Java means DURGA SOFT..

JAVA TOOLS

Material

maven

Maven2

Index

Invoking Maven

General Syntax:
mvn*plugin*:*target*[-D*option1* -D*option2* dots]
mvn help
mvn -X ...
Prints help debugging output, very useful to diagnose

Creating a new Project (jar)

mvnarchetype:create -DgroupId=*Artifact Group*
-DartifactId=*Artifact ID*
Example:
mvnarchetype:create -DgroupId=de.focusdv.bcs
-DartifactId=new-app
Creates a new Project Directory *new-app* with package structure *de.focusdv.bcs.*
Name of the packaged jar will be new-app-*version*.jar

Creating a new Project (war)

mvnarchetype:create
-DgroupId=*Artifact Group*
-DartifactId=*Artifact ID*
-DarchetypeArtifactId=maven-archetype-webapp
Example:
mvnarchetype:create
-DgroupId=de.focusdv.bcs
-DartifactId=new-webapp
-DarchetypeArtifactId=maven-archetype-webapp
Creates a new Directory *new-webapp*with package structure *de.focusdv.bcs.*
Name of the packaged war will be new-app-*version.*war

Standard Project Structure

directory description

/new-app/pom.xml maven2 project file
/new-app/src/ Sources
/new-app/src/main/java/ Java source tree
/new-app/src/test/java/ Java unit tests
/new-app/src/main/resources/ Java classpath resources

/new-app/src/test/resources/ Resources for unit-tests
/new-app/target/classes/ compiles classes
/new-app/target/test-classes/ compiles test classes
/new-app/target/dots other plugins' output
/newwebapp/
src/main/webapp root of webapp

Compiling

mvn compile

Running Unit Tests / Code Coverage

mvn test

compiles and runs unit tests

mvn clean cobertura:cobertura

generates a code-coverage report for the tests. It only works, if the pom.xml is configured as follows:

```
</project>
…
<build>
<plugins>
…
<plugin>
<groupId>org.codehaus.mojo</groupId>
<artifactId>cobertura-maven-plugin</artifactId>
<executions>
<execution>
<goals>
<goal>clean</goal>
</goals>
</execution>
</executions>
</plugin>
…
</plugins>
</build>
…
<reporting>
```

```
<plugins>
<plugin>
<groupId>org.codehaus.mojo</groupId>
<artifactId>cobertura-maven-plugin</artifactId>
</plugin>
</plugins>
</reporting>
…
</project>
```

Packaging (jar, war)

mvn clean package
compiles, runs unit tests and packages the artifact (clean makes sure there are no unwanted files in
the package)

Installing Artifact in Local Repository

mvn clean install
compiles, runs unit tests, packages and installs the artifact in the local repository. (*User Home Directory*/.m2/repository/)

Installing 3rdParty jar in local Repository

```
mvninstall:install-file -Dfile=foo.jar
-DgroupId=org.foosoft -DartifactId=foo
-Dversion=1.2.3 -Dpackaging=jar
```

Cleaning Up

mvn clean

Creating Eclipse Project Structure

mvneclipse:eclipse
If using the eclipse plugin from update-site
http://m2eclipse.codehaus.org
remove the generated dependencies from project.

Maven Project file (pom.xml)

Minimal pom.xml is created with
mvnarchetype:create

(see above).

Adding Dependencies

```
<project>
…
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>org.springframework</groupId>
<artifactId>spring</artifactId>
<version>1.2.6</version>
</dependency>
…
</dependencies>
```

Because of ,junit will not be included in final packaging.

Adding Developers

```
<project>
…
<developers>
<developer>
<id>Baier</id>
<name>Hans Baier</name>
<email>hans.baier::at::focus-dv.de</email>
<organization>focus DV GmbH</organization>
<roles>
<role>Developer</role>
</roles>
</developer>
…
</developers>
```

Setting Compiler Version

```
<project>
…
<build>
<plugins>
<plugin>
<artifactId>maven-compiler-plugin</artifactId>
<configuration>
<source>1.5</source>
<target>1.5</target>
</configuration>
</plugin>
…
</plugins>

</build>
```

Assemblies and Profiles

Creating Assemblies

To package the artifact use the following lines in the .pom-file:

```
<plugin>
<artifactId>maven-assembly-plugin</artifactId>
<configuration>
<descriptors>
<descriptor>src/main/assembly/foo-dep.xml</descriptor>
<descriptor>src/main/assembly/foo.xml</descriptor>
</descriptors>
</configuration>
</plugin>
```


src/main/assembly is the maven standard directory for assemblies.
The first assembly descriptor packages all dependencies into one jar:

```
<assembly>
<id>dep</id>
<formats>
```

```
<format>jar</format>
</formats>
<includeBaseDirectory>false</includeBaseDirectory>
<dependencySets>
<dependencySet>
<outputDirectory></outputDirectory>
<unpack>true</unpack>
<scope>runtime</scope>
<excludes>
<exclude>junit:junit</exclude>
</excludes>
</dependencySet>
</dependencySets>
</assembly>
```

The second descriptor packages the program:

```
<assembly>
<id>bin</id>
<formats>
<format>zip</format>
</formats>
<fileSets>
<fileSet>
<directory>src/main/assembly/files</directory>
<outputDirectory></outputDirectory>
<includes>
<include>**/*.bat</include>
<include>**/native/**</include>
<include>**/*.properties</include>
</includes>
</fileSet>
<fileSet>
<directory>target</directory>
<outputDirectory></outputDirectory>
<includes>
<include>*.jar</include>
</includes>
</fileSet>
</fileSets>
</assembly>
```

Supplementary files in this example are in src/main/assembly/files.

This includes the program starter (.bat), native libraries (/native) and Properties files.

Packaging is invoked by:

mvnassembly:assembly

Using Profiles

Profiles enable different versions of a project to be build, or adapting to different environments by
an option on the command line. Profiles can modify almost all dependencies, plugins and settings in
the*pom.xml*. In *cockpit-model* they are used to generate a restricted demo-version and a releaseversion
like that:

```
<profiles>
<profile>
<id>release-profile</id>
<dependencies>
<dependency>
<groupId>swt</groupId>
<artifactId>swt-win32</artifactId>
<version>3.2.1</version>
</dependency>
</dependencies>
<build>
<filters>
<filter>src/main/filters/releaseVersion.properties</filter>
</filters>
</build>
</profile>
<profile>
<id>demo</id>
<dependencies>
<dependency>
<groupId>swt</groupId>
<artifactId>swt-win32</artifactId>
<version>3.2.1</version>
</dependency>
</dependencies>
<build>
<filters>
<filter>src/main/filters/demoVersion.properties</filter>
</filters>
</build>
</profile>
…
</profiles>
```

Here the *release-profile* uses the windows library of SWT (since our customers' platform is windows (like it or not...), and substitutes the resources files' placeholders with the variables in
releaseVersion.properties. The *demo*-profile is almost the same except it uses *demoVersion.properties*for filtering.

Usage:

mvn -Prelease-profile clean assembly:assembly
or
mvn -Pdemo clean assembly:assembly

Using Profiles by OS

In this example we want to use the Linux SWT Libraries on Linux and the Windows libs on Windows:

```
<profiles>
<profile>
<id>windows</id>
<activation>
<os>
<family>windows</family>
</os>
</activation>
<dependencies>
<dependency>
<groupId>swt</groupId>
<artifactId>swt-win32</artifactId>
<version>3.1.1</version>
</dependency>
</dependencies>
</profile>
<profile>
```

```
<id>unix</id>
<activation>
<os>
<family>unix</family>
</os>
</activation>
<dependencies>
<dependency>
<groupId>swt</groupId>
<artifactId>swt-linux-gtk</artifactId>
<version>3.1.1</version>
</dependency>
</dependencies>
</profile>
</profiles>
```


Versioning, Repositories and Releases

Setting Source Code Control System

```
<project>
…
<scm>
<developerConnection>
scm:svn:https://svnhost.net/svnroot/trunk/new-app
</developerConnection>
</scm>
<build>
<plugins>
<plugin>
<artifactId>maven-release-plugin</artifactId>
<configuration>
<tagBase>
https://svnhost.net/svnroot/tags
</tagBase>
</configuration>
</plugin>
…
</plugins>
</build>
```

Versioning

Keep the Verision of your POM artifact in the form *version*-SNAPSHOT until you release.

Mavens release plugin then removes the -SNAPSHOT suffix.

Using internal Repositories

This assumes that a machine *myhost*exists with a configured and running Web-Server and SSHServer

```
<repositories>
<repository>
<id>focus-repository</id>
<name>Focus BCS Repository</name>
<url>http://myhost/mvn/repository</url>
</repository>
</repositories>
<distributionManagement>
<repository>
<id>focus-repository</id>
<name>Focus BCS Repository</name>
<url>scp://myhost/var/www/mvn/repository/</url>
</repository>
</distributionManagement>
```

Installing Artifact in Remote Repository

mvn clean deploy
compiles, runs unit tests, packages and installs the artifact in the remote repository.

Install 3rdParty jar to Remote Repository

```
mvndeploy:deploy-file -DgroupId=commons-collections
-DartifactId=collections-generic -Dversion=4.0
-Dpackaging=jar -Dfile=collections-generic-4.0.jar
-DrepositoryId=focus-repository
-Durl=scp://host/home/mvn/public_html/repository
```

Preparing Releases

Make sure, the SCM settings in the POM are correct and all changes are committed to the SCM.
Then execute
mvn -Dusername=*USER* -Dpassword=*PASS* release:prepare
Before issuing the above command use it with *-DdryRun=true* first
tags in configured build profiles in the pom.xml

Performing Releases

mvn -P *profile* -Drelease:perform
Checks out the released version from tag in repository, builds, tests, packages and installs package,
javadoc and sources in repository. As preparing the release removes activation tags from build
profiles, it is necessary to supply the profile or the release will fail.

Web-Development

Integration-Test with tomcat

```
<project>
…
<build>
<plugins>
…
<plugin>
<groupId>org.codehaus.cargo</groupId>
<artifactId>cargo-maven2-plugin</artifactId>
<executions>
<execution>
<id>tomcat-execution</id>
<phase>package</phase>
<goals>
<goal>start</goal>
</goals>
<configuration>
<wait>true</wait>
<container>
<containerId>tomcat5x</containerId>
<zipUrlInstaller>
<url><http://www.apache.org/.../jakarta-tomcat.zip></url>
<installDir>${installDir}</installDir>
</zipUrlInstaller>
</container>
<configuration>
<dir>${project.build.directory}/tomcat5x/</dir>
</configuration>
</configuration>
```

```
</execution>
</executions>
</plugin>
</plugins>
</build>
```


Then execute in project directory:
mvn -X integration-test
The war-file will built, tested and packaged. Then tomcat will be downloaded, installed and started
with the war-file of the project deployed to the server.
If you want to use jetty4 (already embedded, fast startup) use:
mvncargo:start
(Press Ctrl-C to stop)

Online web-development with Jetty plugin

Add Maven-Plugin to pom.xml:

```
<plugins>
…
<plugin>
<groupId>org.mortbay.jetty</groupId>
<artifactId>maven-jetty6-plugin</artifactId>
<configuration>
<scanIntervalSeconds>10</scanIntervalSeconds>
</configuration>
</plugin>
…
</plugins>
```

Then run Jetty with
mvn jetty6:run

Online web-development and automatic deployment with tomcat plugin

Add Maven-Plugin to pom.xml:

```
<plugins>
…
<plugin>
<groupId>org.codehaus.mojo</groupId>
<artifactId>tomcat-maven-plugin</artifactId>
<configuration>
<url>http://192.168.129.36:8080/manager/html</url>
</configuration>
</plugin>
<plugin>
<groupId>org.codehaus.cargo</groupId>
<artifactId>cargo-maven2-plugin</artifactId>
</plugin>
…
</plugins>
…
<repositories>
<repository>
<id>codehaus</id>
<name>Codehaus maven repository</name>
<url>http://dist.codehaus.org/</url>
<layout>legacy</layout>
</repository>
…
</repositories>
```

Then run Tomcat with

mvntomcat:run

Deploy the war automatically with

mvntomcat:deploy

If already deployed, the webapp needs to be undeployed first:

mvntomcat:undeploy

Note that automatic deployment/undeployment only works without further configuration in *$MAVEN2_HOME/conf/settings.xml* if the managers username is admin with empty password


LEARN FROM EXPERTS ...
COMPLETE JAVA
CORE JAVA, ADV. JAVA, ORACLE, STRUTS, HIBERNATE, SPRING, WEB SERVICES,...
COMPLETE .NET
C#.NET, ASP.NET, SQL SERVER, MVC 5 & WCF
TESTING TOOLS
MANUAL + SELENIUM
ORACLE
D2K
MSBI
SHARE POINT
HADOOP
ANDROID
C, C++, DS, UNIX
CRT & APTITUDE TRAINING
AN ISO 9001:2008 CERTIFIED
DURGA
Software Solutions®
202, 2nd Floor, HUDA Maitrivanam,
Ameerpet, Hyd. Ph: 040-64512786,
9246212143, 8096969696
www.durgasoft.com