Introduction to the Dependency Mechanism

Dependency management is a core feature of Maven. Managing dependencies for a single project is easy. Managing dependencies for multi-module projects and applications that consist of hundreds of modules is possible. Maven helps a great deal in defining, creating, and maintaining reproducible builds with well-defined classpaths and library versions.

Learn more about:

- Transitive Dependencies
  - Excluded/Optional Dependencies
- Dependency Scope
- Dependency Management
  - Importing Dependencies
- System Dependencies

Transitive Dependencies

Maven avoids the need to discover and specify the libraries that your own dependencies require by including transitive dependencies automatically.

This feature is facilitated by reading the project files of your dependencies from the remote repositories specified. In general, all dependencies of those projects are used in your project, as are any that the project inherits from its parents, or from its dependencies, and so on.

There is no limit to the number of levels that dependencies can be gathered from. A problem arises only if a cyclic dependency is discovered.

With transitive dependencies, the graph of included libraries can quickly grow quite large. For this reason, there are additional features that limit which dependencies are included:

- *Dependency mediation* - this determines what version of an artifact will be chosen when multiple versions are encountered as dependencies. Maven picks the "nearest definition". That is, it uses the version of the closest dependency to your project in the tree of dependencies. You can always guarantee a version by declaring it explicitly in your project's POM. Note that if two dependency versions are at the same depth in the dependency tree, the first declaration wins.

    - "nearest definition" means that the version used will be the closest one to your project in the tree of dependencies. For example, if dependencies for A, B, and C are defined as A -> B -> C -> D 2.0 and A -> E -> D 1.0, then D 1.0 will be used when building A because the path from A to D through E is shorter. You could explicitly add a dependency to D 2.0 in A to force the use of D 2.0.
- *Dependency management* - this allows project authors to directly specify the versions of artifacts to be used when they are encountered in transitive dependencies or in dependencies where no version has been specified. In the example in the preceding section a dependency was directly added to A even though it is not directly used by A. Instead, A can include D as a dependency in its dependencyManagement section and directly control which version of D is used when, or if, it is ever referenced.
- *Dependency scope* - this allows you to only include dependencies appropriate for the current stage of the build. This is described in more detail below.
- *Excluded dependencies* - If project X depends on project Y, and project Y depends on project Z, the owner of project X can explicitly exclude project Z as a dependency, using the "exclusion" element.
- *Optional dependencies* - If project Y depends on project Z, the owner of project Y can mark project Z as an optional dependency, using the "optional" element. When project X depends on project Y, X will depend only on Y and not on Y's optional dependency Z. The owner of project X may then explicitly add a dependency on Z, at her option. (It may be helpful to think of optional dependencies as "excluded by default.")

Dependency Scope

Dependency scope is used to limit the transitivity of a dependency, and also to affect the classpath used for various build tasks.

There are 6 scopes available:

- **compile**
  This is the default scope, used if none is specified. Compile dependencies are available in all classpaths of a project. 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 the dependency at runtime. For example, when building a web application for the Java Enterprise Edition, you would set the dependency on the Servlet API and related Java EE APIs to scope `provided` because the web container provides those classes. This scope 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. This scope 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.
- **import**
  This scope is only supported on a dependency of type `pom` in the `<dependencyManagement>` section. It indicates the dependency to be replaced with the effective list of dependencies in the specified POM's `<dependencyManagement>` section. Since they are replaced, dependencies with a scope of `import` do not actually participate in limiting the transitivity of a dependency.

Each of the scopes (except for `import`) affects transitive dependencies in different ways, as is demonstrated in the table below. If a dependency is set to the scope in the left column, transitive dependencies of that dependency with the scope across the top row will result in a dependency in the main project with the scope listed at the intersection. If no scope is listed, it means the dependency will be omitted.

	compile	provided	runtime	test
compile	compile(*)	-	runtime	-
provided	provided	-	provided	-
runtime	runtime	-	runtime	-
test	test	-	test	-

(*) Note: it is intended that this should be runtime scope instead, so that all compile dependencies must be explicitly listed - however, there is the case where the library you depend on extends a class from another library, forcing you to have available at compile time. For this reason, compile time dependencies remain as compile scope even when they are transitive.

Dependency Management

The dependency management section is a mechanism for centralizing dependency information. When you have a set of projects that inherits a common parent it's possible to put all information about the dependency in the common POM and have simpler references to the artifacts in the child POMs. The mechanism is best illustrated through some examples. Given these two POMs which extend the same parent:

Project A:

```xml
<project>
 ...
 <dependencies>
  <dependency>
    <groupId>group-a</groupId>
    <artifactId>artifact-a</artifactId>
    <version>1.0</version>
    <exclusions>
      <exclusion>
        <groupId>group-c</groupId>
        <artifactId>excluded-artifact</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
  <dependency>
    <groupId>group-a</groupId>
    <artifactId>artifact-b</artifactId>
    <version>1.0</version>
    <type>bar</type>
    <scope>runtime</scope>
  </dependency>
 </dependencies>
</project>
```

Project B:

```xml
<project>
 ...
 <dependencies>
  <dependency>
    <groupId>group-c</groupId>
    <artifactId>artifact-b</artifactId>
    <version>1.0</version>
    <type>war</type>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>group-a</groupId>
    <artifactId>artifact-b</artifactId>
    <version>1.0</version>
    <type>bar</type>
    <scope>runtime</scope>
  </dependency>
 </dependencies>
</project>
```

These two example POMs share a common dependency and each has one non-trivial dependency. This information can be put in the parent POM like this:

```xml
<project>
 ...
 <dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>group-a</groupId>
      <artifactId>artifact-a</artifactId>
      <version>1.0</version>

      <exclusions>
        <exclusion>
          <groupId>group-c</groupId>
          <artifactId>excluded-artifact</artifactId>
        </exclusion>
      </exclusions>

    </dependency>

    <dependency>
```

```
      <groupId>group-c</groupId>
      <artifactId>artifact-b</artifactId>
      <version>1.0</version>
      <type>war</type>
      <scope>runtime</scope>
    </dependency>

    <dependency>
      <groupId>group-a</groupId>
      <artifactId>artifact-b</artifactId>
      <version>1.0</version>
      <type>bar</type>
      <scope>runtime</scope>
    </dependency>
  </dependencies>
 </dependencyManagement>
</project>
```

And then the two child poms would become much simpler:

```
<project>
 ...
 <dependencies>
  <dependency>
   <groupId>group-a</groupId>
   <artifactId>artifact-a</artifactId>
  </dependency>

  <dependency>
   <groupId>group-a</groupId>
   <artifactId>artifact-b</artifactId>
   <!-- This is not a jar dependency, so we must specify type. -->
   <type>bar</type>
  </dependency>
 </dependencies>
</project>
```

```
<project>
 ...
 <dependencies>
  <dependency>
   <groupId>group-c</groupId>
   <artifactId>artifact-b</artifactId>
   <!-- This is not a jar dependency, so we must specify type. -->
   <type>war</type>
  </dependency>

  <dependency>
   <groupId>group-a</groupId>
   <artifactId>artifact-b</artifactId>
   <!-- This is not a jar dependency, so we must specify type. -->
   <type>bar</type>
  </dependency>
 </dependencies>
</project>
```

NOTE: In two of these dependency references, we had to specify the <type/> element. This is because the minimal set of information for matching a dependency reference against a dependencyManagement section is actually **{groupId, artifactId, type, classifier}**. In many cases, these dependencies will refer to jar artifacts with no classifier. This allows us to shorthand the identity set to **{groupId, artifactId}**, since the default for the type field is `jar`, and the default classifier is null.

A second, and very important use of the dependency management section is to control the versions of artifacts used in transitive dependencies. As an example consider these projects:

Project A:

```
<project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>maven</groupId>
 <artifactId>A</artifactId>
 <packaging>pom</packaging>
 <name>A</name>
 <version>1.0</version>
 <dependencyManagement>
   <dependencies>
     <dependency>
       <groupId>test</groupId>
       <artifactId>a</artifactId>
       <version>1.2</version>
     </dependency>
     <dependency>
       <groupId>test</groupId>
       <artifactId>b</artifactId>
       <version>1.0</version>
       <scope>compile</scope>
     </dependency>
     <dependency>
       <groupId>test</groupId>
       <artifactId>c</artifactId>
       <version>1.0</version>
       <scope>compile</scope>
     </dependency>
     <dependency>
       <groupId>test</groupId>
       <artifactId>d</artifactId>
       <version>1.2</version>
     </dependency>
   </dependencies>
 </dependencyManagement>
</project>
```

Project B:

```
<project>
  <parent>
    <artifactId>A</artifactId>
    <groupId>maven</groupId>
    <version>1.0</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <groupId>maven</groupId>
  <artifactId>B</artifactId>
  <packaging>pom</packaging>
  <name>B</name>
  <version>1.0</version>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>test</groupId>
        <artifactId>d</artifactId>
        <version>1.0</version>
      </dependency>
    </dependencies>
  </dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>test</groupId>
      <artifactId>a</artifactId>
      <version>1.0</version>
      <scope>runtime</scope>
    </dependency>
    <dependency>
      <groupId>test</groupId>
      <artifactId>c</artifactId>
      <scope>runtime</scope>
    </dependency>
  </dependencies>
```

```
</project>
```

When maven is run on project B version 1.0 of artifacts a, b, c, and d will be used regardless of the version specified in their pom.

- a and c both are declared as dependencies of the project so version 1.0 is used due to dependency mediation. Both will also have runtime scope since it is directly specified.
- b is defined in B's parent's dependency management section and since dependency management takes precedence over dependency mediation for transitive dependencies, version 1.0 will be selected should it be referenced in a or c's pom. b will also have compile scope.
- Finally, since d is specified in B's dependency management section, should d be a dependency (or transitive dependency) of a or c, version 1.0 will be chosen - again because dependency management takes precedence over dependency mediation and also because the current pom's declaration takes precedence over its parent's declaration.

The reference information about the dependency management tags is available from the project descriptor reference.

Importing Dependencies

The examples in the previous section describe how to specify managed dependencies through inheritence. However, in larger projects it may be impossible to accomplish this since a project can only inherit from a single parent. To accommodate this, projects can import managed dependencies from other projects. This is accomplished by declaring a pom artifact as a dependency with a scope of "import".

Project B:

```
<project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>maven</groupId>
 <artifactId>B</artifactId>
 <packaging>pom</packaging>
 <name>B</name>
 <version>1.0</version>
 <dependencyManagement>
   <dependencies>
     <dependency>
       <groupId>maven</groupId>
       <artifactId>A</artifactId>
       <version>1.0</version>
       <type>pom</type>
       <scope>import</scope>
     </dependency>
     <dependency>
       <groupId>test</groupId>
       <artifactId>d</artifactId>
       <version>1.0</version>
     </dependency>
   </dependencies>
 </dependencyManagement>
 <dependencies>
   <dependency>
     <groupId>test</groupId>
     <artifactId>a</artifactId>
     <version>1.0</version>
     <scope>runtime</scope>
   </dependency>
   <dependency>
     <groupId>test</groupId>
     <artifactId>c</artifactId>
     <scope>runtime</scope>
   </dependency>
 </dependencies>
</project>
```

Assuming A is the pom defined in the preceding example, the end result would be the same. All of A's managed dependencies would be incorporated into B except for d since it is defined in this pom.

Project X:

```
<project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>maven</groupId>
 <artifactId>X</artifactId>
 <packaging>pom</packaging>
 <name>X</name>
 <version>1.0</version>
 <dependencyManagement>
   <dependencies>
     <dependency>
       <groupId>test</groupId>
       <artifactId>a</artifactId>
       <version>1.1</version>
     </dependency>
     <dependency>
       <groupId>test</groupId>
       <artifactId>b</artifactId>
       <version>1.0</version>
       <scope>compile</scope>
     </dependency>
   </dependencies>
 </dependencyManagement>
</project>
```

Project Y:

```
<project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>maven</groupId>
 <artifactId>Y</artifactId>
 <packaging>pom</packaging>
 <name>Y</name>
 <version>1.0</version>
 <dependencyManagement>
   <dependencies>
     <dependency>
       <groupId>test</groupId>
       <artifactId>a</artifactId>
       <version>1.2</version>
     </dependency>
     <dependency>
       <groupId>test</groupId>
       <artifactId>c</artifactId>
       <version>1.0</version>
       <scope>compile</scope>
     </dependency>
   </dependencies>
 </dependencyManagement>
</project>
```

Project Z:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>maven</groupId>
  <artifactId>Z</artifactId>
  <packaging>pom</packaging>
  <name>Z</name>
  <version>1.0</version>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>maven</groupId>
        <artifactId>X</artifactId>
        <version>1.0</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
      <dependency>
        <groupId>maven</groupId>
        <artifactId>Y</artifactId>
        <version>1.0</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
</project>
```

In the example above Z imports the managed dependencies from both X and Y. However, both X and Y contain dependency a. Here, version 1.1 of a would be used since X is declared first and a is not declared in Z's dependencyManagement.

This process is recursive. For example, if X imports another pom, Q, when Z is processed it will simply appear that all of Q's managed dependencies are defined in X.

Imports are most effective when used for defining a "library" of related artifacts that are generally part of a multiproject build. It is fairly common for one project to use one or more artifacts from these libraries. However, it has sometimes been difficult to keep the versions in the project using the artifacts in synch with the versions distributed in the library. The pattern below illustrates how a "bill of materials" (BOM) can be created for use by other projects.

The root of the project is the BOM pom. It defines the versions of all the artifacts that will be created in the library. Other projects that wish to use the library should import this pom into the dependencyManagement section of their 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 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.test</groupId>
  <artifactId>bom</artifactId>
  <version>1.0.0</version>
  <packaging>pom</packaging>
  <properties>
    <project1Version>1.0.0</project1Version>
    <project2Version>1.0.0</project2Version>
  </properties>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>com.test</groupId>
        <artifactId>project1</artifactId>
        <version>${project1Version}</version>
      </dependency>
      <dependency>
        <groupId>com.test</groupId>
        <artifactId>project2</artifactId>
        <version>${project1Version}</version>
      </dependency>
    </dependencies>
  </dependencyManagement>
  <modules>
    <module>parent</module>
  </modules>
</project>
```

The parent subproject has the BOM pom as its parent. It is a normal multiproject 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 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>com.test</groupId>
    <version>1.0.0</version>
    <artifactId>bom</artifactId>
  </parent>

  <groupId>com.test</groupId>
  <artifactId>parent</artifactId>
  <version>1.0.0</version>
  <packaging>pom</packaging>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>log4j</groupId>
        <artifactId>log4j</artifactId>
```

```xml
      <version>1.2.12</version>
    </dependency>
    <dependency>
      <groupId>commons-logging</groupId>
      <artifactId>commons-logging</artifactId>
      <version>1.1.1</version>
    </dependency>
  </dependencies>
 </dependencyManagement>
 <modules>
  <module>project1</module>
  <module>project2</module>
 </modules>
</project>
```

Next are the actual project poms.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <parent>
  <groupId>com.test</groupId>
  <version>1.0.0</version>
  <artifactId>parent</artifactId>
 </parent>
 <groupId>com.test</groupId>
 <artifactId>project1</artifactId>
 <version>${project1Version}</version>
 <packaging>jar</packaging>

 <dependencies>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
  </dependency>
 </dependencies>
</project>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <parent>
  <groupId>com.test</groupId>
  <version>1.0.0</version>
  <artifactId>parent</artifactId>
 </parent>
 <groupId>com.test</groupId>
 <artifactId>project2</artifactId>
 <version>${project2Version}</version>
 <packaging>jar</packaging>

 <dependencies>
  <dependency>
    <groupId>commons-logging</groupId>
    <artifactId>commons-logging</artifactId>
  </dependency>
 </dependencies>
</project>
```

The project that follows shows how the library can now be used in another project without having to specify the dependent project's versions.

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <groupId>com.test</groupId>
 <artifactId>use</artifactId>
 <version>1.0.0</version>
 <packaging>jar</packaging>
```

```

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>com.test</groupId>
        <artifactId>bom</artifactId>
        <version>1.0.0</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.test</groupId>
      <artifactId>project1</artifactId>
    </dependency>
    <dependency>
      <groupId>com.test</groupId>
      <artifactId>project2</artifactId>
    </dependency>
  </dependencies>
</project>
```

Finally, when creating projects that import dependencies beware of the following:

- Do not attempt to import a pom that is defined in a submodule of the current pom. Attempting to do that will result in the build failing since it won't be able to locate the pom.
- Never declare the pom importing a pom as the parent (or grandparent, etc) of the target pom. There is no way to resolve the circularity and an exception will be thrown.
- When referring to artifacts whose poms have transitive dependencies the project will need to specify versions of those artifacts as managed dependencies. Not doing so will result in a build failure since the artifact may not have a version specified. (This should be considered a best practice in any case as it keeps the versions of artifacts from changing from one build to the next).

System Dependencies

Important note: This is marked deprecated.

Dependencies with the scope *system* are always available and are not looked up in repository. They are usually used to tell Maven about dependencies which are provided by the JDK or the VM. Thus, system dependencies are especially useful for resolving dependencies on artifacts which are now provided by the JDK, but where available as separate downloads earlier. Typical example are the JDBC standard extensions or the Java Authentication and Authorization Service (JAAS).

A simple example would be:

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>javax.sql</groupId>
      <artifactId>jdbc-stdext</artifactId>
      <version>2.0</version>
      <scope>system</scope>
      <systemPath>${java.home}/lib/rt.jar</systemPath>
    </dependency>
  </dependencies>
  ...
</project>
```

If your artifact is provided by the JDK's `tools.jar`, the system path would be defined as follows:

```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>sun.jdk</groupId>
      <artifactId>tools</artifactId>
      <version>1.5.0</version>
      <scope>system</scope>
      <systemPath>${java.home}/../lib/tools.jar</systemPath>
    </dependency>
  </dependencies>
  ...
</project>
```