POM Reference

Introduction

- The POM 4.0.0 XSD and descriptor reference documentation

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`. When 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 the developers involved and the roles they play, the defect tracking system, the organization and licenses, the URL of where the project lives, the project's dependencies, and all of the other little pieces that come into play to give code life. It is a one-stop-shop for all things concerning the project. In fact, in the Maven world, a project does not need to contain any code at all, merely a `pom.xml`.

Quick Overview

This 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, and is always required.

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>

 <!-- The Basics -->
 <groupId>...</groupId>
 <artifactId>...</artifactId>
 <version>...</version>
 <packaging>...</packaging>
 <dependencies>...</dependencies>
 <parent>...</parent>
 <dependencyManagement>...</dependencyManagement>
 <modules>...</modules>
 <properties>...</properties>

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

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

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

The Basics

The POM contains all necessary information about a project, as well as configurations of plugins to be used during the build process. It is the declarative manifestation of the "who", "what", and "where", while the build lifecycle is the "when" and "how". That is not to say that the POM cannot affect the flow of the lifecycle - it can. For example, by configuring the `maven-antrun-plugin`, one can embed Apache Ant tasks inside of the POM. It is ultimately a declaration, however. Whereas a `build.xml` tells Ant precisely what to do when it is run (procedural), a POM states its configuration (declarative). If some external force causes the lifecycle to skip the Ant plugin execution, it does not stop the plugins that are executed from doing their magic. This is unlike a `build.xml` file, where tasks are almost always dependant on the lines executed before it.

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>

 <groupId>org.codehaus.mojo</groupId>
 <artifactId>my-project</artifactId>
 <version>1.0</version>
</project>
```

Maven Coordinates

The POM defined above is the bare minimum that Maven allows. `groupId:artifactId:version` are all required fields (although, groupId and version do not need to be explicitly defined if they are inherited from a parent - more on inheritance later). The three fields act much like an address and timestamp in one. This marks a specific place in a repository, acting like a coordinate system for Maven projects:

- **groupId**: This is generally unique amongst an organization or a project. For example, all core Maven artifacts do (well, should) live under the groupId `org.apache.maven`. Group ID's do not necessarily use the dot notation, for example, the junit project. Note that the dot-notated groupId does not have to correspond to the package structure that the project contains. It is, however, a good practice to follow. When stored within a repository, the group acts much like the Java packaging structure does in an operating system. The dots are replaced by OS specific directory separators (such as '/' in Unix) which becomes

a relative directory structure from the base repository. In the example given, the `org.codehaus.mojo` group lives within the directory `$M2_REPO/org/codehaus/mojo`.

- **artifactId**: The artifactId is generally the name that the project is known by. Although the groupId is important, people within the group will rarely mention the groupId in discussion (they are often all be the same ID, such as the MojoHaus project groupId: `org.codehaus.mojo`). It, along with the groupId, creates a key that separates this project from every other project in the world (at least, it should :)). Along with the groupId, the artifactId fully defines the artifact's living quarters within the repository. In the case of the above project, `my-project` lives in `$M2_REPO/org/codehaus/mojo/my-project`.
- **version**: This is the last piece of the naming puzzle. `groupId:artifactId` denotes a single project but they cannot delineate which incarnation of that project we are talking about. Do we want the `junit:junit` of 2018 (version 4.12), or of 2007 (version 3.8.2)? In short: code changes, those changes should be versioned, and this element keeps those versions in line. It is also used within an artifact's repository to separate versions from each other. `my-project` version 1.0 files live in the directory structure `$M2_REPO/org/codehaus/mojo/my-project/1.0`.

The three elements given above point to a specific version of a project, letting Maven know *who* we are dealing with, and *when* in its software lifecycle we want them.

Packaging

Now that we have our address structure of `groupId:artifactId:version`, there is one more standard label to give us a really complete *what*: that is the project's packaging. In our case, the example POM for `org.codehaus.mojo:my-project:1.0` defined above will be packaged as a `jar`. We could make it into a `war` by declaring a different packaging:

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <packaging>war</packaging>
 ...
</project>
```

When no packaging is declared, Maven assumes the packaging is the default: `jar`. The valid types are Plexus role-hints (read more on Plexus for a explanation of roles and role-hints) of the component role `org.apache.maven.lifecycle.mapping.LifecycleMapping`. The current core packaging values are: `pom`, `jar`, `maven-plugin`, `ejb`, `war`, `ear`, `rar`. These define the default list of goals which execute on each corresponding build lifecycle stage for a particular package structure: see Plugin Bindings for default Lifecycle Reference for details.

POM Relationships

One powerful aspect of Maven is its handling of project relationships: this includes dependencies (and transitive dependencies), inheritance, and aggregation (multi-module projects).

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 large and complicated. *"Jar Hell"* follows, where versions of dependencies on one system are not equivalent to the versions developed with, either by the wrong version given, or conflicting versions between similarly named jars.

Maven solves both problems through a common local repository from which to link projects correctly, versions and all.

Dependencies

The cornerstone of the POM is its dependency list. Most projects depend on others to build and run correctly. If all Maven does for you is manage this list, you have gained a lot. Maven downloads and links the dependencies on compilation, as well as on other goals that require them. As an added bonus, Maven brings in the dependencies of those dependencies (transitive dependencies), allowing your list to focus solely on the dependencies your project requires.

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <dependencies>
   <dependency>
     <groupId>junit</groupId>
     <artifactId>junit</artifactId>
     <version>4.12</version>
     <type>jar</type>
     <scope>test</scope>
     <optional>true</optional>
   </dependency>
   ...
 </dependencies>
 ...
</project>
```

groupId, **artifactId**, **version**:
You will see these elements often. This trinity is used to compute the Maven coordinate of a specific project in time, demarcating it as a dependency of this project. The purpose of this computation is to select a version that matches all the dependency declarations (due to transitive dependencies, there can be multiple dependency declarations for the same artifact). The values should be:

- **groupId**, **artifactId**: directly the corresponding coordinates of the dependency,
- **version**: a **dependency version requirement specification**, that is used to compute the dependency's effective version.

Since the dependency is described by Maven coordinates, you may be thinking: "This means that my project can only depend upon Maven artifacts!" The answer is, "Of course, but that's a good thing." This forces you to depend solely on dependencies that Maven can manage.

There are times, unfortunately, when a project cannot be downloaded from the central Maven repository. For example, a project may depend upon a jar that has a closed-source license which prevents it from being in a central repository. There are three methods for dealing with this scenario.

Install the dependency locally using the install plugin. The method is the simplest recommended method. For example:

```
mvn install:install-file -Dfile=non-maven-proj.jar -DgroupId=some.group -DartifactId=non-maven-proj -Dversion=1 -Dpackagi
```

Notice that an address is still required, only this time you use the command line and the install plugin will create a POM for you with the given address.

2. Create your own repository and deploy it there. This is a favorite method for companies with an intranet and need to be able to keep everyone in synch. There is a Maven goal called `deploy:deploy-file` which is similar to the `install:install-file` goal (read the plugin's goal page for more information).
3. Set the dependency scope to `system` and define a `systemPath`. This is not recommended, however, but leads us to explaining the following elements:

classifier:
The classifier distinguishes artifacts that were built from the same POM but differ in content. It is some optional and arbitrary string that - if present - is appended to the artifact name just after the version number.

As a motivation for this element, consider for example a project that offers an artifact targeting Java 11 but at the same time also an artifact that still supports Java 1.8. The first artifact could be equipped with the classifier `jdk11` and the second one with `jdk8` such that clients can choose which one to use.

Another common use case for classifiers is to attach secondary artifacts to the project's main artifact. If you browse the Maven central repository, you will notice that the classifiers `sources` and `javadoc` are used to deploy the project source code and API docs along with the packaged class files.

- **type**:
  Corresponds to the chosen dependency type. This defaults to `jar`. While it usually represents the extension on the filename of the dependency, that is not always the case: a type can be mapped to a different extension and a classifier. The type often corresponds to the packaging used, though this is also not always the case. Some examples are `jar`, `ejb-client` and `test-jar`: see default artifact handlers for a list. New types can be defined by plugins that set `extensions` to true, so this is not a complete list.

  scope:
  This element refers to the classpath of the task at hand (compiling and runtime, testing, etc.) as well as how to limit the transitivity of a dependency. There are five scopes available:
  - **compile** - this is the default scope, used if none is specified. Compile dependencies are available in all classpaths. Furthermore, those dependencies are propagated to dependent projects.
  - **provided** - this is much like compile, but indicates you expect the JDK or a container to provide it at runtime. It is only available on the compilation and test classpath, and is not transitive.
  - **runtime** - this scope indicates that the dependency is not required for compilation, but is for execution. It is in the runtime and test classpaths, but not the compile classpath.
  - **test** - this scope indicates that the dependency is not required for normal use of the application, and is only available for the test compilation and execution phases. It is not transitive.
  - **system** - this scope is similar to `provided` except that you have to provide the JAR which contains it explicitly. The artifact is always available and is not looked up in a repository.
- **systemPath**:
  is used *only* if the dependency `scope` is `system`. Otherwise, the build will fail if this element is set. The path must be absolute, so it is recommended to use a property to specify the machine-specific path (more on `properties` below), such as `${java.home}/lib`. Since it is assumed that system scope dependencies are installed *a priori*, Maven does not check the repositories for the project, but instead checks to ensure that the file exists. If not, Maven fails the build and suggests that you download and install it manually.

  optional:
  Marks a dependency optional when this project itself is a dependency. For example, imagine a project A that depends upon project B to compile a portion of code that may not be used at runtime, then we may have no need for project B for all project. So if project X adds project A as its own dependency, then Maven does not need to install project B at all. Symbolically, if => represents a required dependency, and `-->` represents optional, although A=>B may be the case when building A X=>A-->B would be the case when building X.

  In the shortest terms, `optional` lets other projects know that, when you use this project, you do not require this dependency in order to work correctly.

Dependency Management

Dependencies can be managed in the `dependencyManagement` section to affect the resolution of dependencies which are not fully qualified or to enforce the usage of a specific transitive dependency version. Further information in Introduction to the Dependency Mechanism.

Dependency Version Requirement Specification

Dependencies' `version` elements define version requirements, which are used to compute dependency versions. Soft requirements can be replaced by different versions of the same artifact found elsewhere in the dependency graph. Hard requirements mandate a particular version or versions and override soft

requirements. If there are no versions of a dependency that satisfy all the hard requirements for that artifact, the build fails.

Version requirements have the following syntax:

- `1.0`: Soft requirement for 1.0. Use 1.0 if no other version appears earlier in the dependency tree.
- `[1.0]`: Hard requirement for 1.0. Use 1.0 and only 1.0.
- `(,1.0]`: Hard requirement for any version <= 1.0.
- `[1.2,1.3]`: Hard requirement for any version between 1.2 and 1.3 inclusive.
- `[1.0,2.0)`: 1.0 <= x < 2.0; Hard requirement for any version between 1.0 inclusive and 2.0 exclusive.
- `[1.5,)`: Hard requirement for any version greater than or equal to 1.5.
- `(,1.0],[1.2,)`: Hard requirement for any version less than or equal to 1.0 than or greater than or equal to 1.2, but not 1.1. Multiple requirements are separated by commas.

  `(,1.1),(1.1,)`: Hard requirement for any version except 1.1; for example because 1.1 has a critical vulnerability.

  Maven picks the highest version of each project that satisfies all the hard requirements of the dependencies on that project. If no version satisfies all the hard requirements, the build fails.

Version Order Specification:

If version strings are syntactically correct Semantic Versioning 1.0.0 version numbers, then in almost all cases version comparison follows the precedence rules outlined in that specification. These versions are the commonly encountered alphanumeric ASCII strings such as 2.15.2-alpha. More precisely, this is true if both version numbers to be compared match the "valid semver" production in the BNF grammar in the semantic versioning specification. Maven does not consider any semantics implied by that specification.

Important: This is only true for Semantic Versioning *1.0.0*. The Maven version order algorithm is not compatible with Semantic Versioning *2.0.0*. In particular, Maven does not special case the plus sign or consider build identifiers.

When version strings do not follow semantic versioning, a more complex set of rules is required. The Maven coordinate is split in tokens between dots ('.'), hyphens ('-'), underscore ('_') and transitions between digits and characters. The separator is recorded and will have effect on the order. A transition between digits and characters is equivalent to a hyphen. Empty tokens are replaced with "0". This gives a sequence of version numbers (numeric tokens) and version qualifiers (non-numeric tokens) with "`.`" or "`-`" prefixes. Versions are expected to start with numbers.

Splitting and Replacing Examples:

- `1-1.foo-bar1baz-.1` -> `1-1.foo-bar-1-baz-0.1`

Then, starting from the end of the version, the trailing "null" values (`0`, "`""`", "`final`", "`ga`") are trimmed. This process is repeated at each remaining hyphen from end to start.

Trimming Examples:

- `1.0.0` -> `1`
- `1.ga` -> `1`
- `1.final` -> `1`
- `1.0` -> `1`
- `1.` -> `1`
- `1-` -> `1`
- `1.0.0-foo.0.0` -> `1-foo`
- `1.0.0-0.0.0` -> `1`

The version order is the `lexicographical order` on this sequence of prefixed tokens, the shorter one padded with enough "null" values with matching prefix to have the same length as the longer one. Padded "null" values depend on the prefix of the other version: 0 for '.', "" for '-'. The prefixed token order is:

- if the prefix is the same, then compare the token:
  - Numeric tokens have the natural order.
  - Non-numeric tokens ("qualifiers") have the alphabetical order, except for the following tokens which come first in this order:

    "`alpha`" < "`beta`" < "`milestone`" < "`rc`" = "`cr`" < "`snapshot`" < "" = "`final`" = "`ga`" < "`sp`"

    - the "`alpha`", "`beta`" and "`milestone`" qualifiers can respectively be shortened to "a", "b" and "m" when directly followed by a number.
- else "`.qualifier`" < "`-qualifier`" < "`-number`" < "`.number`"
- `alpha` = a <<<*beta*>> = `b` <<<*milestone*>> = `m` <<<*rc*>> = `cr` <<<*snapshot*>> '<<<>>' = `final` = `ga` = `release` < `sp`

Following semver rules is encouraged, and some qualifiers are discouraged:

- Prefer '`alpha`', '`beta`', and '`milestone`' qualifiers over '`ea`' and '`preview`'.
- Prefer '`1.0.0-RC1`" over '`1.0.0.RC1`'.
- The usage of '`CR`' qualifier is discouraged. Use '`RC`' instead.

- The usage of '`final`', '`ga`', and '`release`' qualifiers is discouraged. Use no qualifier instead.
- The usage of 'SP' qualifier is discouraged. Increment the patch version instead.

End Result Examples:

- "`1`" < "`1.1`" (number padding)
- "`1-snapshot`" < "`1`" < "`1-sp`" (qualifier padding)
- "`1-foo2`" < "`1-foo10`" (correctly automatically "switching" to numeric order)
- "`1.foo`" = "`1-foo`" < "`1-1`" = "`1.1`"
- "`1.ga`" = "`1-ga`" = "`1-0`" = "`1.0`" = "`1`" (removing of trailing "null" values)
- "`1-sp`" > "`1-ga`"
- "`1-sp.1`" > "`1-ga.1`"
- "`1-sp-1`" > "`1-ga-1`"
  "`1-a1`" = "`1-alpha-1`"

  Note: Contrary to what was stated in some design documents, for version order, snapshots are not treated differently than releases or any other qualifier.

  Note: As `2.0-rc1` < `2.0`, the version requirement `[1.0,2.0)` excludes `2.0` but includes version `2.0-rc1`, which is contrary to what most people expect. In addition, Gradle interprets it differently, resulting in different dependency trees for the same POM. If the intention is to restrict it to *1.** versions, the better version requirement is `[1,1.999999)`.

Version Order Testing:

The maven distribution includes a tool to check version order. It was used to produce the examples in the previous paragraphs. Feel free to run it yourself when in doubt. You can run it like this:

```
java -jar ${MAVEN_HOME}/lib/maven-artifact-3.9.7.jar [versions...]
```

example:

```
$ java -jar ./lib/maven-artifact-3.9.7.jar  1 2 1.1
Display parameters as parsed by Maven (in canonical form and as a list of tokens) and comparison result:
1. 1 -> 1; tokens: [1]
  1 < 2
2. 2 -> 2; tokens: [2]
  2 > 1.1
3. 1.1 -> 1.1; tokens: [1, 1]
```

Exclusions

It is sometimes useful to limit a dependency's transitive dependencies. A dependency may have incorrectly specified scopes, or dependencies that conflict with other dependencies in your project. Exclusions tell Maven not to include a specified artifact in the classpath even if it is a dependency of one or more of this project's dependencies (a transitive dependency). For example, `maven-embedder` depends on `maven-core`. Suppose you want to depend on maven-embedder but do not want to include maven-core or its dependencies in the classpath. Then add `maven-core` as an `exclusion` in the element that declares the dependency on maven-embedder:

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <dependencies>
   <dependency>
     <groupId>org.apache.maven</groupId>
     <artifactId>maven-embedder</artifactId>
     <version>3.9.7</version>
     <exclusions>
       <exclusion>
         <groupId>org.apache.maven</groupId>
         <artifactId>maven-core</artifactId>
       </exclusion>
     </exclusions>
   </dependency>
   ...
 </dependencies>
 ...
</project>
```

This only removes the path to maven-core from this one dependency. If maven-core appears as a direct or transitive dependency elsewhere in the POM, it can still be added to the classpath.

Wildcard excludes make it easy to exclude all of a dependency's transitive dependencies. In the case below, you may be working with the maven-embedder and you want to manage the dependencies you use, so you exclude all the transitive dependencies:

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <dependencies>
   <dependency>
     <groupId>org.apache.maven</groupId>
     <artifactId>maven-embedder</artifactId>
     <version>3.8.6</version>
     <exclusions>
       <exclusion>
         <groupId>*</groupId>
         <artifactId>*</artifactId>
       </exclusion>
     </exclusions>
   </dependency>
   ...
 </dependencies>
 ...
</project>
```

- **exclusions**: Exclusions contain one or more `exclusion` elements, each containing a `groupId` and `artifactId` denoting a dependency to exclude. Unlike `optional`, which may or may not be installed and used, `exclusions` actively remove artifacts from the dependency tree.

Inheritance

One powerful addition that Maven brings to build management is the concept of project inheritance. Although in build systems such as Ant inheritance can be simulated, Maven makes project inheritance explicit in the project object model.

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>

 <groupId>org.codehaus.mojo</groupId>
 <artifactId>my-parent</artifactId>
 <version>2.0</version>
 <packaging>pom</packaging>
</project>
```

The `packaging` type is required to be `pom` for *parent* and *aggregation* (multi-module) projects. These types define the goals bound to a set of lifecycle stages. For example, if packaging is `jar`, then the `package` phase will execute the `jar:jar` goal. Now we may add values to the parent POM, which will be inherited by its children. Most elements from the parent POM are inherited by its children, including:

- groupId
- version
- description
- url
- inceptionYear
- organization
- licenses
- developers
- contributors
- mailingLists
- scm
- issueManagement
- ciManagement
- properties
- dependencyManagement
- dependencies
- repositories
- pluginRepositories
- build
  - plugin executions with matching ids
  - plugin configuration

    - etc.
- reporting

Notable elements which are `not` inherited include:

- artifactId
- name
- prerequisites
- profiles (but the effects of active profiles from parent POMs are)

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>

 <parent>
   <groupId>org.codehaus.mojo</groupId>
   <artifactId>my-parent</artifactId>
   <version>2.0</version>
   <relativePath>../my-parent</relativePath>
 </parent>

 <artifactId>my-project</artifactId>
</project>
```

Notice the `relativePath` element. It is not required, but may be used as a signifier to Maven to first search the path given for this project's parent, before searching the local and then remote repositories.

To see inheritance in action, just have a look at the ASF or Maven parent POM's.

Detailed inheritance rules are outlined in Maven Model Builder. All URLs are transformed when being inherited by default. The other ones are just inherited as is. For plugin configuration you can overwrite the inheritance behaviour with the attributes `combine.children` or `combine.self` outlined in Plugins.

The Super POM

Similar to the inheritance of objects in object oriented programming, POMs that extend a parent POM inherit certain values from that parent. Moreover, just as Java objects ultimately inherit from `java.lang.Object`, all Project Object Models inherit from a base Super POM. The snippet below is the Super POM for Maven 3.5.4.

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

 <repositories>
   <repository>
     <id>central</id>
     <name>Central Repository</name>
     <url>https://repo.maven.apache.org/maven2</url>
     <layout>default</layout>
     <snapshots>
       <enabled>false</enabled>
     </snapshots>
   </repository>
 </repositories>

 <pluginRepositories>
   <pluginRepository>
     <id>central</id>
     <name>Central Repository</name>
     <url>https://repo.maven.apache.org/maven2</url>
     <layout>default</layout>
     <snapshots>
       <enabled>false</enabled>
     </snapshots>
     <releases>
       <updatePolicy>never</updatePolicy>
     </releases>
   </pluginRepository>
 </pluginRepositories>

 <build>
   <directory>${project.basedir}/target</directory>
```

```
  <outputDirectory>${project.build.directory}/classes</outputDirectory>
  <finalName>${project.artifactId}-${project.version}</finalName>
  <testOutputDirectory>${project.build.directory}/test-classes</testOutputDirectory>
  <sourceDirectory>${project.basedir}/src/main/java</sourceDirectory>
  <scriptSourceDirectory>${project.basedir}/src/main/scripts</scriptSourceDirectory>
  <testSourceDirectory>${project.basedir}/src/test/java</testSourceDirectory>
  <resources>
    <resource>
      <directory>${project.basedir}/src/main/resources</directory>
    </resource>
  </resources>
  <testResources>
    <testResource>
      <directory>${project.basedir}/src/test/resources</directory>
    </testResource>
  </testResources>
  <pluginManagement>
    <!-- NOTE: These plugins will be removed from future versions of the super POM -->
    <!-- They are kept for the moment as they are very unlikely to conflict with lifecycle mappings (MNG-4453) -->
    <plugins>
      <plugin>
        <artifactId>maven-antrun-plugin</artifactId>
        <version>1.3</version>
      </plugin>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <version>2.2-beta-5</version>
      </plugin>
      <plugin>
        <artifactId>maven-dependency-plugin</artifactId>
        <version>2.8</version>
      </plugin>
      <plugin>
        <artifactId>maven-release-plugin</artifactId>
        <version>2.5.3</version>
      </plugin>
    </plugins>
  </pluginManagement>
</build>

<reporting>
  <outputDirectory>${project.build.directory}/site</outputDirectory>
</reporting>

<profiles>
  <!-- NOTE: The release profile will be removed from future versions of the super POM -->
  <profile>
    <id>release-profile</id>

    <activation>
      <property>
        <name>performRelease</name>
        <value>true</value>
      </property>
    </activation>

    <build>
      <plugins>
        <plugin>
          <inherited>true</inherited>
          <artifactId>maven-source-plugin</artifactId>
          <executions>
            <execution>
              <id>attach-sources</id>
              <goals>
                <goal>jar-no-fork</goal>
              </goals>
            </execution>
          </executions>
```

```
        </plugin>
        <plugin>
          <inherited>true</inherited>
          <artifactId>maven-javadoc-plugin</artifactId>
          <executions>
            <execution>
              <id>attach-javadocs</id>
              <goals>
                <goal>jar</goal>
              </goals>
            </execution>
          </executions>
        </plugin>
        <plugin>
          <inherited>true</inherited>
          <artifactId>maven-deploy-plugin</artifactId>
          <configuration>
            <updateReleaseInfo>true</updateReleaseInfo>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </profile>
 </profiles>

</project>
```

You can take a look at how the Super POM affects your Project Object Model by creating a minimal `pom.xml` and executing on the command line: `mvn help:effective-pom`

Dependency Management

Besides inheriting certain top-level elements, parents have elements to configure values for child POMs and transitive dependencies. One of those elements is `dependencyManagement`.

> **dependencyManagement**: is used by a POM to help manage dependency information across all of its children. If the `my-parent` project uses `dependencyManagement` to define a dependency on `junit:junit:4.12`, then POMs inheriting from this one can set their dependency giving the `groupId=junit` and `artifactId=junit` only and Maven will fill in the `version` set by the parent. The benefits of this method are obvious. Dependency details can be set in one central location, which propagates to all inheriting POMs.
>
> Note that the version and scope of artifacts which are incorporated from transitive dependencies are also controlled by version specifications in a dependency management section. This can lead to unexpected consequences. Consider a case in which your project uses two dependences, `dep1` and `dep2`. `dep2` in turn also uses `dep1`, and requires a particular minimum version to function. If you then use `dependencyManagement` to specify an older version, `dep2` will be forced to use the older version, and fail. So, you must be careful to check the entire dependency tree to avoid this problem; `mvn dependency:tree` is helpful.

Aggregation (or Multi-Module)

A project with modules is known as a multi-module, or aggregator project. Modules are projects that this POM lists, and are executed as a group. A `pom` packaged project may aggregate the build of a set of projects by listing them as modules, which are relative paths to the directories or the POM files of those projects.

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>

 <groupId>org.codehaus.mojo</groupId>
 <artifactId>my-parent</artifactId>
 <version>2.0</version>
 <packaging>pom</packaging>

 <modules>
   <module>my-project</module>
   <module>another-project</module>
   <module>third-project/pom-example.xml</module>
 </modules>
</project>
```

You do not need to consider the inter-module dependencies yourself when listing the modules; i.e. the ordering of the modules given by the POM is not important. Maven will topologically sort the modules such that dependencies are always build before dependent modules.

To see aggregation in action, have a look at the Maven base POM.

A final note on Inheritance v. Aggregation

Inheritance and aggregation create a nice dynamic to control builds through a single, high-level POM. You often see projects that are both parents and aggregators. For example, the entire Maven core runs through a single base POM `org.apache.maven:maven`, so building the Maven project can be executed by a single command: `mvn compile`. However, an aggregator project and a parent project are both POM projects, they are not one and the same and should not be confused. A POM project may be inherited from - but does not necessarily have - any modules that it aggregates. Conversely, a POM project may aggregate projects that do not inherit from it.

Properties

Properties are the last required piece to understand POM basics. Maven properties are value placeholders, like properties in Ant. Their values are accessible anywhere within a POM by using the notation `${X}`, where `X` is the property. Or they can be used by plugins as default values, for example:

```
<project>
 ...
 <properties>
   <maven.compiler.source>1.7</maven.compiler.source>
   <maven.compiler.target>1.7</maven.compiler.target>
   <!-- Following project.-properties are reserved for Maven in will become elements in a future POM definition. -->
   <!-- Don't start your own properties properties with project. -->
   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
   <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
 </properties>
 ...
</project>
```

They come in five different styles:

`env.X`: Prefixing a variable with "env." will return the shell's environment variable. For example, `${env.PATH}` contains the PATH environment variable.

Note: While environment variables themselves are case-insensitive on Windows, lookup of properties is case-sensitive. In other words, while the Windows shell returns the same value for `%PATH%` and `%Path%`, Maven distinguishes between `${env.PATH}` and `${env.Path}`. **The names of environment variables are normalized to all upper-case** for the sake of reliability.

2. `project.x`: A dot (.) notated path in the POM will contain the corresponding element's value. For example: `<project><version>1.0</version></project>` is accessible via `${project.version}`.
3. `settings.x`: A dot (.) notated path in the `settings.xml` will contain the corresponding element's value. For example: `<settings><offline>false</offline></settings>` is accessible via `${settings.offline}`.
4. Java System Properties: All properties accessible via `java.lang.System.getProperties()` are available as POM properties, such as `${java.home}`.
5. `x`: Set within a `<properties />` element in the POM. The value of `<properties><someVar>value</someVar></properties>` may be used as `${someVar}`.

Build Settings

Beyond the basics of the POM given above, there are two more elements that must be understood before claiming basic competency of the POM. They are the `build` element, that handles things like declaring your project's directory structure and managing plugins; and the `reporting` element, that largely mirrors the build element for reporting purposes.

Build

According to the POM 4.0.0 XSD, the `build` element is conceptually divided into two parts: there is a `BaseBuild` type which contains the set of elements common to both `build` elements (the top-level build element under `project` and the build element under `profiles`, covered below); and there is the `Build` type, which contains the `BaseBuild` set as well as more elements for the top level definition. Let us begin with an analysis of the common elements between the two.

Note: These different `build` *elements may be denoted "project build" and "profile build".*

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <!-- "Project Build" contains more elements than just the BaseBuild set -->
 <build>...</build>

 <profiles>
   <profile>
     <!-- "Profile Build" contains a subset of "Project Build"s elements -->
     <build>...</build>
```

```
   </profile>
 </profiles>
</project>
```

The BaseBuild Element Set

`BaseBuild` is exactly as it sounds: the base set of elements between the two `build` elements in the POM.

```
<build>
 <defaultGoal>install</defaultGoal>
 <directory>/home/jenkins/82467a7c/workspace/aven_maven-box_maven-site_master/target</directory>
 <finalName>${artifactId}-${version}</finalName>
 <filters>
   <filter>filters/filter1.properties</filter>
 </filters>
 ...
</build>
```

- **defaultGoal**: the default goal or phase to execute if none is given. If a goal is given, it should be defined as it is in the command line (such as `jar:jar`). The same goes for if a phase is defined (such as install).
- **directory**: This is the directory where the build will dump its files or, in Maven parlance, the build's target. It aptly defaults to `${project.basedir}/target`.
- **finalName**: This is the name of the bundled project when it is finally built (sans the file extension, for example: `my-project-1.0.jar`). It defaults to `${artifactId}-${version}`. The term "finalName" is kind of a misnomer, however, as plugins that build the bundled project have every right to ignore/modify this name (but they usually do not). For example, if the `maven-jar-plugin` is configured to give a jar a `classifier` of `test`, then the actual jar defined above will be built as `my-project-1.0-test.jar`.

  filter: Defines `*.properties` files that contain a list of properties that apply to resources which accept their settings (covered below). In other words, the "`name=value`" pairs defined within the filter files replace `${name}` strings within resources on build. The example above defines the `filter1.properties` file under the `filters/` directory. Maven's default filter directory is `${project.basedir}/src/main/filters/`.

  For a more comprehensive look at what filters are and what they can do, take a look at the quick start guide.

Resources

Another feature of `build` elements is specifying where resources exist within your project. Resources are not (usually) code. They are not compiled, but are items meant to be bundled within your project or used for various other reasons, such as code generation.

For example, a Plexus project requires a `configuration.xml` file (which specifies component configurations to the container) to live within the `META-INF/plexus` directory. Although we could just as easily place this file within `src/main/resources/META-INF/plexus`, we want instead to give Plexus its own directory of `src/main/plexus`. In order for the JAR plugin to bundle the resource correctly, you would specify resources similar to the following:

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 <build>
   ...
   <resources>
     <resource>
       <targetPath>META-INF/plexus</targetPath>
       <filtering>false</filtering>
       <directory>/home/jenkins/82467a7c/workspace/aven_maven-box_maven-site_master/src/main/plexus</directory>
       <includes>
         <include>configuration.xml</include>
       </includes>
       <excludes>
         <exclude>**/*.properties</exclude>
       </excludes>
     </resource>
   </resources>
   <testResources>
     ...
   </testResources>
   ...
 </build>
</project>
```

- **resources**: is a list of resource elements that each describe what and where to include files associated with this project.
- **targetPath**: Specifies the directory structure to place the set of resources from a build. Target path defaults to the base directory. A commonly specified target path for resources that will be packaged in a JAR is META-INF.

- **filtering**: is `true` or `false`, denoting if filtering is to be enabled for this resource. Note, that filter `*.properties` files do not have to be defined for filtering to occur - resources can also use properties that are by default defined in the POM (such as ${project.version}), passed into the command line using the "-D" flag (for example, "`-Dname=value`") or are explicitly defined by the properties element. Filter files were covered above.
- **directory**: This element's value defines where the resources are to be found. The default directory for a build is `${project.basedir}/src/main/resources`.
- **includes**: A set of files patterns which specify the files to include as resources under that specified directory, using * as a wildcard.
- **excludes**: The same structure as `includes`, but specifies which files to ignore. In conflicts between `include` and `exclude`, `exclude` wins.
- **testResources**: The `testResources` element block contains `testResource` elements. Their definitions are similar to `resource` elements, but are naturally used during test phases. The one difference is that the default (Super POM defined) test resource directory for a project is `${project.basedir}/src/test/resources`. Test resources are not deployed.

Plugins

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 <build>
   ...
   <plugins>
     <plugin>
       <groupId>org.apache.maven.plugins</groupId>
       <artifactId>maven-jar-plugin</artifactId>
       <version>2.6</version>
       <extensions>false</extensions>
       <inherited>true</inherited>
       <configuration>
         <classifier>test</classifier>
       </configuration>
       <dependencies>...</dependencies>
       <executions>...</executions>
     </plugin>
   </plugins>
 </build>
</project>
```

Beyond the standard coordinate of `groupId:artifactId:version`, there are elements which configure the plugin or this builds interaction with it.

- **extensions**: `true` or `false`, whether or not to load extensions of this plugin. It is by default false. Extensions are covered later in this document.
- **inherited**: `true` or `false`, whether or not this plugin configuration should apply to POMs which inherit from this one. Default value is `true`.

  configuration: This is specific to the individual plugin. Without going too in depth into the mechanics of how plugins work, suffice it to say that whatever properties that the plugin Mojo may expect (these are getters and setters in the Java Mojo bean) can be specified here. In the above example, we are setting the classifier property to test in the `maven-jar-plugin`'s Mojo. It may be good to note that all configuration elements, wherever they are within the POM, are intended to pass values to another underlying system, such as a plugin. In other words: values within a `configuration` element are never explicitly required by the POM schema, but a plugin goal has every right to require configuration values.

  If your POM declares a parent, it inherits plugin configuration from either the **build/plugins** or **pluginManagement** sections of the parent.

  default configuration inheritance:

  To illustrate, consider the following fragment from a parent POM:

  ```
  <plugin>
   <groupId>my.group</groupId>
   <artifactId>my-plugin</artifactId>
   <configuration>
     <items>
       <item>parent-1</item>
       <item>parent-2</item>
     </items>
     <properties>
       <parentKey>parent</parentKey>
     </properties>
   </configuration>
  </plugin>
  ```

  And consider the following plugin configuration from a project that uses that parent as its parent:

  ```
  <plugin>
   <groupId>my.group</groupId>
   <artifactId>my-plugin</artifactId>
   <configuration>
  ```

```
   <items>
     <item>child-1</item>
   </items>
   <properties>
     <childKey>child</childKey>
   </properties>
 </configuration>
</plugin>
```

The default behavior is to merge the content of the **configuration** element according to element name. If the child POM has a particular element, that value becomes the effective value. if the child POM does not have an element, but the parent does, the parent value becomes the effective value. Note that this is purely an operation on XML; no code or configuration of the plugin itself is involved. Only the elements, not their values, are involved.

Applying those rules to the example, Maven comes up with:

```
<plugin>
 <groupId>my.group</groupId>
 <artifactId>my-plugin</artifactId>
 <configuration>
   <items>
     <item>child-1</item>
   </items>
   <properties>
     <childKey>child</childKey>
     <parentKey>parent</parentKey>
   </properties>
 </configuration>
</plugin>
```

advanced configuration inheritance: `combine.children` and `combine.self`

You can control how child POMs inherit configuration from parent POMs by adding attributes to the children of the **configuration** element. The attributes are `combine.children` and `combine.self`. Use these attributes in a child POM to control how Maven combines plugin configuration from the parent with the explicit configuration in the child.

Here is the child configuration with illustrations of the two attributes:

```
<configuration>
 <items combine.children="append">
   <!-- combine.children="merge" is the default -->
   <item>child-1</item>
 </items>
 <properties combine.self="override">
   <!-- combine.self="merge" is the default -->
   <childKey>child</childKey>
 </properties>
</configuration>
```

Now, the effective result is the following:

```
<configuration>
 <items combine.children="append">
   <item>parent-1</item>
   <item>parent-2</item>
   <item>child-1</item>
 </items>
 <properties combine.self="override">
   <childKey>child</childKey>
 </properties>
</configuration>
```

combine.children="append" results in the concatenation of parent and child elements, in that order. **combine.self="override"**, on the other hand, completely suppresses parent configuration. You cannot use both **combine.self="override"** and **combine.children="append"** on an element; if you try, *override* will prevail.

Note that these attributes only apply to the configuration element they are declared on, and are not propagated to nested elements. That is if the content of an *item* element from the child POM was a complex structure instead of text, its sub-elements would still be subject to the default merge strategy unless they were themselves marked with attributes.

The `combine.*` attributes are inherited from parent to child POMs. Take care when adding those attributes to a parent POM as this might affect child or grand-child POMs.

- **dependencies**: Dependencies are seen a lot within the POM, and are an element under all plugins element blocks. The dependencies have the same structure and function as under that base build. The major difference in this case is that instead of applying as dependencies of the project, they now apply as dependencies of the plugin that they are under. The power of this is to alter the dependency list of a plugin, perhaps by removing an unused runtime dependency via `exclusions`, or by altering the version of a required dependency. See above under **Dependencies** for more information.

  executions: It is important to keep in mind that a plugin may have multiple goals. Each goal may have a separate configuration, possibly even binding a plugin's goal to a different phase altogether. `executions` configure the `execution` of a plugin's goals.

  For example, suppose you wanted to bind the `antrun:run` goal to the `verify` phase. We want the task to echo the build directory, as well as avoid passing on this configuration to its children (assuming it is a parent) by setting `inherited` to `false`. You would get an `execution` like this:

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <build>
   <plugins>
     <plugin>
       <artifactId>maven-antrun-plugin</artifactId>
       <version>1.1</version>
       <executions>
         <execution>
           <id>echodir</id>
           <goals>
             <goal>run</goal>
           </goals>
           <phase>verify</phase>
           <inherited>false</inherited>
           <configuration>
             <tasks>
               <echo>Build Dir: /home/jenkins/82467a7c/workspace/aven_maven-box_maven-site_master/target</echo>
             </tasks>
           </configuration>
         </execution>
       </executions>

     </plugin>
   </plugins>
 </build>
</project>
```

  - **id**: Self explanatory. It specifies this execution block between all of the others. When the phase is run, it will be shown in the form: `[plugin:goal execution: id]`. In the case of this example: `[antrun:run execution: echodir]`
  - **goals**: Like all pluralized POM elements, this contains a list of singular elements. In this case, a list of plugin `goals` which are being specified by this `execution` block.
  - **phase**: This is the phase that the list of goals will execute in. This is a very powerful option, allowing one to bind any goal to any phase in the build lifecycle, altering the default behavior of Maven.
  - **inherited**: Like the `inherited` element above, setting this to false will suppress Maven from passing this execution onto its children. This element is only meaningful to parent POMs.
  - **configuration**: Same as above, but confines the configuration to this specific list of goals, rather than all goals under the plugin.

Plugin Management

- **pluginManagement**: is an element that is seen along side plugins. Plugin Management contains plugin elements in much the same way, except that rather than configuring plugin information for this particular project build, it is intended to configure project builds that inherit from this one. However, this only configures plugins that are actually referenced within the plugins element in the children or in the current POM. The children have every right to override `pluginManagement` definitions.

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <build>
   ...
   <pluginManagement>
     <plugins>
       <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-jar-plugin</artifactId>
         <version>2.6</version>
         <executions>
           <execution>
```

```
            <id>pre-process-classes</id>
            <phase>compile</phase>
            <goals>
              <goal>jar</goal>
            </goals>
            <configuration>
              <classifier>pre-process</classifier>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </pluginManagement>
  ...
 </build>
</project>
```

If we added these specifications to the plugins element, they would apply only to a single POM. However, if we apply them under the `pluginManagement` element, then this POM *and all inheriting POMs* that add the `maven-jar-plugin` to the build will get the `pre-process-classes` execution as well. So rather than the above mess included in every child `pom.xml`, only the following is required:

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <build>
   ...
   <plugins>
     <plugin>
       <groupId>org.apache.maven.plugins</groupId>
       <artifactId>maven-jar-plugin</artifactId>
     </plugin>
   </plugins>
   ...
 </build>
</project>
```

The Build Element Set

The `Build` type in the XSD denotes those elements that are available only for the "project build". Despite the number of extra elements (six), there are really only two groups of elements that project build contains that are missing from the profile build: directories and extensions.

Directories

The set of directory elements live in the parent build element, which set various directory structures for the POM as a whole. Since they do not exist in profile builds, these cannot be altered by profiles.

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <build>
   <sourceDirectory>/home/jenkins/82467a7c/workspace/aven_maven-box_maven-site_master/src/main/java</sourceDirectory>
   <scriptSourceDirectory>/home/jenkins/82467a7c/workspace/aven_maven-box_maven-site_master/src/main/scripts</scriptSourceDire
   <testSourceDirectory>/home/jenkins/82467a7c/workspace/aven_maven-box_maven-site_master/src/test/java</testSourceDirectory>
   <outputDirectory>/home/jenkins/82467a7c/workspace/aven_maven-box_maven-site_master/target/classes</outputDirectory>
   <testOutputDirectory>/home/jenkins/82467a7c/workspace/aven_maven-box_maven-site_master/target/test-classes</testOutputDirec
   ...
 </build>
</project>
```

If the values of a `*Directory` element above is set as an absolute path (when their properties are expanded) then that directory is used. Otherwise, it is relative to the base build directory: `${project.basedir}`. **Please note that the `scriptSourceDirectory` is nowhere used in Maven and is obsolete**.

Extensions

Extensions are a list of artifacts that are to be used in this build. They will be included in the running build's classpath. They can enable extensions to the build process (such as add an ftp provider for the Wagon transport mechanism), as well as make plugins active which make changes to the build lifecycle. In short, extensions are artifacts that are activated during build. The extensions do not have to actually do anything nor contain a Mojo. For this reason, extensions are excellent for specifying one out of multiple implementations of a common plugin interface.

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
 ...
 <build>
   ...
   <extensions>
     <extension>
       <groupId>org.apache.maven.wagon</groupId>
       <artifactId>wagon-ftp</artifactId>
       <version>1.0-alpha-3</version>
     </extension>
   </extensions>
   ...
 </build>
</project>
```

Reporting

Reporting contains the elements that correspond specifically for the `site` generation phase. Certain Maven plugins can generate reports defined and configured under the reporting element, for example: generating Javadoc reports. Much like the build element's ability to configure plugins, 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 a plugin `configuration` under the `reporting` element works as `build` plugin `configuration`, although the opposite is not true (a `build` plugin `configuration` does not affect a `reporting` plugin).

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 plugin like any other, it may also be suppressed in the following, more verbose, way, which effectively turns off project-info reports.

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <reporting>
   <outputDirectory>/home/jenkins/82467a7c/workspace/aven_maven-box_maven-site_master/target/site</outputDirectory>
   <plugins>
     <plugin>
       <artifactId>maven-project-info-reports-plugin</artifactId>
       <version>2.0.1</version>
       <reportSets>
         <reportSet></reportSet>
       </reportSets>
     </plugin>
   </plugins>
 </reporting>
 ...
</project>
```

The other difference is the `outputDirectory` element under `plugin`. In the case of reporting, the output directory is `${project.basedir}/target/site` by default.

Report Sets

It is important to keep in mind that an individual plugin may have multiple goals. Each goal may have a separate configuration. Report sets configure execution of a report plugin's goals. Does this sound familiar - deja-vu? The same thing was said about build's `execution` element with one difference: you cannot bind a report to another phase. Sorry.

For example, suppose you wanted to configure the `javadoc:javadoc` goal to link to "http://java.sun.com/j2se/1.5.0/docs/api/", but only the `javadoc` goal (not the goal `maven-javadoc-plugin:jar`). We would also like this configuration passed to its children, and set `inherited` to true. The `reportSet` would resemble the following:

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <reporting>
   <plugins>
     <plugin>
       ...
       <reportSets>
         <reportSet>
           <id>sunlink</id>
           <reports>
```

```
            <report>javadoc</report>
          </reports>
          <inherited>true</inherited>
          <configuration>
            <links>
              <link>http://java.sun.com/j2se/1.5.0/docs/api/</link>
            </links>
          </configuration>
        </reportSet>
      </reportSets>
    </plugin>
  </plugins>
 </reporting>
 ...
</project>
```

Between build `executions` and reporting `reportSets`, it should be clear now as to why they exist. In the simplest sense, they drill down in configuration. The POM must have a way not only to configure plugins, but must also configure the goals of those plugins. That is where these elements come in, giving the POM ultimate granularity in control of its build destiny.

More Project Information

Several elements do not affect the build, but rather document the project for the convenience of developers. Many of these elements are used to fill in project details when generating the project's web site. However, like all POM declarations, plugins can use them for anything. The following are the simplest elements:

- **name**: Projects tend to have conversational names, beyond the `artifactId`. The Sun engineers did not refer to their project as "java-1.5", but rather just called it "Tiger". Here is where to set that value.
- **description**: A short, human readable description of the project. Although this should not replace formal documentation, a quick comment to any readers of the POM is always helpful.
- **url**: The project's home page.
- **inceptionYear**: The year the project was first created.

Licenses

```
<licenses>
 <license>
   <name>Apache-2.0</name>
   <url>https://www.apache.org/licenses/LICENSE-2.0.txt</url>
   <distribution>repo</distribution>
   <comments>A business-friendly OSS license</comments>
 </license>
</licenses>
```

Licenses are legal documents defining how and when a project (or parts of a project) may be used. A project should list licenses that apply directly to this project, and not list licenses that apply to the project's dependencies.

- **name**, **url** and **comments**: are self explanatory, and have been encountered before in other contexts. Using an SPDX identifier as the license **name** is recommended. The fourth license element is:
- **distribution**: This describes how the project may be legally distributed. The two stated methods are repo (they may be downloaded from a Maven repository) or manual (they must be manually installed).

Organization

Most projects are run by some sort of organization (business, private group, etc.). Here is where the most basic information is set.

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <organization>
   <name>Codehaus Mojo</name>
   <url>http://mojo.codehaus.org</url>
 </organization>
</project>
```

Developers

All projects consist of files that were created, at some time, by a person. Like the other systems that surround a project, so to do the people involved with a project have a stake in the project. Developers are presumably members of the project's core development. Note that, although an organization may have many

developers (programmers) as members, it is not good form to list them all as developers, but only those who are immediately responsible for the code. A good rule of thumb is, if the person should not be contacted about the project, they do not need to be listed here.

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <developers>
   <developer>
     <id>jdoe</id>
     <name>John Doe</name>
     <email>jdoe@example.com</email>
     <url>http://www.example.com/jdoe</url>
     <organization>ACME</organization>
     <organizationUrl>http://www.example.com</organizationUrl>
     <roles>
       <role>architect</role>
       <role>developer</role>
     </roles>
     <timezone>America/New_York</timezone>
     <properties>
       <picUrl>http://www.example.com/jdoe/pic</picUrl>
     </properties>
   </developer>
 </developers>
 ...
</project>
```

- **id**, **name**, **email**: These correspond to the developer's ID (presumably some unique ID across an organization), the developer's name and email address.
- **organization**, **organizationUrl**: As you probably guessed, these are the developer's organization name and it's URL, respectively.
- **roles**: A `role` should specify the standard actions that the person is responsible for. Like a single person can wear many hats, a single person can take on multiple `roles`.
- **timezone**: A valid time zone ID like `America/New_York` or `Europe/Berlin`, or a numerical offset in hours (and fraction) from UTC where the developer lives, e.g., `-5` or `+1`. Time zone IDs are highly preferred because they are not affected by DST and time zone shifts. Refer to the IANA for the official time zone database and a listing in Wikipedia.
- **properties**: This element is where any other properties about the person goes. For example, a link to a personal image or an instant messenger handle. Different plugins may use these properties, or they may simply be for other developers who read the POM.

Contributors

Contributors are like developers yet play an ancillary role in a project's lifecycle. Perhaps the contributor sent in a bug fix, or added some important documentation. A healthy open source project will likely have more contributors than developers.

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <contributors>
   <contributor>
     <name>Noelle</name>
     <email>some.name@gmail.com</email>
     <url>http://noellemarie.com</url>
     <organization>Noelle Marie</organization>
     <organizationUrl>http://noellemarie.com</organizationUrl>
     <roles>
       <role>tester</role>
     </roles>
     <timezone>America/Vancouver</timezone>
     <properties>
       <gtalk>some.name@gmail.com</gtalk>
     </properties>
   </contributor>
 </contributors>
 ...
</project>
```

Contributors contain the same set of elements than developers sans the `id` element.

Environment Settings

Issue Management

This defines the defect tracking system (*Bugzilla*, *TestTrack*, *ClearQuest*, etc) used. Although there is nothing stopping a plugin from using this information for something, it's primarily used for generating project documentation.

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <issueManagement>
   <system>Bugzilla</system>
   <url>http://127.0.0.1/bugzilla/</url>
 </issueManagement>
 ...
</project>
```

Continuous Integration Management

Continuous integration build systems based upon triggers or timings (such as, hourly or daily) have grown in favor over manual builds in the past few years. As build systems have become more standardized, so have the systems that run the trigger those builds. Although the majority of the configuration is up to the specific program used (Continuum, Cruise Control, etc.), there are a few configurations which may take place within the POM. Maven has captured a few of the recurring settings within the set of notifier elements. A notifier is the manner in which people are notified of certain build statuses. In the following example, this POM is setting a notifier of type `mail` (meaning email), and configuring the email address to use on the specified triggers `sendOnError`, `sendOnFailure`, and not `sendOnSuccess` or `sendOnWarning`.

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <ciManagement>
   <system>continuum</system>
   <url>http://127.0.0.1:8080/continuum</url>
   <notifiers>
     <notifier>
       <type>mail</type>
       <sendOnError>true</sendOnError>
       <sendOnFailure>true</sendOnFailure>
       <sendOnSuccess>false</sendOnSuccess>
       <sendOnWarning>false</sendOnWarning>
       <configuration><address>continuum@127.0.0.1</address></configuration>
     </notifier>
   </notifiers>
 </ciManagement>
 ...
</project>
```

Mailing Lists

Mailing lists are a great tool for keeping in touch with people about a project. Most mailing lists are for developers and users.

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <mailingLists>
   <mailingList>
     <name>User List</name>
     <subscribe>user-subscribe@127.0.0.1</subscribe>
     <unsubscribe>user-unsubscribe@127.0.0.1</unsubscribe>
     <post>user@127.0.0.1</post>
     <archive>http://127.0.0.1/user/</archive>
     <otherArchives>
       <otherArchive>http://base.google.com/base/1/127.0.0.1</otherArchive>
     </otherArchives>
   </mailingList>
 </mailingLists>
 ...
</project>
```

- **subscribe**, **unsubscribe**: There elements specify the email addresses which are used for performing the relative actions To subscribe to the user list above, a user would send an email to user-subscribe@127.0.0.1.

- **archive**: This element specifies the url of the archive of old mailing list emails, if one exists. If there are mirrored archives, they can be specified under otherArchives.
- **post**: The email address which one would use in order to post to the mailing list. Note that not all mailing lists have the ability to post to (such as a build failure list).

SCM

SCM (Software Configuration Management, also called Source Code/Control Management or, succinctly, version control) is an integral part of any healthy project. If your Maven project uses an SCM system (it does, doesn't it?) then here is where you would place that information into the POM.

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <scm>
   <connection>scm:svn:http://127.0.0.1/svn/my-project</connection>
   <developerConnection>scm:svn:https://127.0.0.1/svn/my-project</developerConnection>
   <tag>HEAD</tag>
   <url>http://127.0.0.1/websvn/my-project</url>
 </scm>
 ...
</project>
```

connection, **developerConnection**: The two connection elements convey to how one is to connect to the version control system through Maven. Where connection requires read access for Maven to be able to find the source code (for example, an update), developerConnection requires a connection that will give write access. The Maven project has spawned another project named Maven SCM, which creates a common API for any SCMs that wish to implement it. The most popular are CVS and Subversion, however, there is a growing list of other supported SCMs. All SCM connections are made through a common URL structure.

```
scm:[provider]:[provider_specific]
```

Where provider is the type of SCM system. For example, connecting to a CVS repository may look like this:

```
scm:cvs:pserver:127.0.0.1:/cvs/root:my-project
```

- **tag**: Specifies the tag that this project lives under. HEAD (meaning, the SCM root) is the default.
- **url**: A publicly browsable repository. For example, via ViewCVS.

Prerequisites

The POM may have certain prerequisites in order to execute correctly. The only element that exists as a prerequisite in POM 4.0.0 is the `maven` element, which takes a minimum version number.

Use Maven Enforcer Plugin's `requireMavenVersion` rule, or other rules for **build-time** prerequisites. For packaging `maven-plugin` this is still used at **run-time** to make sure that the minimum Maven version for the plugin is met (but only in the pom.xml of the referenced plugin).

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <prerequisites>
   <maven>2.0.6</maven>
 </prerequisites>
 ...
</project>
```

Repositories

Repositories are collections of artifacts which adhere to the Maven repository directory layout. In order to be a Maven repository artifact, a POM file must live within the structure `$BASE_REPO/groupId/artifactId/version/artifactId-version.pom`. `$BASE_REPO` can be local (file structure) or remote (base URL); the remaining layout will be the same. Repositories exist as a place to collect and store artifacts. Whenever a project has a dependency upon an artifact, Maven will first attempt to use a local copy of the specified artifact. If that artifact does not exist in the local repository, it will then attempt to download from a remote repository. The repository elements within a POM specify those alternate repositories to search.

The repository is one of the most powerful features of the Maven community. By default Maven searches the central repository at https://repo.maven.apache.org/maven2/. Additional repositories can be configured in the pom.xml `repositories` element.

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <repositories>
   <repository>
```

```
    <releases>
      <enabled>false</enabled>
    </releases>
    <snapshots>
      <enabled>true</enabled>
      <updatePolicy>always</updatePolicy>
      <checksumPolicy>fail</checksumPolicy>
    </snapshots>
    <name>Nexus Snapshots</name>
    <id>snapshots-repo</id>
    <url>https://oss.sonatype.org/content/repositories/snapshots</url>
    <layout>default</layout>
  </repository>
 </repositories>
 <pluginRepositories>
   ...
 </pluginRepositories>
 ...
</project>
```

- **releases**, **snapshots**: These are the policies for each type of artifact, Release or snapshot. With these two sets, a POM has the power to alter the policies for each type independent of the other within a single repository. For example, one may decide to enable only snapshot downloads, possibly for development purposes.
- **enabled**: `true` or `false` for whether this repository is enabled for the respective type (`releases` or `snapshots`). By default this is `true`.
- **updatePolicy**: This element specifies how often Maven tries to update its local repository from the remote repositories. Maven will compare the local POM's timestamp (stored in a repository's maven-metadata file) to the remote. The choices are: `always`, `daily` (default), `interval:X` (where X is an integer in minutes) or `never` (only downloads if not yet existing in the local repository). As this affects both artifacts and metadata (supposed to be changed in Maven 4) be careful with `never`, as metadata changes over time (even for release repositories).
- **checksumPolicy**: When Maven deploys files to the repository, it also deploys corresponding checksum files. Your options are to `ignore`, `fail`, or `warn` on missing or incorrect checksums. The default value is `warn`.
- **id**: The repository id is mandatory and connects the repository with the servers from `settings.xml`. Its default value is `default`. The id is used also in the local repository metadata to store the origin.
- **name**: An optional name for the repository. Used as label when emitting log messages related to this repository.
- **layout**: In the above description of repositories, it was mentioned that they all follow a common layout. This is mostly correct. The layout introduced with Maven 2 is the default layout for repositories used by Maven both 2 & 3. However, Maven 1.x had a different layout. Use this element to specify whether it is `default` or `legacy`. Its default value is `default`.

Plugin Repositories

Repositories are home to two major types of artifacts. The first are artifacts that are used as dependencies of other artifacts. These are the majority of artifacts that reside within central. The other type of artifact is plugins. Maven plugins are themselves a special type of artifact. Because of this, plugin repositories may be separated from other repositories (although, I have yet to hear a convincing argument for doing so). In any case, the structure of the `pluginRepositories` element block is similar to the `repositories` element. The `pluginRepository` elements each specify a remote location of where Maven can find new plugins.

Distribution Management

Distribution management acts precisely as it sounds: it manages the distribution of the artifact and supporting files generated throughout the build process.

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <distributionManagement>
   <repository>...</repository>
   <snapshotRepository>...<snapshotRepository>
   <site>...</site>
   <relocation>...</relocation>
   <downloadUrl>...</downloadUrl>
   <status>...</status>
 </distributionManagement>
 ...
</project>
```

Repository

Whereas the repositories element specifies in the POM the location and manner in which Maven may download remote artifacts for use by the current project, distributionManagement specifies where (and how) this project will get to a remote repository when it is deployed. The repository elements will be used for snapshot distribution if the snapshotRepository is not defined.

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <distributionManagement>
   <repository>
     <uniqueVersion>false</uniqueVersion>
     <id>corp1</id>
     <name>Corporate Repository</name>
     <url>scp://repo/maven2</url>
     <layout>default</layout>
   </repository>
   <snapshotRepository>
     <uniqueVersion>true</uniqueVersion>
     <id>propSnap</id>
     <name>Propellors Snapshots</name>
     <url>sftp://propellers.net/maven</url>
     <layout>legacy</layout>
   </snapshotRepository>
   ...
 </distributionManagement>
 ...
</project>
```

- **id**, **name**: The `id` is used to uniquely identify this repository amongst many, and the `name` is a human readable form.
- **uniqueVersion**: The unique version takes a `true` or `false` value to denote whether artifacts deployed to this repository should get a uniquely generated version number, or use the version number defined as part of the address.
- **url**: This is the core of the repository element. It specifies both the location and the transport protocol used to transfer a built artifact (and POM file, and checksum data) to the repository.
- **layout**: These are the same types and purpose as the layout element defined in the repository element. They are `default` and `legacy`.

Site Distribution

More than distribution to the repositories, `distributionManagement` is responsible for defining how to deploy the project's site and documentation.

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <distributionManagement>
   ...
   <site>
     <id>mojo.website</id>
     <name>Mojo Website</name>
     <url>scp://beaver.codehaus.org/home/projects/mojo/public_html/</url>
   </site>
   ...
 </distributionManagement>
 ...
</project>
```

- **id**, **name**, **url**: These elements are similar to their counterparts above in the `distributionManagement` `repository` element.

Relocation

```
<project xmlns="http://maven.apache.org/POM/4.0.0"1 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <distributionManagement>
   ...
   <relocation>
     <groupId>org.apache</groupId>
     <artifactId>my-project</artifactId>
     <version>1.0</version>
     <message>We have moved the Project under Apache</message>
   </relocation>
   ...
 </distributionManagement>
 ...
</project>
```

Projects are not static; they are living things (or dying things, as the case may be). A common thing that happens as projects grow, is that they are forced to move to more suitable quarters. For example, when your next wildly successful open source project moves under the Apache umbrella, it would be good to give users a heads-up that the project is being renamed to `org.apache:my-project:1.0`. Besides specifying the new address, it is also good form to provide a message explaining why.

downloadUrl

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <distributionManagement>
   ...
   <downloadUrl>http://mojo.codehaus.org/my-project/download.html</downloadUrl>
 </distributionManagement>
 ...
</project>
```

downloadUrl is the URL of the project's download page. If not given users will be referred to the homepage given by url. This is given to assist in locating artifacts that are not in the repository due to licensing restrictions. See for example MNG-2083 for a typical workflow.

status

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <distributionManagement>
   ...
   <status>deployed</status>
 </distributionManagement>
 ...
</project>
```

Warning! Like a baby bird in a nest, the status should never be touched by human hands! The reason for this is that Maven will set the status of the project when it is transported out to the repository. It is described here just for understanding, but should never be configured in your `pom.xml`.

Status valid values are as follows:

- **none**: No special status. This is the default for a POM.
- **converted**: The manager of the repository converted this POM from an earlier version to Maven 2.
- **partner**: This artifact has been synchronized with a partner repository.
- **deployed**: By far the most common status, meaning that this artifact was deployed from a Maven 2 or 3 instance. This is what you get when you manually deploy using the command-line deploy phase.
- **verified**: This project has been verified, and should be considered finalized.

Profiles

A new feature of the POM 4.0 is the ability of a project to change settings depending on the environment where it is being built. A `profile` element contains both an optional activation (a profile trigger) and the set of changes to be made to the POM if that profile has been activated. For example, a project built for a test environment may point to a different database than that of the final deployment. Or dependencies may be pulled from different repositories based upon the JDK version used. The elements of profiles are as follows:

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <profiles>
   <profile>
     <id>test</id>
     <activation>...</activation>
     <build>...</build>
     <modules>...</modules>
     <repositories>...</repositories>
     <pluginRepositories>...</pluginRepositories>
     <dependencies>...</dependencies>
     <reporting>...</reporting>
     <dependencyManagement>...</dependencyManagement>
     <distributionManagement>...</distributionManagement>
   </profile>
 </profiles>
</project>
```

Activation

Activations are the key of a profile. The power of a profile comes from its ability to modify the basic POM only under certain circumstances. Those circumstances are specified via an `activation` element.

```
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">
 ...
 <profiles>
   <profile>
     <id>test</id>
     <activation>
       <activeByDefault>false</activeByDefault>
       <jdk>1.5</jdk>
       <os>
         <name>Windows XP</name>
         <family>Windows</family>
         <arch>x86</arch>
         <version>5.1.2600</version>
       </os>
       <property>
         <name>sparrow-type</name>
         <value>African</value>
       </property>
       <file>
         <exists>${basedir}/file2.properties</exists>
         <missing>${basedir}/file1.properties</missing>
       </file>
     </activation>
     ...
   </profile>
 </profiles>
</project>
```

Before Maven 3.2.2 activation occurs when one or more of the specified criteria have been met. When the first positive result is encountered, processing stops and the profile is marked as active. Since Maven 3.2.2 activation occurs when all of the specified criteria have been met.

- **activeByDefault**: Is `false` by default. Boolean flag which determines if the profile is active by default. This flag is only evaluated if no other profile is explicitly activated via command line, `settings.xml` or implicitly activated through some other activator, otherwise it has no effect.

  jdk: `activation` has a built in, Java-centric check in the `jdk` element. The value is one of the following three types:
  - A version range according to the definition of maven-enforcer-plugin in case the value starts with either `[` or `(`,
  - A negated prefix if the value starts with `!` or
  - A (non-negated) prefix for all other cases

  (Negated) prefix values match if the JDK version used for running Maven starts/doesn't start with the given prefix (excluding the potentially leading `!`). The value ranges match if the JDK version used for running Maven is between lower and upper bounds (either inclusive or exclusive).

  os: The `os` element can require some operating system specific properties having specific values. Each value may start with `!` which means the condition is fulfilled if the value following does *not* equal the actual value, otherwise the condition is fulfilled if the value equals the according system property (or derived value).
  - **name**, is matched against system property `os.name`
  - **family**, is matched against the family derived from the other **os.*** system properties
  - **arch**, is matched against system property `os.arch`
  - **version**, is matched against system property `os.version`. Since Maven 3.9.7 the value for `version` may be prefixed with `regex:`. In that case regular pattern matching is applied for the version matching.

  See the maven-enforcer-plugin's Require OS Rule for more details about OS values.

- **property**: The `profile` will activate if Maven detects a system property or CLI user property (a value which can be dereferenced within the POM by `${name}`) of the corresponding `name=value` pair and it matches the given value (if given). Since Maven 3.9.0 one can also evaluate the **packaging value** of the pom via property name `packaging`.
- **file**: Finally, a given filename may activate the `profile` by the `existence` of a file, or if it is `missing`. **NOTE**: interpolation for this element is limited to `${basedir}`, System properties and request properties.

The POM based profile activation only refers to the container profile (not all profiles with the same `id`).

The `activation` element is not the only way that a `profile` may be activated. The `settings.xml` file's `activeProfile` element may contain the profile's `id`. They may also be activated explicitly through the command line via a comma separated list after the `-P` flag (e.g. `-P codecoverage`).

To see which profile will activate in a certain build, use the `maven-help-plugin`.

```
mvn help:active-profiles
```

Further information about profiles is available in Introduction to Build Profiles.

The BaseBuild Element Set *(revisited)*

As mentioned above, the reason for the two types of build elements reside in the fact that it does not make sense for a profile to configure build directories or extensions as it does in the top level of the POM. Reguardless of in which environment the project is built, some values will remain constant, such as the directory structure of the source code. *If you find your project needing to keep two sets of code for different environments, it may be prudent to investigate refactoring the project into two or more separate projects.*

Final

The Maven POM is big. However, its size is also a testament to its versatility. The ability to abstract all of the 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 Maven's other stars that make up the Maven galaxy - a well defined build lifecycle, easy to write and maintain plugins, centralized repositories, system-wide and user-based configurations, as well as the increasing number of tools to make developers' jobs easier to maintain complex projects - the POM is the large, but bright, center.

Aspects of this guide were originally published in the Maven 2 Pom Demystified.