Sponsored by:

This story appeared on JavaWorld at
http://www.javaworld.com/javaworld/jw-05-2006/jw-0529-maven.html

The Maven 2 POM demystified

The evolution of a project model

By Eric Redmond, JavaWorld.com, 05/29/06

Building a project is complex business. Due to the dozens of tasks required in converting your hodge-podge of files into a working program, there exist literally hundreds of tools that do everything from generating source code, to compiling, to testing, to distribution, to brewing your morning coffee (if you find one, dear reader, let me know). Many of these programs are excellent at what they do. Unfortunately, for those of us who manage large-scale build systems for a living, there is rarely much commonality; each program requires its own disparate installation and esoteric configuration. It has become an inevitable fact of our lives that the majority of build systems are custom built by hand-gluing these tools with several homebrew scripts (yeah, Ant scripts count).

More than another build tool, Maven is a build *framework*. It cleanly separates your code from configuration files, documentation, and dependencies. Maven is surprisingly flexible in letting users configure most aspects of their code, as well as in controlling the behavior of plug-ins, individual goals, and even the build lifecycle itself. Maven is the actual structure, and within these walls, your project dwells; it wants to be an accommodating host.

But the problem still remains: managing the work of thousands of custom build scripts within a single framework is tough and, to be done correctly, requires much information. Fortunately, the Maven 2 team has been quite successful. Learning from the mistakes of Maven 1, countless user requests, tweaking, and updating, Maven 2 is more powerful than ever. Unfortunately, with great power comes great configuration. In order for Maven 2 artifacts to be easily portable units, that complex configuration falls into a single file. Enter the Maven POM.

What is the POM?

POM stands for project object model. It is an XML representation of a Maven project held in a file named pom.xml. In the presence of Maven folks, speaking of a project is speaking in the philosophical sense, beyond a mere collection of files containing code. A project contains configuration files, as well as developers involved and roles they play, the defect tracking system, the organization and licenses, the URL where the project lives, the project's dependencies, and all the other little pieces that come into play to give code life. A project is a one-stop shop for all things related to it. In fact, in the Maven world, a project need not contain any code at all, merely a pom.xml. We will encounter a couple such types of projects later in the article.

A quick structural overview

The POM is large and complex, so breaking it into pieces eases digestion. For the purposes of this discussion, these pieces are regrouped into four logical units, as shown in Figure 1: POM relationships, project information, build settings, and build environment. We shall begin by discussing POM relationships.

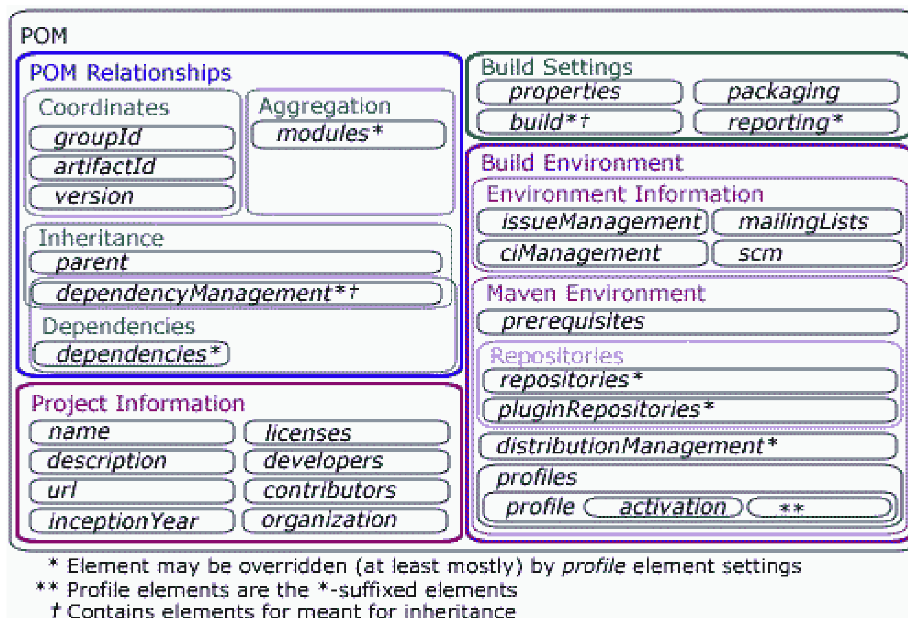

Figure 1. POM overview

Below is a listing of the elements directly under the POM's project element. Notice that `modelVersion` contains *4.0.0*. That is currently the only supported POM version for Maven 2 and is always required. The Maven 4.0.0 XML schema definition is located at http://maven.apache.org/maven-v4_0_0.xsd. Its top-level elements are as follows:

```
<project>
  <modelVersion>4.0.0</modelVersion>


  <!-- POM Relationships -->
  <groupId>...</groupId>
  <artifactId>...</artifactId>
  <version>...</version>
  <parent>...</parent>
  <dependencyManagement>...</dependencyManagement>
  <dependencies>...</dependencies>
  <modules>...</modules>


  <!-- Project Information -->
  <name>...</name>
  <description>...</description>
  <url>...</url>
  <inceptionYear>...</inceptionYear>
  <licenses>...</licenses>
  <developers>...</developers>
  <contributors>...</contributors>
  <organization>...</organization>


  <!-- Build Settings -->
  <packaging>...</packaging>
  <properties>...</properties>
  <build>...</build>
  <reporting>...</reporting>


  <!-- Build Environment -->
  <!-- Environment Information -->
  <issueManagement>...</issueManagement>
  <ciManagement>...</ciManagement>
  <mailingLists>...</mailingLists>
  <scm>...</scm>
  <!-- Maven Environment -->
  <prerequisites>...</prerequisites>
  <repositories>...</repositories>
  <pluginRepositories>...</pluginRepositories>
  <distributionManagement>...</distributionManagement>
  <profiles>...</profiles>
</project>
```

POM relationships

Our first order of business is to investigate project relationships, represented in Figure 2 as the top-left corner of the chart in Figure 1.

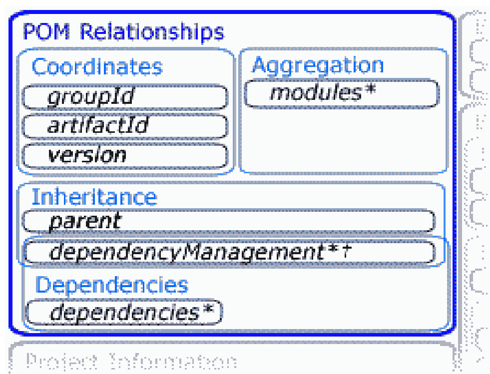

Figure 2. POM relationships

Projects must relate to each other in some way. Since the creation of the first assemblers, software projects have had dependencies; Maven has introduced more forms of relationships hitherto unused in such a form for Java projects. These relationships are Maven coordinates, coordinate-based dependencies, project inheritance, and aggregation.

Coordinates

Each Maven project contains its own unique identifier, dubbed the project's *coordinates*, which acts like an artifact's address, giving it a unique place in the Maven universe. If projects had no way of relating to each other, coordinates would not be needed. That is, if a universe had just one house, why would it need an address like 315 Cherrywood Lane?

The code below is the minimum POM that Maven 2 will allow—`<groupId>`, `<artifactId>`, and `<version>` are all required fields. They act as a vector in Maven space with the elements grouper, identifier, and timestamp.

```
 <project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>a</artifactId>
  <version>1</version>
</project>
```

In the Maven world, these three main elements (The Maven trinity—behold its glory!) make up a POM's coordinates. The coordinates are represented by Figure 3.

Figure 3. A discrete Maven unit

Perhaps this POM is not so impressive by itself. It gets better.

Dependencies

One of the most powerful aspects of Maven is its handling of project dependencies, and in Maven 2, that includes

transitive dependencies. Figure 4 illustrates how we shall represent them graphically.

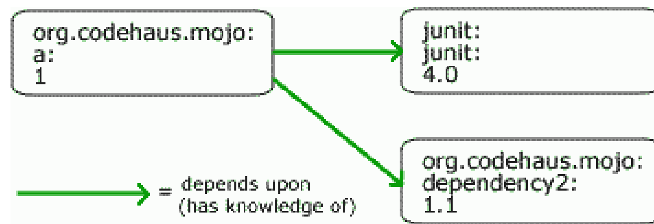

Figure 4. Project dependencies

Dependency management has a long tradition of being a complicated mess for anything but the most trivial of projects. "Jarmageddon" quickly ensues as the dependency tree becomes huge, complicated, and embarrassing to architects who are scorned by new graduates who "totally could have done it better." "Jar Hell" follows, where versions of dependencies on one system are not quite the same versions as those used for development; they have either the wrong version or conflicting versions between similarly named JARs. Hence, things begin breaking and pinpointing why proves difficult. Maven solves both of these problems by having a common local repository from which to link to the correct projects, versions and all.

Inheritance

One of the features that Maven 2 brings from the Maven 1 days is project inheritance, as represented in Figure 5. In build systems, such as Ant, inheritance can certainly be simulated, but Maven has taken the extra step in making project inheritance explicit to the project object model.

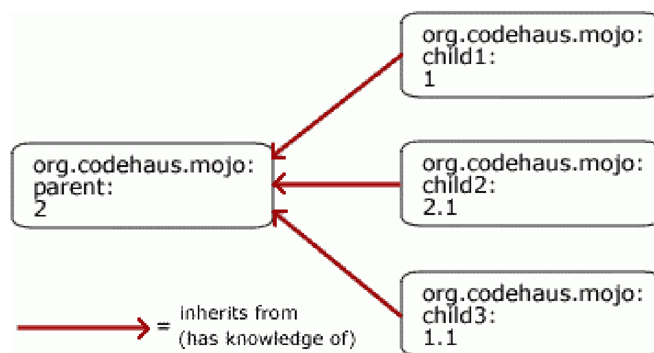

Figure 5. Project inheritance

The following code defines a parent POM in Maven 2:

```
 <project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>b</artifactId>
  <version>2</version>
  <packaging>pom</packaging>
</project>
```

This parent looks similar to our first POM, with a minor difference. Notice that we have set the `packaging` type as *pom*, which is required for both parent and aggregator projects (we will cover more on `packaging` in the "Build Settings" section). If we want to use the above project as a parent, we can alter the project `org.codehaus.mojo:a` POM to be:

```
 <project>
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>b</artifactId>
    <version>2</version>
  </parent>
```

```
  <!-- Notice no groupId or version. They were inherited from parent-->
  <artifactId>a</artifactId>
</project>
```

It is important to note that all POMs inherit from a parent whether explicitly defined or not. This base POM is known as the "super POM," and contains values inherited by default. An easy way to look at the default configurations of the super POM is by creating a simple pom.xml with nothing but `modelVersion`, `groupId`, `artifactId`, and `version`, and running the command `mvn help:effective-pom`.

Beyond simply setting values to inherit, parents also have the power to create default configurations for their children without actually imposing values upon them. Dependency management is an especially powerful instrument for configuring a set of dependencies through a common location (a POM's parent). The `dependencyManagement` element syntax is similar to that of the dependency section. What it does, however, is allow children to inherit dependency settings, but not the dependency itself. Adding a dependency with the `dependencyManagement` element does not actually add the dependency to the POM, nor does it add a dependency to the children; it creates a default configuration for any dependency that a child may choose to add within its own dependency section. Settings by `dependencyManagement` also apply to the current POM's dependency configuration (although configurations overridden inside the dependency element always take precedence).

Aggregation

A project with modules is known as a multimodule project. Modules are projects that a POM lists, executed as a set. Multimodule projects know of their modules, but the reverse is not necessarily true, as represented in Figure 6.

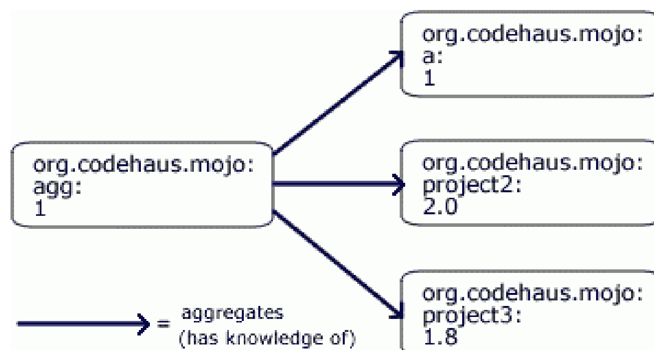

Figure 6. A multimodule project

Assuming that the parent POM resides in the parent directory of where POM for project *a* lives, and that the project *a* also resides in a directory of the same name, we may alter the parent POM *b* to aggregate the child *a* by adding it as a module:

```
 <project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>b</artifactId>
  <version>2</version>
  <packaging>pom</packaging>
  <modules>
    <module>a</module>
  </modules>
</project>
```

Now if we ran `mvn compile` in the base directory, you would see the build start with:

```
 [INFO] Scanning for projects...
```

```
[INFO] Reactor build order:
[INFO]   Unnamed - org.codehaus.mojo:b:pom:2
[INFO]   Unnamed - org.codehaus.mojo:a:jar:2
```

The Maven lifecycle will now execute up to the lifecycle phase specified in correct order; that is, each artifact is built one at a time, and if one artifact requires another to be built first, it will be.

A note on inheritance vs. aggregation
Inheritance and aggregation create a nice dynamic for controlling builds through a single, high-level POM. You will often see projects that are both parents and multimodules, such as the example above. Their complementariness makes them a natural match. Even the Maven 2 project core runs through a single parent/multimodule POM `org.apache.maven:maven`, so building a Maven 2 project can be executed by a single command: `mvn compile`. Although used in conjunction, however, a multimodule and a parent are not one in the same, and should not be confused. A POM project (acting as a parent) may be inherited from, but that parent project does not necessarily aggregate any modules. Conversely, a POM project may aggregate projects that do not inherit from it.

When all four pieces of the equation are put together, hopefully you will see the power of the Maven 2 relationship mechanism, as shown in Figure 7.

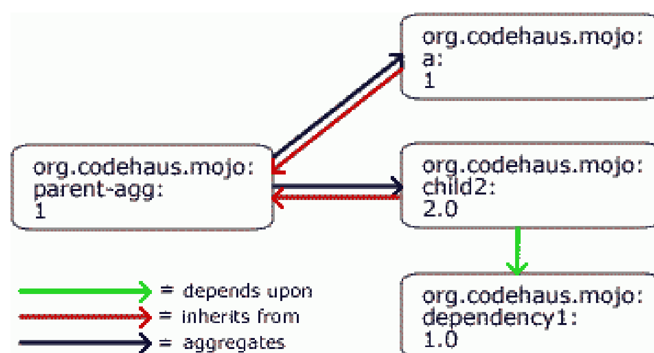

Figure 7. A relationship graph

Maven gives us a nice framework for relating projects to each other, and through these relationships, we may create plug-ins reusable by any project following Maven's conventions. But the ability to manage project relationships is only a part of the overall Maven equation. The rest of the POM is concerned not with other projects, but with its build settings, its information, and with its environment. With a quick understanding of how projects relate to one another out of the way, let's begin to look at how a POM contains information about the project proper.

Project information

POM information is only as useful as the plug-in that uses it. With this in mind, we will not drill down into the elements of this section in much detail. These elements are mostly self-explanatory and frankly uninteresting beyond their use for reporting plug-ins. The idea that they are used for reporting, however, does not automatically qualify them as build settings. The project information elements are merely used as *part of a* build process and not actively involved in *configuring*
it. I know—it's a fine line. This batch of information, although not explicitly required to perform most builds, is designed to communicate to other humans some general information about the project. As illustrated in Figure 8, here is where the POM author can define the project's human-readable name, description, homepage (URL), inception year, licenses, the organization in charge of development, as well as the developers and contributors (with a little data about them too).

Figure 8. General project information

A big piece of the POM's bulk is this human-readable information. If you learn to use it, your documentation readers will thank you. I would never claim that this information is not important, just not worth looking into in an overview.

Build settings

Build settings is where the POM gets interesting, where we get to the real meat of it.

Figure 9. Project build settings

Half of Maven's power lies within the two elements `build` and `reporting`. "But there are four elements under build settings" (as shown in Figure 9), I hear you say. Quite so, and we shall quickly cover those two first: the `packaging` and `properties` elements.

Packaging

The `packaging`
element describes to Maven what default goals to bind under the lifecycle and offers a hint of the project's type. If not specified, then `packaging` will default to `jar`. The other valid types are: `pom`, `maven-plugin`, `ejb`, `war`, `ear`, `rar`, `par`, and `ejb3`. These values are each associated with a default list of goals to execute for each corresponding build lifecycle stage for a particular packaging structure. For example, the `jar` packaging type executes the `jar:jar` goal during the build lifecycle's package phase, where the `ejb` packaging type executes `ejb:ejb`. For those who simply must know everything, these types are defined under the Maven core project's Plexus component settings. Each is a role-hint for the `org.apache.maven.lifecycle.mapping.LifecycleMapping` role.

Properties

The `properties`
element is used throughout a POM and Maven plug-ins as a replacement for values. When a property is set, you may use the property name as a value in the form of `${name}`, which will be replaced by the set value at runtime. Consider the following:

```
 <properties>
  <env.name>tiger</env.name>
</properties>
```

Now, wherever the property is used within the POM as `${env.name}`, Maven will replace that string with the value *`tiger`*.

There are many other ways to set properties—through the command line, through the Maven settings.xml files, through filters. Most of these approaches are preferred over using the `properties` element. The simple reason being: if you insert a value into the POM, why use a property over the value itself? I find it best to avoid this approach except for testing purposes, for setting defaults, or for advanced use-cases (plug-ins injecting properties, or very rarely, long and oft-used static variables).

Build

In the simplest terms, the `build`
element contains information describing how a project's build is to proceed when executed. It contains all sorts of

useful information, such as where the source code lives or how plug-ins are to be configured. Much of this information is inherited from the super POM. Like most everything in the POM, you may override these defaults (though not generally recommended). Like elsewhere in the POM that expects paths, all directories not beginning with "/" or some other absolute path delimiter are relative to *${baseDir}*—the directory where the POM lives. The build conventions defined by the super POM are show below:

```
 <project>
  ...
  <build>
    <sourceDirectory>src/main/java</sourceDirectory>
    <scriptSourceDirectory>src/main/scripts</scriptSourceDirectory>
    <testSourceDirectory>src/test/java</testSourceDirectory>
    <directory>target</directory>
    <outputDirectory>target/classes</outputDirectory>
    <testOutputDirectory>target/test-classes</testOutputDirectory>
    <finalName>${artifactId}-${version}</finalName>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
      </resource>
    </resources>
    <testResources>
      <testResource>
        <directory>src/test/resources</directory>
      </testResource>
    </testResources>
  </build>


  ...
</project>
```

Besides setting a project's source and target directories, the build section configures plug-in executions, adds extensions to them, mucks with the build lifecycle, and has a role to play in POM inheritance via the `dependencyManagement` element.

Although Maven 2 is good at defining defaults, plug-ins should attempt to guess only so much information. At a certain point, they must be configured explicitly, which is mostly done through the `configuration` element. One of the more common plug-ins configured is the compiler plug-in. The `maven-compiler-plugin` defaults to compile Java code as J2SE 1.3-compliant, and must be configured for any other setting. For a project using Java SE 5, the plug-in would be configured as follows:

```
 <project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>         <source>1.5</source>
          <target>1.5</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

Note: You are not actually required to set `groupId` for Maven plug-ins with the `groupId` of `org.apache.maven.plugins`. These are considered to be special core plug-ins by Maven (just like `org.apache.maven.plugins` need not be specified when calling goals from the command line).

Although plug-in configuration is useful in and of itself, it does not afford you the granularity of control that is always required. Most plug-ins have multiple goals that you may execute—each with differing configurations.

Moreover, you may wish to bind a goal to a particular phase (you cannot bind a whole plug-in to a phase). Enter the `executions` element: to give goal control to the POM designer.

For example, let's make our build echo the `java.home` property during the validate phase of the build lifecycle, just to be sure it is has been set correctly. Although we could write our own plug-in in Java and bind it to the validate phase, it would be far easier to use the `antrun` plug-in's `run` goal, bind it to the validate phase, and configure the plug-in to echo the property:

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-antrun-plugin</artifactId>
        <executions>
          <execution>
            <id>echohome</id>
            <phase>validate</phase>
            <goals>
               <goal>run</goal>
            </goals>
            <configuration>             <tasks>
                <echo>JAVA_HOME=${java.home}</echo>
              </tasks>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

In the above code, we specified the `antrun:run` goal to be bound to `validate`, giving it a configuration to echo a `task`. In the POM, all XML within a `configuration` element is free text; it is up to the plug-in to decide what to do with the configuration data. This is a mixed blessing, however. It is a good thing in that it gives power and versatility to Maven 2 plug-in writers. The downside is that plug-in users are responsible for understanding the configuration quirks of each plug-in that they use. Some plug-ins and goals are simple, having one or two `configuration` elements; some are more complex like `antrun:run`, which has the entire Ant library at its disposal.

It is a powerful matter to manipulate plug-ins as you see fit. But you undoubtedly will run across cases where the `configuration`
element must be duplicated for every POM in a set of projects. The Maven 2 developers foresaw this issue and thus created the `pluginManagement` element in the build section. This element contains its own `plugins` element—the difference being that plug-in settings defined under this section will be passed onto this POM's children in much the same way that `dependencyManagement`
manages its children's dependencies configurations. Maven 2 will then pass the above configuration mess to any of this POM's children, who may then flag their desire to use the `[antrun:run {execution: echohome}]` settings by asking to use the plug-in, like so:

```
<project>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-antrun-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

The last major elements under the `build` element are `resource` and `filter`. Resources are not (usually) code. They are not meant to be compiled, but are items used for `build` and `reporting` settings, or bundled within your project—for example, configuration files or class file stubs used in code generation.

Filters, on the other hand, are a way in which the POM can externalize properties to another file (rather than setting them via the `properties` element mentioned above). Each `*.properties filter` follows standard `java.util.Properties` file layout: a new-line separated list of `name=value` pairs. These properties may then be applied to resource files (or any other plug-in that uses properties, for that matter).

As an example of how these elements may be used together, let's look at a Plexus configuration file. Plexus is an Inversion of Control container along the same vein as PicoContainer, Nano, and Spring—upon which Maven is based. Like most IoC solutions, Plexus requires a configuration file to specify component configurations to the container. A valid Plexus JAR requires that the configuration.xml file live under the META-INF/plexus directory. We could just as easily place this file within src/main/resource/META-INF/plexus; however, we instead give Plexus resources their own directory at src/main/plexus. Although it is the job of the configuration.xml file to contain a component's concrete implementation data, we instead replace the `implementation` string with a key whose real value is defined within a `components.properties filter` file.

The configuration.xml stub with the implementation key looks as follows:

```
 <component-set>
  <components>
    <component>
      <role>org.codehaus.plexus.component.configurator.ComponentConfigurator</role>
      <implementation>${compConfig.impl}</implementation>
    </component>
  </components>
</component-set>
```

We create a `filter` file named `components.properties` in the same directory that maps the `${compConfig.impl}` property to a value:

```
 compConfig.impl=org.codehaus.mojo.ruby.plexus.component
   .configurator.RubyComponentConfigurator
```

This file's value will be included in the POM as a filter and will replace the configuration.xml file's property of the same name.

Finally, the POM will contain the following:

```
 <project>
  ...
  <build>
    <filters>
      <filter>src/main/plexus/components.properties</filter>
    </filters>
    <resources>
      <resource>
        <targetPath>META-INF/plexus</targetPath>
        <filtering>true</filtering>
        <directory>src/main/plexus</directory>
        <includes>
          <include>configuration.xml</include>
        </includes>
      </resource>
    </resources>
  </build>
  ...
</project>
```

Besides including files explicitly, we may instead have excluded `*.properties` files to similar effect. After running the process-resources phase of the build lifecycle, the resource will move to the build output directory with the property filtered out:

```
<implementation>org.codehaus.mojo.ruby.plexus.component.configurator.
  RubyComponentConfigurator</implementation>
```

And, yea, the world is as it should be. For a look at what more filters and resources can do, take a look at the Maven 2 project's own quick start guide.

Reporting

Maven defines more than the default build lifecycle. One of the more impressive ideas comes from the site generation lifecycle. Certain Maven plug-ins can generate reports defined and configured under the `reporting` element—for example, generating Javadoc reports. Much like the `build` element's ability to configure plug-ins, `reporting`
commands the same ability. The glaring difference is that rather than fine-grained control of plug-in goals within the `executions` block, `reporting` configures goals within `reportSet` elements. And the subtler difference is that plug-in configurations under the `reporting` element work as `build` plug-in configurations, although the opposite is not true (`build` plug-in configurations do not affect `reporting`).

Possibly the only item under the `reporting` element that would not be familiar to someone who understood the `build` element is the Boolean `excludeDefaults`
element. This element signifies to the site generator to exclude reports normally generated by default. When a site is generated via the site build cycle, a "Project Info" section is placed in the left-hand menu, chock full of reports, such as the *Project Team* report or *Dependencies* list report. These report goals are generated by `maven-project-info-reports-plugin`. Being a plug-in like any other, it may also be suppressed in the following, more verbose, way, which effectively turns off project-info reports:

```
 <project>
  ...
  <reporting>
    <plugins>
      <plugin>
        <artifactId>maven-project-info-reports-plugin</artifactId>
        <reportSets>
          <reportSet></reportSet>
        </reportSets>
      </plugin>
    </plugins>
  </reporting>
  ...
</project>
```

The `reporting` element is similar to `build`. So similar, in fact, that plug-in configuration can be conceptualized as effectively as a subset of `build`, focused entirely on the site phase. Although I find its existence increases the POM, and hence generates more work to learn for little gain, the Maven 2 team obviously disagreed. It is too early to set up camp on either side. For the time being, however, we must learn and understand both.

Build environment

We now examine how the POM defines and interacts with its environment to finally encompass most aspects of a standard build system. We finish with the lower right block of our POM chart introduced in Figure 1.

Figure 10. Project environment settings

Environment information

Most of the elements here are descriptive of the type of lifestyle in which the project makes itself comfortable. Elements by such names as `ciManagement` (Continuum, CruiseControl, etc.), `issueManagement` (Bugzilla, etc.), `scm` (CVS, Subversion, etc.), and `mailingLists`
(emails and archives) all outline the programs and the settings that this build system has. As great as Maven 2 may be, it does not do everything needed in a project's lifecycle. These elements are concerned with the programs that live in conjunction with Maven to create a healthy environment where our fat little project can grow up big and strong. Think of these programs as Maven's family.

Among these elements, `scm` is one that deserves a second look. The `scm` element is connected to a separate subproject named (surprise!) SCM, which seeks to make a common Java interface through which all software configuration management tools may communicate. The SCM project is of particular interest to the `scm` element because of its use of the SCM URL format. When an `scm`
element is defined in Maven 2, it uses the URL to decode implementation, location, connection, and other information (much like a Java Database Connectivity URL). Its format is: `scm:[provider]:[provider_specific]`, where *provider* is the type of SCM system. For example, connecting to a CVS repository may look like: `scm:cvs:pserver:127.0.0.1:/cvs/root:a`. More detailed information on the SCM project may be found from the Maven SCM page.

Maven environment

The remainder of the environment elements are Maven-focused, such as `prerequisites` for building the POM or the repository elements: `repositories` and `pluginRepositories`. These elements define from where a project may download dependency projects, how often it should try, what types of projects it wants (releases or snapshots), as well as all of the same information for downloading plug-ins. On the flip side of the repository is `distributionManagement`, which specifies how a project may join the ranks of a remote repository. The distribution may define a release and snapshot repository, as well as a location for deploying this project's generated site and, finally, a good place for a deployed project status.

All of the build environment elements we have glanced at are straightforward and simple. In comparison, the `profile` element has a powerful yet potentially dangerous role to play within the POM. `profiles` are unique in that they do not define the setup of a project's build environment, but instead tell Maven how to react under different environments. A `profile`
element does this by allowing certain POM elements to be overwritten according to environmental factors. Those `profiles` may be explicitly activated via the command line, or a settings.xml file, or even activated automatically, such as by the (non)existence of a file or the value of a given property. For example, if we know that our test environments always contain a file called test1.properties under the Java installation directory, then we can activate a test profile as follows:

```
<project>

  ...
```

```
  <profiles>
    <profile>
      <id>test</id>
      <activation>
        <file>
          <exists>${java.home}/test1.properties</exists>
        </file>
      </activation>
      ...
    </profile>
  </profiles>
  ...
</project>
```

All of the elements that may be set under `profiles`
will now override their POM counterparts whenever this project is built under a test environment. The elements that may be overridden by an active profile are a subset of the `build` and `reporting` elements, relationship elements (`dependencies`, `dependencyManagement`, and `modules`), and certain build-and-deployment-related environment elements (`repositories`, `pluginRepositories`, and `distributionManagement`). You can find the exhaustive list on the Maven Model page. Note in particular the missing `*Directory` elements under `build`.

Sometimes you may want a POM to flex, but do not want to define `profiles` within the POM. These situations call for the settings.xml and profiles.xml files. Although beyond the scope of this article, settings.xml is useful for defining properties that should encompass a build system as a whole (or at least an individual user's requirements). The profiles.xml file, on the other hand, proves useful when you wish to add profiles to an individual build, but do not wish to/cannot make modifications to the pom.xml, or where such changes may not be appropriate.

An astute reader may notice that `profiles` may introduce a point of failure for a build. This is entirely possible. For example, if one of our test servers did not contain the test1.properties file, then the profile would never activate, possibly causing undefined builds. If you begin to notice erratic behaviors in builds involving profiles, you can easily find out which profiles are active by executing the `help:active-profiles` goal, which will list the active profiles by ID. Believe me, this can save you hours of headaches and spare your walls from head-shaped dents.

Conclusion

As you can see, the Maven 2 POM is big. That is of no doubt. However, its size is also a testament to its versatility. The ability to abstract all aspects of a project into a single artifact is powerful, to say the least. Gone are the days of dozens of disparate build scripts and scattered documentation concerning each individual project. Along with the other stars that make up the Maven 2 galaxy—a well-defined build lifecycle, easy-to-write-and-maintain plug-ins, centralized repositories, system-wide and user-based configurations, and the increasing number of tools to make developers' jobs easier to maintain complex projects—the POM is the large, but bright, center.

Author Bio

Eric Redmond has been an active enthusiast of Maven 2 since its beta?and of Maven 1 before that. Currently, in his position as a senior engineer of a five-man IT department, he is using Maven 2 to coordinate new development with a loosely coupled team of consultants. Redmond holds a bachelor's of science in computer science from Purdue University and is the author of Maven 2 Ruby support.

All contents copyright 1995-2008 Java World, Inc. http://www.javaworld.com