DEPENDENCIES

INTRODUCTION :

* In this module , we will be covering dependencies and how they affect our application.
* We will cover versions , types , what transitive dependencies are and how they are used in our application , scopes of dependencies.
* What it means to have a test code , runtime and compile.

DEPENDENCY OVERVIEW

* Dependencies are simply other resources that we want to use within our application.
* Maven will pull transitive dependencies based on the dependency that we list.
* People generally use Maven for this purpose alone.
* We need to have three things :  
  GroupId – package structure Eg : com.mycompany.product  
  ArtifactId – Name of the actual Item I want to use.  
  version – simply the version number , nothing more , nothing less.
* See the example screenshot in the folder.
* The naming convention was created by the people who created the Artifact that we are going to use .

VERSION :

* Version are the release number for the artifact that we want to use.
* There is only one version that we will be discussing which is the SNAPSHOT.
* All of your internal development should start off as a SNAPSHOT.
* We should also be aware of the third party tools that we may be using and if we want the updated code , we can take the SNAPSHOT for eg if we are using Spring.
* SNAPSHOT allows us to push new code upto our repository and have our IDE or the command line also check for changes every time.
* Every time that we compile , it should check if there is new or updated code and there is a way of specifying the SNAPSHOT.
* Other than the SNAPSHOT version , everything else based on the naming convention is left to the user.
* Myapp-1.0-SNAPSHOT.jar 🡪 1.0-SNAPSHOT is version , it should be all CAPS.
* Our changes are always download. It helps us so much that it doesn’t make us re-release our versions , it automatically checks for updates and updates our code.
* We never want to deploy to a production with a SNAPSHOT.
* Because we cant reproduce or recreate our code.
* A release doesn’t have to have a specific naming convention.  
  Eg :   
  myapp-1.0.jar  
  myapp-1.0.1.jar 🡪 For maintenance releases for bug fixes  
  myapp-1.1 🡪 Maybe used for some major releases
* These are industry common terms but doesn’t affect Maven , things like milestone releases.  
  Eg : myapp-1.0-M1.jar 🡪 Indicates a milestone release  
  myapp-1.0-RC1.jar 🡪 For release candidate  
  Some people when they do their final release they name it as  
  myapp-1.0-RELEASE.jar 🡪 Final release
* Some people use the final keyword to denote the final release.

TYPES :

* Types denote the type of resource that we want to include in our application.
* Most of the times , we want to include a JAR but there are other types worth mentioning.
* Current code packaging types are : pom , jar , maven-plugin , ejb , war , ear , rar , par  
  The default packaging type is JAR.
* JAR , WAR , EAR , RAR , PAR – these are all glorified zip files.
* The one here that is interesting is the type POM.
* All those dependencies that are specified into our POM are downloaded into our application.
* Lets say we have Web Services in our organization , we want to group all those dependencies together so that any time we want to do a web service , like jersey or metro libraries. I can put those on a restful services POM and refer that into my application so that it would download all the dependencies into our application.
* If we are building an Artifact for other people to consume :  
  <groupId>com.pluralsight</groupId>  
  <artifactId>HelloWorld</artifactId>  
  <version>1.0-SNAPSHOT</version>  
  <modelVersion>4.0.0</modelVersion>  
  <packaging>JAR</packaging>
* Your packaging and what someone is going to request as the type say JAR are the same things.
* You can refer your core-app JAR within the web-app JAR to utilize all the business logic that you wrote within the core-app

TRANSITIVE DEPENDENCIES :

* Without a doubt , the main reason why people started using Maven was because of transitive dependencies.
* If we add a dependency :   
  Something like hibernate  
  It will go and pull all the transitive dependency that hibernate needs.
* If suppose I have conflicts , it will resolve those two and always go for the newer version.
* The transitive dependencies that org.hibernate 🡪 hibernate-core needs :  
  antlr 2.7.7  
  dom4j 1.6.1  
  hibernate-commons-annotations 4.0.1  
  hibernate-core 4.1.6  
  hibernate-jpa-2.0-api 1.0.1  
  javassist 3.15.0-GA  
  jboss-logging 3.1.0GA  
  jboss-transaction-api\_1.1\_spec 1.0.0
* It will automatically pull all these transitive dependency into my application.
* They often scare people because we don’t know from where all of these are coming from.
* You gotta have faith on the people who developed the hibernate framework , because they knew what all dependencies it needs when people coded with hibernate , therefore they all sat down and told that these are all the transitive dependencies that we need to add.
* It means hibernate 4.1.6 needs all those specific versions of the above transitive dependencies.

SCOPES

* A lot of people use Maven without taking the advantage of scopes.
* There are some scopes though that are particularly worth mentioning.
* There are in particular 6 scopes available for your dependencies :   
  compile – is the default scope , we don’t need to list it , artifacts are available everywhere inside my application  
  provided – It’s a lot like compile , but artifact is provided only where it is deployed. Eg : servlet-api or xml-api , Its available in all my phases but not included in my final artifact.   
  runtime – not needed for compilation but needed for execution , these are dynamically loaded libraries like my jdbc jar , we need a driver that’s dynamically loaded at runtime. Its kind of the opposite of provided. We don’t need it for compiling , packaging , developing but we need it to be loaded during runtime. Its also not included in the final artifact  
  test – only available for test compilation and testing phase , not included in the final artifact  
  system – similar to provided , not recommended to use , very brittle , we are going to hardcode a path to a jar on my file system . This is why we are using Maven , don’t use system  
  import – advanced topic used with dependencyManagement sections , means sharing resources across multiple POMs.

DEMO OVERVIEW AND DEMO :

* We are going to do the following in the demo :  
  Add a dependency  
  Refer transitive dependencies  
  Use scopes
* Go to POM.xml
* Add a dependency  
  <dependencies>  
  <dependency>  
  <groupId>commons-lang</groupId>  
  <artifactId>commons-lang</artifactId>  
  <version>2.1</version>  
  </dependency>  
  </dependencies>
* Click on save.

Add one more dependency  
<dependency>  
<groupId> junit</groupId>  
<artifactId> junit </artifactId>  
<version>4.12</version>  
<scope>test</scope> 🡪 We are telling that this is available only for test phase , we have a   
</dependency> transitive dependency called as Hamcrest for JUnit

* Inside our IDE we have a dependencies tab , Add one more dependency  
    
  <dependencies>  
  <dependency>  
  <groupId>org.hibernate</groupId>  
  <artifactId>hibernate-core</artifactId>  
  <version>4.1.6.Final</version>  
  </dependency>  
  </dependencies>
* Save the POM.xml and go on to the dependencies tab and see what other dependencies it pulled.
* We can notice that hibernate-commons-annotations already include a jboss-logging and since hibernate-core already pulled that dependency , it resolved the conflict with jboss-logging and used the version that was the latest.

SUMMARY :

* In this module , we went through dependencies
* Versions , Scope and how to install transitive dependencies.
* SNAPSHOT 🡪 Download the latest version of code and never want to go to production with SNAPSHOT.
* Types of packaging , JAR and POM are very important.
* Pull hibernate and pull all transitive dependencies along with resolving conflicts.