Maven 2 for developers
- Georges Polyzois

Background

Coming from a project using Maven 1 for over a year now - it was a true revelation starting to use Maven 2 and leverage the multi project build functionality which now is built into the core of Maven. In Maven 1 – if you were using a multi project build you had quite a task figuring out how to set it up. Maven 1 did not help much, besides providing the reactor plug in and adhoc configuration in all kinds of files. What if you wanted a central point for dependency version for common libraries like log4j? Well one way to do this would be to configure an xml extension in all your project.xml and maven.xml files to be able to leverage a global file with version numbers for common dependencies like e.g. Log4j.
Another problem with Maven 1 was the lack of speed with multi project builds – especially the traversal part of the reactor plugin to find project.xml files recursively. In Maven 2 you specify these explicitly (you probably could do that in Maven 1 too).

News in Maven 2

Transitive dependencies

Lazy loaded dependencies or what Maven refers to as transitive dependencies, are dependencies with other dependencies in turn. With transitive dependencies a pom.xml file is published along with the projects artifact (typically a jar file with version number). E.g. say you have a project for modelling a rich domain model. Then you probably would have a dependency to Hibernate (or some other orm tool) and would therefore need the hibernate jar artifact along with all other Hibernate dependent artifacts, like commons-logging. The dependencies Hibernate has in turn are published to a replicated remote repository like (http://www.ibiblio.org/maven2/org/hibernate/hibernate/3.1rc2/) specifying its dependencies. In Hibernates case the pom.xml file would look like in the table below. If you do find a public dependency that is incorrect then go ahead and file report at http://jira.codehaus.org/browse/MEV. Maven 2 will report any cyclic dependencies it will found for your transitive dependencies.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate</artifactId>
  <version>3.0.5</version>
  <dependencies>
    <dependency>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
      <version>1.0.4</version>
    </dependency>
    <dependency>
      <groupId>ant</groupId>
      <artifactId>ant</artifactId>
      <version>1.6.3</version>
      <scope>provided</scope>
    </dependency>
...
```

Table: Transitive dependency for Hibernate at http://www.ibiblio.org/maven2/org/hibernate/hibernate/3.0.5/hibernate-3.0.5.pom

Build phases

Maven 2 introduces a clean build life cycle with well defined build steps outlined in the table below. The steps are used in a plugin which register itself with annotation to be run when a certain phase is to be run by Maven.

Build phase	Description
validate	validates the project pom.xml and sees to that all necessary information is available.
generate-sources	generates any source code for inclusion in compilation, e.g. Corba idl or XSD generation.
process-sources	process the source code, for example to filter any values.
generate-resources	generate resources for inclusion in the package.
process-resources	copy and process the resources into the destination directory, ready for packaging.
compile	compile the source code of the project.
process-classes	post-process the generated files from compilation, for example to do byte code enhancement on Java classes.
generate-test-sources	generate any test source code for inclusion in compilation.
process-test-sources	process the test source code, for example to filter any values.
generate-test-resources	create resources for testing.
process-test-resources	copy and process the resources into the test destination directory.
test-compile	compile the test source code into the test destination directory
test	run tests using a suitable unit testing framework. These tests should not require the code be packaged or deployed.
package	take the compiled code and package it in its distributable format, such as a JAR.
integration-test	process and deploy the package if necessary into an environment where integration tests can be run.
verify	run any checks to verify the package is valid and meets quality criteria.
install	install the package into the local repository, for use as a dependency in other projects locally.
deploy	done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.

Table: Build life cycle phases

```
package org.apache.maven.plugins.jacorb;

import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoExecutionException;
import org.apache.commons.io.FileUtils;

import java.util.ArrayList;
import java.io.File;
import java.io.IOException;


/**
 * A code generator which will be invoked to generate sources for an
 * corba idl file or xml schema file.
 *
 * @goal  generate
 * @phase generate-sources    injected when maven hits build process phase
 *                            generate-sources
 * @description Build a .java files from idl, compile them and produce a jar.
 */
public class JacorbPlugin  extends AbstractMojo
{
    /**
     * @parameter expression="${project.version}"
     * @required
     * @readonly
     */
    private String version;

    /**
     * @parameter expression="${project.build.outputDirectory}"
     * @required
     * @readonly
     */
    private String outputDirectory;
    ...
    public void execute() throws MojoExecutionException
    {
            // Generate source
    }
}
```
Table: Plugin which will be invoked on generate-source

Dependency scoping

The dependencies you have in your pom.xml file can be scoped, meaning that they are based on what scope the build has. With enterprise applications there is a need to distinguish libraries for compiling, running and testing so if you use JUnit (a testing library) there is no need to provide that library for a runtime environments classpath. Dependency scopes help you solve that and is something introduced in Maven 2.

Dependency scope	Description
compile	A compile scoped dependency is available on all classpaths.
test	A test scoped dependency is available only on the test compilation and execution classpaths.
runtime	The dependency is not needed for the compile or test classpath but rather at runtime.
provided	Provided means that it is in the runtime context for this project this dependency will be provided, e.g. By a J2EE container like Geronimo. Available in the compile classpath but is not a transitive dependency.
system	Always available and never looked up. An example would be: `<dependencies>` `<dependency>` `<groupId>javax.sql</groupId>` `<artifactId>jdbc-stdext</artifactId>` `<version>2.0</version>` `<scope>system</scope>` `<systemPath>${java.home}/lib/rt.jar</systemPath>` `</dependency>` `</dependencies>`

Table: Dependencies

Profiles

The local repository is where Maven stores dependent artifacts but also meta data for a build to What if you would like a continuous build on a test server to use different configuration than in your development environment? Or you want a QA server to deploy and perform integration tests using a J2EE container with the integration plugin? Maven 2 lets you do that using profiles.
There are three ways of specifying profiles:

1. Using the Maven 2 settings file (settings.xml)
2. Using a file in the basedir of your project (profiles.xml)
3. Using a xml element called activeProfiles in the POM itself.

The first and second choice of specifying a profile is less portable and has restrictions on what you can alter, namely artifact repositories, plugin repositories and free-form properties. Meta data about the build is only used from the pom.xml, so using any of these two will not externalize the information in these files.
The third option of specifying a profile within your projects pom.xml file provides fas more options for modifications – you may alter things like repositories, pluginRepositories, dependencies, plugins, modules, reporting, dependencyManagement, distributionManagement etc. The pom.xml file build data is externalized during a build and deployed to the repository. An example of using a the pom.xml profile section to set the path of an application server.
An example of the two main different ways of specifying a profile for a plugin could look like below.

```
<!-- Example from Maven site -->
<build>
  <plugins>
    <plugin>
      <groupId>org.myco.plugins</groupId>
      <artifactId>spiffy-integrationTest-plugin</artifactId>
      <version>1.0</version>
      <configuration>
        <appserverHome>$\{appserver.home\}</appserverHome>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Table: Our projects pom.xml build section.

Using the pom.xml style	`<!-- Example from Maven site -->`
	`<profiles>`
	`    <profile>`
	`        <id>appserverConfig-dev</id>`
	`        <!--`
	`        | This profile will be triggered by the system property env=test`
	`        |     mvn -Denv=test integration-test`
	`        | but using this would fail`
	`        -->`
	`        <activation>`
	`            <property>`
	`                <name>env</name>`
	`                <value>test</value>`
	`            </property>`
	`        </activation>`
	`        <properties>`
	`            <appserver.home>/path/to/dev/appserver</appserver.home>`
	`        </properties>`
	`    </profile>`
	`</profiles>`

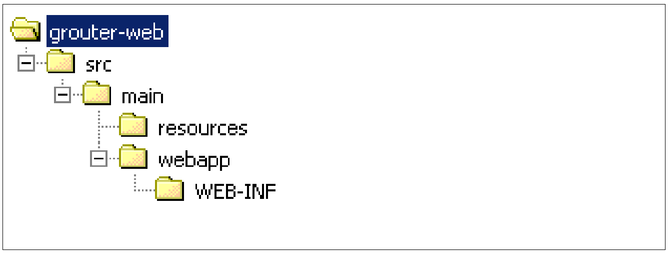
Using the settings.xml style	`<!-- Example from Maven site -->`
	`<profiles>`
	`  <profile>`
	`    <id>appserverConfig-dev</id>`
	`    <properties>`
	`      <appserver.home>/path/to/appserver/deploy/dir</appserver.home>`
	`    </properties>`
	`  </profile>`
	`</profiles>`
	`...`
	`<!--`
	` | This is how this profile is being triggered`
	`-->`
	`<activeProfiles>`
	`  <activeProfile>appserverConfig-dev</activeProfile>`
	`</activeProfiles>`

Table: Using two different ways of specifying a profile.

Archetypes or project templates

To speed up a new project something called archetypes – or project templates - have been added to Maven 2. Maven 2 helps you create a single type project structure (a project with one pom.xml) and a pom.xml file with the basic must have xml tags inserted. What. if you want to create a web type project which will produce a war artifact? Let Maven 2 to do this for you using:

```
$ mvn archetype:create -DgroupId=org.grouter -DartifactId=grouter-web –
DpackageName=org.grouter.client.web  -DarchetypeArtifactId=maven-archetype-webapp
```



```
grouter-web
  src
    main
      resources
      webapp
        WEB-INF
```

Table: Project structure for a web type archetype.

The generated pom.xml file will look like below:

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
v4_0_0.xsd">
   <modelVersion>4.0.0</modelVersion>
   <groupId>org.grouter</groupId>
   <artifactId>grouter-web</artifactId>
   <packaging>war</packaging>
   <version>1.0-SNAPSHOT</version>
   <name>Maven Webapp Archetype</name>
   <url>http://maven.apache.org</url>
   <dependencies>
     <dependency>
       <groupId>junit</groupId>
       <artifactId>junit</artifactId>
       <version>3.8.1</version>
       <scope>test</scope>
     </dependency>
   </dependencies>
   <build>
     <finalName>grouter-web</finalName>
   </build>
</project>
```

Table: A pom.xml structure for a web type archetype.

What archetypes exists in Maven 2 – at http://www.ibiblio.org/maven2/org/apache/maven/ under archetypes you will find the ones publicly available. At the time of this writing the archetypes I found were web-app, site, mojo, marmelade and quickstart. If you want some exotic archetype then Maven 2 lets you define your own archetypes using an archetype descriptor file called archetype.xml. Powerful indeed!!

Plugins

What if you wanted to update a plugin to the latest version? With Maven 2 you can do this if you let out the version number of the plugin you are using. Maven 2 will then check on a daily basis for a new for the plugin and download and installing it on the fly. The update interval can be set to always, daily, interval or never. Well what if you want to reproduce a build using a certain version of a plugin? Plugins that are related directly to a build are specified in the pom.xml file and can of course be versioned – the build process will then use the specified version regardless of any other version known. This transparency makes it easier to distribute a project to people wanting to build your project without having to install the latest plugins. Maven 2 actually comes with no plugins at all installed and uses a transparent discovery process to download the plugins needed. Also in Maven 1 the configuration of a plugin was made through the project.properties file, in Maven 2 the properties are declared in the pom.xml file where the plugin is declared as a dependency – se below:

```xml
    <build>
        <plugins>

            <!--
            | Plugin dependencies. If you let out the version number
            | of the plugin you are using then Maven 2 will check
            | on a daily basis for a new plugin and download and
            | install it on the fly. The update intervall can be set to
            | always, daily, interval or never. Of course if you want
            | your builds to be reproducable then do specify a version
            | number.
            -->
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <!-- version specify for reproducable builds/ -->
                <configuration>
                    <source>1.5</source>
                    <target>1.5</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-war-plugin</artifactId>
                <configuration>
                    <warSourceDirectory >src/webapp</warSourceDirectory >
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-site-plugin</artifactId>
                <configuration>
                    <locales>en</locales>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

Table: Plugin dependencies with configuration parameters

The clean goal in Maven 1 required that you had all dependencies set up before doing an actual clean – this is no longer the case in Maven 2.

Faster and smaller core

Maven2 has been rewritten from the ground up in parallel with the Maven 1 code base. It is now a faster and smaller core in Maven 2, making the distribution much smaller (dependencies are downloaded on the fly on first run). This can make it harder if you are working in an environment where you are not allowed direct access to Internet (there is of course ways around this).

Some important goals

When you start using Maven 2 the one thing you will notice directly is that the command line argument has changed from Maven to mvn. Some useful mvn command are listed below:

Maven goal	Description
mvn clean:clean	Clean all artifacts and intermediary files created by Maven
mvn idea:idea	Generate project files for the IntelliJ IDEA IDE
mvn eclipse:eclipse	Generate project files for the Eclipse IDE
mvn clover:clover	Generate a coverage report for the project
mvn checkstyle:checkstyle	Generate a checkstyle report for the project
mvn site:site	Generate a site
mvn deploy	Deploy artifacts

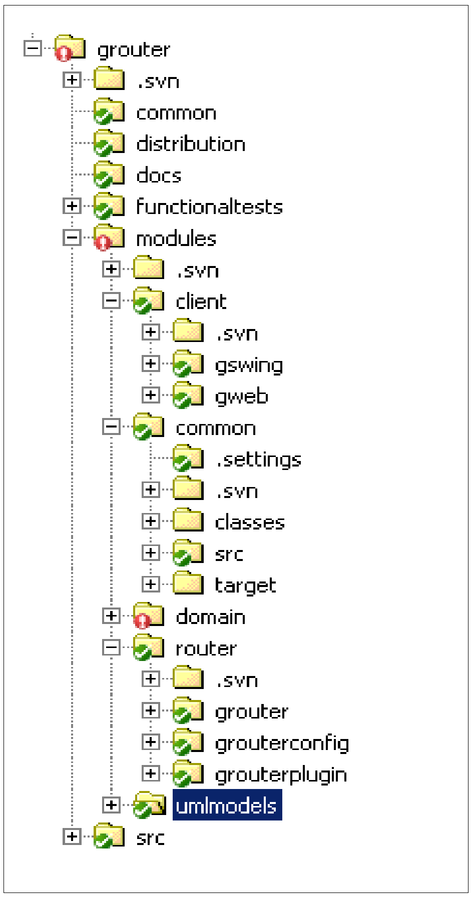
Grouter - introduction

To spice things up we will now do a "real" world project – or at least a project with more than one artifact. The complexity increases a bit but it is far from rocket science. The aim is to give some insight on how things are coupled between projects in a multi project build.

At my spare time I do some coding in an open source project called grouter, which if you are interested is registered at BerliOS (http://developer.berlios.de/projects/grouter/). It is a work in progress and keeps me busy from time to time.

Grouter – project structure

The grouter project serves as a good starting point for a Maven 2 multi project build. Below is the project structure outlined which will form the basis for further discussions around Maven 2.



```
grouter
    .svn
    common
    distribution
    docs
    functionaltests
    modules
        .svn
        client
            .svn
            gswing
            gweb
        common
            .settings
            .svn
            classes
            src
            target
        domain
        router
            .svn
            grouter
            grouterconfig
            grouterplugin
        umlmodels
    src
```

Table: Project structure for grouter

In the root folder – grouter – we will create a project.xml file which will serve as the parent of all child modules for this project (see the table below for the pom.xml file contents). The modules folder located directly under grouter contains all modules (i.e. Maven artifacts) built for this project.

The pom.xml file for our parent project is outlined in the table below – it is heavily commented and should be self explanatory.

C:\gepo\grouter\grouter\pom.xml

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <!--
    | This is the version for the POM model Maven 2 uses, it is something
    | you will probably never change (unless Maven decides to upgrade
    | the version number - but then you probably have to look through
    | the whole pom.xml file)
    -->
    <modelVersion>4.0.0</modelVersion>
    <!--
    | The groupId element is a unique identifier of the organization
    | or group that created the project. This value is used to group
    | all jars for a project into directories. E.g. in this example
    | GroupId will generate a folder structure of $MVNREPOS/org/grouter
    | on a mvn install. You will notice that many artifacts at ibiblio
    | do not use this type of structure (org.something)- instead they
    | place the groupId(folder) in the root structure. Maven encourages
    | you to use your reverse www address as a starting point for
    | this element, so a e.g. org.apache.jakarta.commons would be a
    | good candidate.
    -->
    <groupId>org.grouter</groupId>
    <!--
    | This is the name of the artifact which will be produced and put
    | under the groupId structure discussed above. It is a unique
    | identifier which identifies this artifact under the specified
    | groupId and will produce something like e.g.
    | grouter-1.0-SNAPSHOT.jar or on a release grouter-1.0.o.jar
    -->
    <artifactId>grouter</artifactId>
    <!--
    | The packaging element tells you what type of artifact will be
    | produced. There are several type of artifacts supported, like
    | war and ear type artifacts. The packaging element can also play a
    | part in customizing the build process.
    -->
    <packaging>pom</packaging>
    <!--
    | The version elelent is appended to the artifactId. During development
    | you should use a SNAPSHOT dependency. What it means is that
    | if you are working towards a 2.0 release then as a developer
    | you enter 2.0-SNAPSHOT
    -->
    <version>1.0-SNAPSHOT</version>
    <!--
    | Description will go to generated documentation reports.
    -->
    <description>GRouter is aimed to provide routing mechanism for messages
    with destinations including jms, ejb, file, email etc.</description>
    <!--
    | A name used for documentation.
    -->
    <name>grouter parent project</name>
    <!--
    | Url used for documentation.
    -->
    <url>http://grouter.berlios.de/</url>


    <!--
    | The developers element describe the committers to a project. Enter
    | all the project developers committing to the project.
    -->
    <developers>
        <developer>
            <id>gepo01</id>
            <name>Georges Polyzois</name>
            <email>gepo01 at yahoo dot not com</email>
            <url/>
            <organization/>
            <organizationUrl/>
            <roles/>
            <timezone/>
            <properties/>
        </developer>
    </developers>


    <!--
    | If you have any internal repository you will have to put them here.
    | This is great if e.g. you are working offline or you are producing
    | project specific plugins you want to distribute to developer in
    | the project transparently.
    -->
    <repositories>
        <repository>
            <id>maven2</id>
            <name>Danish Maven2 repository</name>
            <url>http://mirrors.sunsite.dk/maven2</url>
        </repository>
    </repositories>

    <!--
    | Put all global dependencies here instead of spreading them out in
    | different child modules - an oo way of handling dependencies.
    -->
    <dependencies>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>1.2.8</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>commons-lang</groupId>
            <artifactId>commons-lang</artifactId>
            <version>2.1</version>
        </dependency>
        <dependency>
            <groupId>commons-beanutils</groupId>
            <artifactId>commons-beanutils</artifactId>
            <version>1.7.0</version>
        </dependency>
        <dependency>
            <groupId>commons-collections</groupId>
            <artifactId>commons-collections</artifactId>
            <version>3.1</version>
        </dependency>
    </dependencies>


    <!--
    | Put semi-global dependencies here, i.e. dependencies which
    | should be inherited by some of the child modules/projects of this
    | parent, but not all.
    -->
    <dependencyManagement>
        <dependencies>
            <!--
            | Dependencies to J2EE spec - using geronimo generated
            | artifacts to get transparent builds without installing
            | the J2EE kit or pointing to your J2EE app jars.
            -->
            <dependency>
                <groupId>geronimo-spec</groupId>
                <artifactId>geronimo-spec-jms</artifactId>
                <version>1.1-rc4</version>
            </dependency>
            <dependency>
                <groupId>geronimo-spec</groupId>
                <artifactId>geronimo-spec-ejb</artifactId>
                <version>2.1-rc4</version>
            </dependency>
            <dependency>
                <groupId>geronimo-spec</groupId>
                <artifactId>geronimo-spec-j2ee-connector</artifactId>
                <version>1.5-rc4</version>
            </dependency>
            <dependency>
                <groupId>geronimo-spec</groupId>
                <artifactId>geronimo-spec-activation</artifactId>
                <version>1.0.2-rc4</version>
            </dependency>
            <dependency>
                <groupId>geronimo-spec</groupId>
                <artifactId>geronimo-spec-jta</artifactId>
                <version>1.0.1B-rc4</version>
            </dependency>
            <dependency>
                <groupId>org.hibernate</groupId>
                <artifactId>hibernate</artifactId>
                <version>3.1rc2</version>
                <!--
                | Excluding jars is a powerful feature. Here I know I
                | will not use swarmcache, so I do not need downloading
                | that dependency.
                -->
                <exclusions>
                    <exclusion>
                        <groupId>ant</groupId>
                        <artifactId>ant</artifactId>
                    </exclusion>
                    <exclusion>
                        <groupId>odmg</groupId>
                        <artifactId>odmg</artifactId>
                    </exclusion>
                    <exclusion>
                        <groupId>c3p0</groupId>
                        <artifactId>c3p0</artifactId>
                    </exclusion>
                    <exclusion>
                        <groupId>proxool</groupId>
                        <artifactId>proxool</artifactId>
                    </exclusion>
                    <exclusion>
                        <groupId>opensymphony</groupId>
                        <artifactId>oscache</artifactId>
                    </exclusion>
                    <exclusion>
                        <groupId>swarmcache</groupId>
                        <artifactId>swarmcache</artifactId>
                    </exclusion>
                    <exclusion>
                        <groupId>jboss</groupId>
                        <artifactId>jboss-cache</artifactId>
                    </exclusion>
                    <exclusion>
                        <groupId>javax.security</groupId>
                        <artifactId>jacc</artifactId>
                    </exclusion>
                    <exclusion>
                        <groupId>javax.transaction</groupId>
                        <artifactId>jta</artifactId>
                    </exclusion>
                </exclusions>
            </dependency>
            <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring</artifactId>
                <version>1.2.6</version>
                <scope>compile</scope>
                <exclusions>
                    <exclusion>
                        <groupId>javax.activation</groupId>
                        <artifactId>jta</artifactId>
                    </exclusion>
                    <exclusion>
                        <groupId>javax.resource</groupId>
                        <artifactId>connector</artifactId>
                    </exclusion>
                </exclusions>
            </dependency>
        </dependencies>
    </dependencyManagement>


    <!--
    | Contains different elements for distributing artifacts or site
    | generated html to remote repositories or web sites.
    -->
    <distributionManagement>
        <!--
        | The repostiory tag contain information needed for deploying
        | project generated artifacts to remote repository.
        | To deploy your artifacts to a server use:
        |    mvn deploy
        |
        | To make a release you can do
        |   mvn release:perform
        | This will make a release to local and remote repositories.
        | The release will include:
        |  1 - artifactid-version.jar
        |  2 -  artifactid-version-javadoc.jar
        |  3 - artifactid-version-source.jar
        |  4 - artifactid-version.pom
        -->
        <repository>
            <id>berlios</id>
            <name>Berlios Repository</name>
            <url>scp://grouter.berlios.org/dist</url>
        </repository>
        <!--
        | The site tag contain information needed for deploying
        | project generated html site to remote repository.
        | To be able to publish your html site use
        |     mvn site
        | followed by
        |     mvn site-deploy
        | The id element is a unique id stored in your settings.xml file
        | under element settings/servers/server which could look like
        | <server>
        |    <id>website</id>
        |     <username>repouser</username>
        |     <password>repopwd</password>
        | </server>
        | The url element specifies where to copy the artifact. Maven
        | currently only supports SSH, which in this case would copy
        | the site to host grouter.berlios.org in the path
        | /www/docs/project/
        -->
        <site>
            <id>website</id>
            <url>scp://grouter.berlios.org/www/docs/project/</url>
        </site>
    </distributionManagement>

    <!--
    | The scm element contain information of where to find a source
    | configuration management system where the project is resided.
    | The connection is used by Maven and the developerConnection
    | is used by developers to check out the svn trunk. The url
    | is provided to the browsable svn site.
    -->
    <scm>
        <connection>http://svn.berlios.de/svnroot/repos/grouter/trunk</connection>
<developerConnection>https://developername@svn.berlios.de/svnroot/repos/grouter/trunk</developerConnection>
        <url>http://svn.berlios.de/wsvn/grouter</url>
    </scm>

    <!--
    | This is where we put all child modules to be built. In Maven 1
    | the reactor plugin was used and you pointed to a root directory from w
    | where is would start a traversal for project.xml file - this is
    | no longer the case. Instead we explicitly point to all child modules
    | which makes the build process a lot more faster.
    -->
    <modules>
        <module>modules/common</module>
        <module>modules/domain</module>
        <!--module>modules/router/grouter</module>
        <module>modules/router/grouterconfig</module -->
    </modules>


    <!--
    | Used on report creation. Put in something meaningful - this is
    | your place in the sun ;-)
    -->
    <organization>
        <name>grouter</name>
        <url>http://developer.berlios.org/grouter</url>
    </organization>

    <!--
    | The build section is used to provide Maven with plugins and
    | configuration of those plugins. E.g. to change compiler
    | you will need to enter a plugin  maven-compiler-plugin
    | and vonfigure it to use source and target 1.5
    | You could also override the default project structure for
    | your project by entering source path directories etc.
    -->
    <build>
        <plugins>
            <plugin>
                <!--
                | The maven.surefire-plugin is the plugin used for
                | JUnit tests.
                -->
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <configuration>
                    <!--
                    | Dangerous since we will skip testing if true!
                    | Another way would be to use
                    |  mvn -Dmaven.test.skip=true install
                    -->
                    <skip>true</skip>
                    <includes>
                        <include implementation="java.lang.String">**/*.java</include>
                    </includes>
                    <!-- Manually exclude these tests... excludes>
                        <exclude implementation="java.lang.String">**/*Point*.java</exclude>
                    </excludes -->
                </configuration>
            </plugin>
            <!--
            | Plugin dependencies. If you let out the version number
            | of the plugin you are using then Maven 2 will then check
            | on a daily basis for a new plugin and download and
            | install it on the fly. The update interval can be set to
            | always, daily, interval or never. Of course if you want
            | your builds to be reproducible then do specify a version
            | number
            -->
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <!-- version specify for reproducible builds -->
                <configuration>
                    <source>1.5</source>
                    <target>1.5</target>
                </configuration>
            </plugin>
            <!--
            | For any J2EE web based applications we need to generate
            | a war file with a web.xml etc. This plugin helps us
            | with that.
            -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-war-plugin</artifactId>
                <configuration>
                    <warSourceDirectory >src/webapp</warSourceDirectory >
                </configuration>
            </plugin>
            <!--
            | The site plugin will produce a nice Maven looking
            | project site - with reports according to your
            | settings. It requires a project/src/site/site.xml file
            | in your projects.
            -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-site-plugin</artifactId>
                <configuration>
                    <locales>en</locales>
                </configuration>
            </plugin>
            <!--
            | To enbale more verbose output in the MANIFEST.MF class
            | of this jar we add som properties amnually to the
            | maven-ja-plugin
            | Output in the MANIFEST.MF file will look like below:
            | Manifest-Version: 1.0
            | Archiver-Version: Plexus Archiver
            | Created-By: Apache Maven
            | Built-By: geopol
            | Package: org.grouter
            | Build-Jdk: 1.5.0_03
            | Extension-Name: common
            | Specification-Title:
            | Specification-Vendor: grouter
            | Implementation-Vendor: grouter
            | Implementation-Title: common
            | Implementation-Version: 1.0-SNAPSHOT
            | Main-Class: none
            | mode: development
            | url: http://maven.apache.org
            -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <configuration>
                    <archive>
                        <manifest>
                            <mainClass>none</mainClass>
                            <packageName>org.grouter</packageName>
                        </manifest>
                        <manifestEntries>
                            <mode>development</mode>
                            <url>${pom.url}</url>
                        </manifestEntries>
                    </archive>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>

```
Table: Grand parent pom.xml

Next we will have a look at a child project of our grand parent pom.xml.It too should be self explanatory – most of the elements are inherited from your grand parent and the only important element worth mentioning is the parent.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <!--
    | Parent to this project is specified here. In Maven 1 we used
    | the extend element.
    -->
    <parent>
        <groupId>org.grouter</groupId>
        <artifactId>grouter</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <!--
    | See parent for explanations
    -->
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.grouter</groupId>
    <artifactId>domain</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>grouter domain artifact</name>
    <url>http://maven.apache.org</url>

    <!--
    | Dependencies specific to this project are entered here.
    | Common dependencies should go into the parent to simplify
    | the dependency section of this child project.
    | If no version is mentioned in this child's dependency section
    | the version is feteched from the parent's dependencyManagement
     | element.
     -->
    <dependencies>
        <dependency>
            <groupId>org.grouter</groupId>
            <artifactId>common</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring</artifactId>
        </dependency>
        <dependency>
            <groupId>hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>1.8.0.1</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>geronimo-spec</groupId>
            <artifactId>geronimo-spec-jta</artifactId>
        </dependency>
        <dependency>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate</artifactId>
        </dependency>
    </dependencies>

    <build>
        <resources>
            <!-- Bring on all mapping files for Hibernate -->
            <resource>
                <directory>${basedir}/src/main/java</directory>
            </resource>
            <resource>
                <directory>${basedir}/src/config</directory>
            </resource>
        </resources>
        <testResources>
            <!-- Bring on log4j.xml to test classes output folder -->
            <testResource>
                <directory>${basedir}/src/config</directory>
            </testResource>
        </testResources>
    </build>
</project>
```