Maven

The goal of this document is to introduce the Maven tool. This document just shows some of the functionalities of Maven. A complete guide about Maven can be found in http://maven.apache.org/.

Maven is a software project management tool. Maven is a Java tool, therefore you need to have Java installed in your machine in order to use this tool. Maven simplifies and standardizes the project build process. It handles compilation, distribution, documentation, team collaboration and other tasks seamlessly.

Using Maven means developers are not required to create or specify the build process themselves or to mention each and every configuration detail. Maven provides sensible default behavior for projects. When a Maven project is created, Maven creates a default project structure. Developer is only required to place files accordingly. The information about the project and various configuration details used by Maven to build the project is represented in the Project Object Model (POM). The POM is an XML file that resides in base directory of the project.

POM

Each Maven project is described by a file named *pom.xml* stored in the home directory of the project. This file contains the Project Object Model (POM) for the project. The POM is the basic unit of work in Maven. The POM file contains every important piece of information about the project. Understanding the POM is important. The key elements that every POM file contains are the following:

- **project** This is the top-level element in all Maven pom.xml files.
- **modelVersion** This element indicates what version of the object model this POM is using. The version of the model itself changes very infrequently but it is mandatory in order to ensure stability of use if and when the Maven developers deem it necessary to change the model.
- **groupId** This element indicates the unique identifier of the organization or group that created the project. The *groupId* is one of the key identifiers of a project and is typically based on the fully qualified domain name of your organization. For example *org.apache.maven.plugins* is the designated *groupId* for all Maven plug-ins.
- **artifactId** This is the identifier of the project. This is generally the name of the project. This element indicates the unique base name of the primary artifact being generated by this project. The primary artifact for a project is typically a JAR file. Secondary artifacts like source bundles also use the *artifactId* as part of their final name. A typical artifact produced by Maven would have the form <*artifactId*>-<*version*>.<*extension*>.
- **packaging** This element indicates the package type to be used by this artifact (e.g. JAR, WAR, EAR, etc.). This not only means if the artifact produced is JAR, WAR, or EAR but can also indicate a specific lifecycle to use as part of the build process. The default value for the packaging element is JAR so you do not have to specify this for most projects.
- **version** This element indicates the version of the artifact generated by the project. Maven goes a long way to help you with version management and you will often see the SNAPSHOT designator in a version, which indicates that a project is in a state of development.
- **name** This element indicates the display name used for the project. This is often used in Maven's generated documentation.

- **url** This element indicates where the project's site can be found. This is often used in Maven's generated documentation.
- **description** This element provides a basic description of your project. This is often used in Maven's generated documentation.

POM also contains the goals and plugins. While executing a task or goal for a project, Maven looks for the POM in the current directory. It reads the POM, gets the needed configuration information, and then executes the goal.

When you start a Maven project, the first thing you need to do is to create it. To create a Maven project you can use the Maven's archetype mechanism. An archetype is defined as *an original pattern or model from which all other things of the same kind are made*. In Maven, an archetype is a template of a project that is combined with some user input to produce a working Maven project that has been tailored to the user's requirements.

Build Lifecycle Basics

The current version of Maven is based around the central concept of a build lifecycle. This means that the process applied by Maven for building and distributing a particular artifact (project) is clearly defined. For the person building a project, this means that it is only necessary to learn a small set of commands to build any Maven project, and the POM will ensure they get the results they desired. There are three built-in build lifecycles:

- *default* – handles the project deployment;
- *clean* - handles project cleaning;
- *site* - handles the creation of the project's site documentation.

A build lifecycle is a well defined sequence of phases that define the order in which the goals are to be executed. Here phase represents a stage in life cycle.

Each build lifecycle is defined by a different list of build phases. A build phase represents a stage in the lifecycle. For example, the default lifecycle has the following build phases:

- *validate* - validate the project is correct and all necessary information is available
- *compile* - compile the source code of the project
- *test* - test the compiled source code 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.

The build phases of a build lifecycle are executed sequentially to complete the build lifecycle. Considering the build phases above, this means that when the default lifecycle is used, Maven will first validate the project, then will try to compile the sources, run those against the tests, package the binaries (e.g. jar), run integration tests against that package, verify the package, install the verified package to the local repository and then deploy the installed package in a specified environment. To do all those, you only need to call the last build phase to be executed, in this case, deploy:

$> mvn deploy

That is because if you call a build phase, it will execute not only that build phase, but also every build phase prior to the called build phase. Thus, doing

$> mvn integration-test

will do every build phase before it (validate, compile, package, etc.), before executing integration-test.

A **goal** represents a specific task that contributes to the building and managing of a project. It may be bound to zero or more build phases. A goal not bound to any build phase could be executed outside of the build lifecycle by direct invocation.
The order of execution depends on the order in which the goal(s) and the build phase(s) are invoked. Consider the following command:

$> mvn clean dependency:copy-dependencies package

In this command, the *clean* and *package* arguments are build phases while the *dependency:copy-dependencies* argument is a goal. The *clean* phase will be executed first, followed by the execution of the *dependency:copy-dependencies* goal, and finally the *package* phase will be executed.

Maven Plugins

Maven is actually a plugin execution framework where every task is actually done by plugins. A plugin generally provides a set of goals that can be executed using following syntax:

$> mvn [plugin-name]:[goal-name]

A plugin goal represents a specific task (finer than a build phase) which contributes to the building and managing of a project. It may be bound to zero or more build phases. A goal not bound to any build phase could be executed outside of the build lifecycle by direct invocation. The order of execution of a Maven command depends on the order in which the goal(s) and the build phase(s) are invoked, as we saw before. Moreover, if a goal is bound to one or more build phases, that goal will be called in all those phases.

Furthermore, a build phase can also have zero or more goals bound to it. If a build phase has no goals bound to it, that build phase will not execute. But if it has one or more goals bound to it, it will execute all those goals. A build phase is made up of plugin goals.

A build phase is responsible for a specific step in the build lifecycle, but the manner in which it carries out those responsibilities may depend on the type of project. The plugin provides a way of customization of the build phases. It is possible to specify a plugin and the define the phase or phases where the plugin should start its processing in the POM of the project.

Create a Java Project

Maven uses the **archetype** plugins to create projects. To create a simple java application, we will use maven-archetype-quickstart plugin (there are other archetype plugins for different types of Java projects). In the example below, we are creating a maven based java application project in the current directory:

```
$>mvn archetype:generate -DgroupId=pt.ist.phonebook  -DartifactId=phonebook
-DarchetypeArtifactId=maven-archetype-quickstart  -DinteractiveMode=false
```

When you execute this command in a shell, Maven will start processing and will create a complete java application with the project structure described bellow.
If you have just installed Maven, this command may take a while on the first run. This is because Maven is downloading the most recent artifacts (plugin jars and other files) into your local repository. You may also need to execute the command a couple of times before it succeeds. This is because the remote server may time out before your downloads are complete.

The first parameter of this command says that Maven is going to apply the archetype plugin. This plugin allows you to create a Maven project from a pre-defined model. Next, we define the goal to apply for this plugin: **generate**. This goal creates a project from a template. There are more goals available for this archetype.

As we saw before, the second parameter, **groupId**, just defines the organization or the project. The third parameter specifies the name of the project (phonebook in the example). The fourth parameter, **archetypeArtifactId,** defines the model that it will be used for the generation of the new Maven project. The model that we are using in this case is **maven-archetype-quickstart**. This model should be used when you want to create a simple Java project.
Finally, the last parameter, **interactiveMode**, just specifies whether the user will provide extra information during the creation of the project.

After the execution of this command, you will notice that the *generate* goal created a directory with the same name given as the *artifactId*. This directory will contain all files related to this project. Maven projects have a common directory layout. This allows users familiar with one Maven project to immediately feel at home in another Maven project. The generated Maven project created before has this standard project structure:

```
phonebook
 |-- pom.xml
 `-- src
     |-- main
     |   `-- java
```

```
|       `-- pt
|           `-- ist
|               `-- phonebook
|                   `-- App.java
`-- test
    `-- java
        `-- pt
            `-- ist
                `-- phonebook
                    `-- AppTest.java
```

The *src* directory is the root directory of the project's source code and test code. The *main* directory is the root directory for source code related to the application itself (not test code). The *test* directory contains the test source code. The *java* directories under *main* and *test* contains the Java code for the application itself (under main) and the Java code for the tests (under test).

You can also have a *resources* directory under main and test directories. This directory holds other resources needed by your project, for instance, property files used for internationalization of the application.

Is the Maven project is a web application then the project directory structure will also have the *webapp* directory. This directory will then be the root directory of the web application. Thus the *webapp* directory contains the WEB-INF directory, etc.

Finally, Maven automatically creates the *target* directory. This directory contains all the compiled classes, JAR files etc. produced by Maven. When executing the clean build phase, this directory is removed.

Compile and Run the Unit Tests

Once the application´s sources compile without errors, you should test the code using the JUnit tests developed by you. To execute these tests you should execute the following command:

$> mvn test

Executing this command line causes Maven to execute all lifecycle phases up to the *test* phase. This phase executes all unit tests of this project it can find under *src/test/java* with filenames matching ***/Test*.java, **/*Test.java and* ***/*TestCase.java* excluding those that match ***/Abstract*Test.java* or ***/Abstract*TestCase.java*.

The first time you execute this command Maven may download more dependencies. These are the dependencies and plugins necessary for executing the tests. Before compiling and executing the tests, Maven compiles the main code but it only compiles those classes that have been modified since the last time the project was compiled.

It is also possible to simply compile the test sources without executing the test. For doing this, execute the following:

$>mvn test-compile

The Maven Surefire plugin has a test goal that is bound to the *test* phase. When the Surefire plugin runs the JUnit tests, it also generates XML and text reports in the *target/surefire-reports* directory. If your tests are failing, you should look in this directory for details like stack traces and error messages generated by your unit tests.

To run a single test class (for instance, TestApp1), you can issue the following command: *mvn -Dtest=TestApp1* ***test***.

Maven always by default run your unit tests whenever Maven executes a phase that comes after the *test* phase. Sometimes, however, you want to execute the phase without running the unit tests. You can do so setting the maven.test.skip property to true. Thus, executing the following command:

$> mvn install -Dmaven.test.skip=true

builds the project but skips the execution of the unit tests.

Package a Project

Since it's unlikely that a developer will want want to distribute their project with the *.class* files directly, it is necessary to package the project first. This is accomplished by executing the following command:

$> mvn package

The *package* phase takes the compiled code and packages it in its distributable format, such as a JAR, WAR, or EAR file. The distributable format is chosen taking into account the value specified for the *packing* element present in the *pom.xml* file of the Maven project. The name of the package file will be based on the project's <artifactId> and <version>. This file is placed in the target directory present in the project's home directory. If a developer wants to install this generated file in the local repository, he just has to execute the following command:

$> mvn install

Run a Java Class

You can run a Java class using the Maven *exec* plugin using the following command:

$>mvn exec:java -Dexec.mainClass="com.example.Main" [-Dexec.args="arguments"]

The last parameter of this command, **exec.args**, is optional and it is needed only if you have to provide arguments to the execution of the Java class.
The *exec* plugin has supports two goals:

- exec:exec executes programs and Java programs in a separate process.
- exec:java executes Java programs in the same VM.

Clean the Project

To remove all files automatically generated during the build project you should execute the following:

$> mvn clean

This will remove the *target* directory, which contains all the build data, from the project´s home directory.

Declare Dependencies

The simple Hello World sample that is created when the is usedis completely self-contained and does not depend on any additional libraries. However, most applications depend on external libraries to handle common and complex functionality. These dependencies must be defined in the POM.

For example, suppose that in addition to saying "Hello World!", you want the sample application to print the current date and time using the Joda Time libraries.

First, you should change HelloWorld.java to look like this:

```
package hello;
import org.joda.time.LocalTime;
public class HelloWorld {
    public static void main(String[] args) {
      LocalTime currentTime = new LocalTime();
      System.out.println("The current local time is: " + currentTime);
      Greeter greeter = new Greeter();
      System.out.println(greeter.sayHello());
   }
 }
```

In this example we are using Joda Time's LocalTime class to get and print the current time.
Now, if you were to run mvn compile to build the project, the build would fail because you have not declared Joda Time as a compile dependency in the build for this project. You can fix that by adding the following lines to *pom.xml* (within the <project> element):

```
  <dependencies>
     <dependency>
       <groupId>joda-time</groupId>
       <artifactId>joda-time</artifactId>
       <version>2.2</version>
     </dependency>
   </dependencies>
```

This block of XML declares a list of dependencies for the project. Specifically, it declares a single dependency for the Joda Time library. Within the <dependency> element, the dependency coordinates are defined by three sub-elements:

- <groupId> - The group or organization that the dependency belongs to.
- <artifactId> - The library that is required.

- <version> - The specific version of the library that is required.

By default, all dependencies are scoped as compile dependencies. That is, they should be available at compile-time (and if you were building a WAR file, including in the */WEB-INF/libs*folder of the WAR). Additionally, you may specify a <scope> element to specify one of the following scopes:

- provided - Dependencies that are required for compiling the project code, but that will be provided at runtime by a container running the code (e.g., the Java Servlet API).
- test - Dependencies that are used for compiling and running tests, but not required for building or running the project's runtime code.

Now if you run *mvn compile* or *mvn package*, Maven should resolve the Joda Time dependency from the Maven Central repository and the build will be successful.

Maven Repositories

In Maven terminology, a repository is a place i.e. directory where all the project jars, library jar, plugins or any other project specific artifacts are stored and can be used by Maven easily. A repository in Maven is used to hold build artifacts and dependencies of varying types.
There are strictly only two types of repositories: local and remote. A Maven local repository is a folder location on your machine. The local repository refers to a copy on your own installation that is a cache of the remote downloads (library jars, plugin jars, etc), and also contains the temporary build artifacts that you have not yet released. The local repository keeps all dependencies of your Maven projects.

The local repository is automatically created when you run any Maven command for the first time. By default, this directory is named *repository* and it is placed in directory *~/.m2*.

When you build a Maven's project, Maven will check the project´s POM to identify the dependencies of the project. All dependencies that are missing in the local repository are automatically downloaded from a remote repository. First, Maven will get the dependency from the local repository. If the dependency is not found in the local repository, Maven will try to get it from the *Maven central repository* – http://repo.maven.org/maven2/. This central repository is a repository provided by Maven community. It contains a large number of commonly used libraries.

To browse the content of central maven repository, maven community has provided the following URL:http://search.maven.org/#browse. Using this URL, a developer can search all the available libraries in central repository.

Sometime, Maven does not find a mentioned dependency in the central repository as well. When this happens, the build process stops and it prints an output error message in the console. To prevent such situation, Maven allows users to specify other remote repositories that will contain required libraries or other project jars. These extra remote repositories are specified in the project's POM. A dependency is searched in this remote repository if the dependency is not found neither in the local repository nor in the central repository.

Eclipse IDE

To make a Maven project as an Eclipse project, execute the following command in the project's home directory:
$> mvn eclipse:eclipse

The execution of this command will generate all project files that are required by Eclipse IDE. To import the project into the Eclipse IDE, select "File -> Import… -> General->Existing Projects into Workspace".