MAVEN STRUCTURE

INTRODUCTION

* This module is on the structure of Maven.
* By structure we are referring to folder and file structure , location of file and how code gets compiled.
* We will also discuss the POM file and where these elements get pulled from.

OUTLINE :

* Folder Structure  
  POM file basics – common elements we look inside the POM file  
  Basic Commands and Goals  
  Dependencies – how to add a dependency and stuff  
  Local Repository
* We will look at how the local repository is created , what are all stored in there and the naming convention followed.

FOLDER STRUCTURE :

* By default Maven looks for a SRC/MAIN/JAVA directory.
* It compiles all of our code to a Target directory and then it does all of this by referencing our defaults or whatever we have overridden in POM.xml file.
* If you can see the folder structure we followed for HelloWorld project.
* SRC,Target and Pom file are the top of our application.
* SRC/MAIN/JAVA directory :   
  Its where we store our java code   
  It’s the beginning of our package declaration  
  Eg : com.yourcompanydivision.division
* If we are using other languages like groovy or scala  
  SRC/MAIN/GROOVY
* Testing in Maven :   
  We actually set up a SRC/TEST/JAVA  
  We can keep test code separate from production code but we can use the same package names  
  Specifically SRC/TEST/JAVA directory is better suited for Unit Testing and not necessarily for Integration Test.
* Target Directory:  
  Where everything gets compiled to  
  Where our unit test gets run from  
  Contents in this directory gets packaged in the form of JAR,WAR,EAR etc  
  When we do a mvn package command :
* We get the following structure.  
    
    
    
  classes  
  maven-archiver  
  surefire  
  test-classes  
  HelloWorld-1.0-SNAPSHOT.jar
* The maven-archiver directory is what we are referring to our package and surefire and test classes are how our code gets unit tested.
* We can also see how our code gets artifacted into a JAR package as HelloWorld-1.0-SNAPSHOT.jar

POM.XML :

* Refer to the folder for snapshot of POM.xml
* POM.xml the basic one doesn’t describe the package structure , our directory structure cause we are following all of the defaults.
* It can be divided into 4 basic parts :  
  Project Information – groupId Eg : com.pluralsight , org.junit , com.acme   
  artifactId – name of our application.  
  version – what our current version is.  
  packaging - how we want to distribute our application , like JAR , WAR  
  Dependencies –What artifacts that we want to use within our application  
  What are the direct dependencies used and the transitive  
  Build Section – What plugins we want to use , what should we need to build our code , and the directory structure for a customized one specified by the user , XML files.  
  Repositories – Where we want to download our artifacts from , Right now we are downloading everything from central maven repo , we will get into why it can always be relied upon , what if our own organization uses a separate artifact to download and use , we wont be posting it on the maven repo because its something that we use.

DEPENDENCIES :

* Dependencies are what we want to use inside of our application.
* The next module covers dependencies in a very thorough fashion.
* Dependencies are imported by their naming convention.
* What their artfactId , groupId and version is
* The advantage is that it pulls all the transitive dependencies that I need.
* Adding a dependency to our POM.xml is very simple :   
  Just list the dependency that we want to use  
  Transitive dependencies will be pulled by Maven  
  We need to know three things  
  GroupId , ArtifactId , Version
* How it looks within a POM.xml file  
    
    
    
    
  <dependencies>  
  <dependency>  
  <groupId>commons-lang</groupId>  
  <artifactId>commons-lang</artifactId>  
  <version>2.1</version>  
  </dependency>  
  </dependencies>
* The above example is of the commons-lang library which is a library from apache.
* Look at the screenshot for the POM.xml with the dependency section.

GOALS :

* We have a clean goal – deletes the target directory and any of the generated resources.
* Compile- compile all the source code , generate any files like web services , copy our resources into our classes directory underneath our target.
* Package – Runs the compile command first , runs any tests , packages the app based on the packaging type that we have specified.
* Install – Runs the package command first and then install in the local repository , By default C:/Users/Home Directory/.m2
* Deploy - It doesn’t mean deploy it to an appServer , It runs an install command and deploy it to an corporate repository , like a file share.

GOALS DEMO :

* Lets go through a small demo now to illustrate the things we looked at so far.
* Open the HelloWorld application and open up the pom.xml file
* Add a dependency section  
  <dependencies>  
  <dependency>  
  <groupId>commons-lang</groupId>  
  <artifactId>commons-lang</artifactId>  
  <version>2.1</version>  
  </dependency>  
  </dependencies>
* Open the command prompt.
* Mvn clean  
  mvn compile  
  mvn clean package – I can daisy chain my goals , It will delete my target directory first , then compile and package based on the packaging method given in the POM.xml as JAR  
  mvn install – to copy it to a local repository C:\Users\STG Training\.m2\repository\com\pluralsight\HelloWorld\1.0-SNAPSHOT\HelloWorld-1.0-SNAPSHOT.jar

LOCAL REPO STRUCTURE :

* So once we ran an install command , or deployed to our remote server , Maven is going to go and store it in a repository.
* Since we ran mvn install , it stored it in the local repository.
* Its notated as C:\Users\<username>\.m2\repository
* Its generally the default.
* Stores artifacts using the information that you provided for artifactId , groupId , version.
* It uses all of the above combined to make the folder or directory structure to store the final artifact packaged as one of the packaging methods provided in the POM.xml
* It stores it as groupId\artifactId\version\artifactId-version.jar
* It avoids duplication by copying it in every project and storing it in your SCM.
* We don’t have version conflicts or such and we don’t need to copy it for every project that we develop. Its all taken from a common place which is our local repository.

DEFAULTS :

* The defaults that maven has to offer : How do I override the defaults ?
* Inside the build section we can override those defaults.
* We will see one or two examples from the build section , nothing super details , we will cover them in plugins module.
* We will change the artifact name of the JAR that gets built.

BUILD SECTION DEMO :

* To override the default name of our artifact to be used , add the following
* Sometimes when building a WAR file , it will assume the name of the JAR file that we are building for our application as the URL when deployed on Tomcat server. We don’t want that to happen.  
  <build>  
  <finalName>foo</finalName> -- Its one of the defaults that I can override.  
  <build>
* Go to command line
* Mvn clean package -> Build my application and build it with the target name with foo.jar
* Go into the target directory and check.
* Go to the pom.xml and reverse that ie remove the build section that we created now
* Go to the command line and then again run the command mvn clean package
* It again renamed it as HelloWorld-1.0-SNAPSHOT.jar

SUMMARY :

* We saw where our source code goes into our application
* Everything gets compiled into our target directory.
* Four major parts of the POM.xml
* Goals and chaining goals.
* Basic example of a dependency
* Where things are stored in the local repository.
* We also looked at how we can override the default behavior.