MAVEN PLUGINS

INTRODUCTION :

* In this module , we will cover plugins. Plugins are what Maven uses to build or package our application outside of just downloading and storing artifacts for us.
* We are going to see what GOALS are.
* We will also discuss PHASES cause GOALS are tied to PHASES.
* We will go over the compiler plugin.
* We will see the jar plugin
* We will also see the sources and javaDoc plugins

GOALS AND PHASES :

* Compile , clean , package ,test ,deploy , install all come under GOALS.
* The above goals are defined inside the SuperPOM.
* The SuperPOM has these goals inside of it which are transferred to your effective POM.
* We can configure all of these plugins inside our POM to tune and tweak whatever they can do based on what we want.
* The example : Clean-plugin  
  <plugin>  
  <artifactId>maven-clean-plugin</artifactId>  
  <version>2.4.1</version>  
  <executions>  
  <execution>  
  <id>default-clean</id>  
  <phase>clean</phase>  
  <goals>  
  <goal>clean</goal>  
  </goals>  
  </execution>  
  </executions>  
  </plugin>
* If we want to force it to run everytime we want to compile , you can change which phase its tied to , so goals are always tied to a phase.
* What are the different phases that are available to tie your plugin to?   
  validate – validates the project is correct and all the necessary information is available  
  all the plugins that it needs, all the artifacts that are provided , all of our structures in place , permission  
  compile – where we compile our source code at , this is not when testing code gets compiled.  
  test – tests the compiled source code , all sources are in line with test packages  
  package – packages all of our code in the packaging style provided in the POM.xml  
  integration-test phase – It is new to Maven 3 , it deploys and runs integration test.  
  verify – runs check against our package to check its good and meets all the requirement before we install it to our local repo or the remote repo  
  install – install will package and put it in the local repo  
  deploy – It will copy the final package to a remote repo.

COMPILER PLUGIN AND DEMO :

* It’s the plugin that we use to compile our source code and test code.
* It runs in different phases.
* <http://maven.apache.org/plugins/maven-compiler-plugin/index.html> to view some more information about the plugin.
* Invoke javac , it does so by setting the classpath with the dependencies.
* Default to Java 1.5 regardless of what JDK is installed.
* Configuration section allows customization.  
  <plugin>  
  <groupId>org.apache.maven.plugins</groupId>  
  <artifactId>maven-compiler-plugin</artifactId>  
  <version>2.5.1</version>  
  <configuration>  
  <fork>true</fork> 🡪 To work on its own thread  
  <meminitial>128m</meminitial>  
  <maxmem>512m</maxmem>  
  <source>1.7</source>  
  <target>1.7</target>  
  </configuration>  
  </plugin>
* We have our groupId , artifactId and version.
* We also have specified the min and max memory on the heap should be.
* We also have our source and target to tell it to use a different target for our JAVA version.
* We will look at the demo :   
  Go to POM.xml  
  Define a <build> section   
  Add a <plugins> section and then add the above XML code for compiler plugin.  
  Once everything is added , do a Maven -> Update Project

THE JAR PLUGIN & DEMO :

* It is used to package our code into a JAR file.
* <http://maven.apache.org/plugins/maven-jar-plugin/index.html>
* It is tied to the package phase of our build life cycle.
* Configuration section allows us customization   
  We can include or exclude   
  It will also build our Manifest for us
* An example is given below :   
    
    
    
  <build>  
  <plugins>  
  <plugin>  
  <groupId>org.apache.maven.plugins</groupId>  
  <artifactId>maven-jar-plugin</artifactId>  
  <version>2.4</version>  
  <configuration>  
  <useDefaultManifestFile>true</useDefaultManifestFile>  
  </configuration>  
  </plugin>  
  </plugins>  
  </build>
* The manifest file , will take in our name , organization name and build that manifest file with some descriptive information.
* Open up POM.xml
* Add the above code for the JAR plugin.
* Save
* Go to the command prompt  
  mvn compile  
  Inside the target directory there is classes directory within which we have our META-INF which contains the MANIFEST.MF file.

THE SOURCE PLUGIN & DEMO :

* The source plugin can confuse people because they tend to think it has got something to do with our source code.
* It is used to attach our source code to a JAR
* It generates our source code for us so that we can upload it to a repository.
* <http://maven.apache.org/plugins/maven-source-plugin/index.html>
* It is tied by default to the package phase.
* Often overridden to a later phase , install or deploy
* The plugin code is :  
  <build>  
  <plugins>  
  <plugin>  
  <groupId>org.apache.maven.plugins</groupId>  
  <artifactId>maven-source-plugin</artifactId>  
  <version>2.2.1</version>  
  <executions>  
  <execution>  
  <id>attach-sources</id>  
  <phase>verify</phase>  
  <goals>  
  <goal>jar</goal>  
  </goals>   
  </execution>  
  </executions>  
  </plugin>  
  </plugins>  
  </build>
* The verify phase is ran before the install and deploy phase and I have tied it to the goal JAR.
* So therefore when it goes to the verify phase or we jar our application it will go and attach the source code for us.
* Go to the POM.xml file
* Add the above code for source plugin
* Save
* Before u have seen the configuration section where we can tune and tweak what the application can do accordingly , but here we are going to use an executions section that tells when to do.
* Go to the command prompt and run the following  
  mvn package  
  It didn’t actually run our attach sources yet.  
  mvn install  
  Now we can see that it attached our sources and we can see the ID <attach-sources>
* If we go to the springsource.org to see the snapshots , we saw the sources and the javadocs separately , so that when we have problems , we can get the source code and java docs separately to do some debugging and understanding.

JAVA DOC PLUGIN AND DEMO :

* The JavaDoc plugin is almost identical to the source plugin , in the sense that its going to take our java docs and attach it to a jar file to upload to our repository.
* <http://maven.apache.org/plugins/maven-javadoc-plugin/index.html>
* Its tied to the package phase. It can hinder performance if its running a lot , so we will override it to the verify phase.
* Usually we use the defaults , but customization is available.
* The following code describes the JavaDoc plugin  
  <build>  
  <plugins>  
  <plugin>  
  <groupId>org.apache.maven.plugins</groupId>  
  <artifactId>maven-javadoc-plugin</artifactId>  
  <version>2.9</version>  
  <executions>  
  <execution>  
  <id>attach-javadocs</id>  
  <phase>verify</phase>  
  <goals>  
  <goal>jar</goal>  
  </goals>   
  </execution>  
  </executions>  
  </plugin>  
  </plugins>  
  </build>
* Its defined almost identical to the source plugin.
* Execution section is used instead of configuration section.
* Open up POM.xml
* Add the above code
* We are telling it , when we want it to run.
* Switch over to the command line and run the following  
  mvn install  
  It generated our sources and java docs and attached it to the JAR file.  
  We can see that we have our source code , sources and javadocs getting uploaded to our local repo.
* Check the local repo as to where it is placed.

SUMMARY :

* In this module we learned that goals are just configured plugin , we can accept the default coming out of the Super POM.
* We saw compiler plugin , jar plugin and generate manifest file for us, sources plugin , Javadoc plugin and install all to our corporate repository.
* We can use the above to check where our code breaks and can use it for debugging.