Notes For Maven 3.9.x Plugin Developers

Contents:

This page will take a stab on announcing breaking changes with latest upcoming Maven versions, and to discuss "best practices" for building Maven Plugins. This page assumes that reader know the general "best practices" for Java Projects (as Maven Plugins are basically Java projects).

Incoming Breaking Changes

Maven Project resources are limited we cannot cover "backward compatilibty" across two major versions. Hence, Maven 2 support is about to be removed. Below is the summary of breaking changes, that will happen in upcoming Maven 3.9.0 and 4.0.0:

- MNG-6965 The `org.codehaus.plexus:plexus-utils` artifact is not anymore "auto injected" (auto provided) to plugins classpath. Maven 2 did provide this dependency from Maven Core automatically to plugins and extensions, then Maven 3.0.0-alpha-3 added this feature to ease plugins transition MNG-3819. This is not the case anymore. Plugin developers have to prepare for this change. Backward compatible change is really simple: just declare dependency on `plexus-utils` in `compile` scope, if your plugin does use classes from it, but does not have it declared (or have it in `provided` scope).
- If you depend on `org.apache.maven:maven-compat` (the Maven2 compatibility layer), it's really time to look for alternatives. Note: this dependency in `test` scope is "acceptable" and actually required by some testing frameworks (see below). This **module is *not* removed in 3.9.0, nor in first releases of 4.0.x**, but as part of Maven 2 backward compatibility layer, is to be removed somewhere in future.

Incoming Notable Changes

Maven 4.0.0 brings some new things in the play:

- A brand new immutable Maven API. Finally Maven gets a proper API, that solves two things: developers of plugins can unlesh all the Maven power without jumping hoops and loops, and for us (Maven developers) even better: the Maven implementation finally gets hidden, we can freely hack away. The idea is that the new Maven API becomes the "only touch point" between Plugins and Maven, so Maven Core should not be hindered to evolve with preserving binary compatibility of it's internals (as they will be not accessible, sealed off). Maven API will be available starting with 4.0.0 release of Maven, and we will offer "transition time" for plugins as first Maven 4.0.x (undefined yet for how long) will support both, "old way written Maven Plugins" (as today) and Maven API. But, once you convert to Maven API, your plugin will have a prerequisite of Maven 4+.
- Default Maven Transport changes from legacy Wagon to more modern Resolver HTTP "native" transport. For plugin developers this should not be a huge change, as Wagon is still exposed via (deprecated Maven 3.x APIs), while new Maven 4 Transport API was introduced, that is intentionally simplistic, to cover most common (simple) cases. Still, if plugin needs something more, that cannot me achieved using this Transport API (nor using M4 Resolver API), it should roll it's own transport.

Minimal Set Of Best Practices

Maven Plugins are meant to be invoked by and run within Maven. Hence, one can draw a parallel between them and, for example, Java Servlets, where Servlet Container "provides" some dependencies to implementations. In this aspect: Maven is also a Container, container for Maven Plugins. Maven provides to plugins the "Maven API" classloader as parent, but to build a plugin, you still need to declare some depedencies.

A Maven Plugin aside of usually "Java project" things like Java source/target/release version needs to declare several extra things:

- mandatory: packaging is `maven-plugin`
- recommended: lock down the `maven-plugin-plugin` version (as otherwise you depend on version that Maven brings, your build is not reproducible).
- recommended: the "required" (minimal) Maven version to run within (this is usually same as Maven version you build against) in POM as `project/prerequisites/maven` field.
- recommended: the Maven provided bits should be put in `provided` scope. Recent `maven-plugin-plugin` versions will warn you about this.
- mandatory: declare needed dependencies, minimally required are org.apache.maven:`maven-plugin-api` and `org.apache.maven.plugin-tools:maven-plugin-annotations` (to be able to annotate your Mojos, all the older ways like Javadoc taglets are being deprecated).

We can already see, that we have at least two repeating versions, so they are "potential" properties to lessen duplication:

- the `mavenVersion` to declare Maven version we build against (and use it in prerequisites and dependency declarations)
- the `mavenPluginToolsVersion` to declare Maven Plugin Tools version, once as dependency version for `org.apache.maven.plugin-tools:maven-plugin-annotations` dependency and once for `org.apache.maven.plugins:maven-plugin-plugin` build plugin (and optionally for `org.apache.maven.plugins:maven-plugin-report-plugin`).

Hence, a "minimal" Maven Plugin POM (showing elements related to Maven Plugins only) should have elements like this:

```
<prerequisites>
  <maven>${mavenVersion}</maven>
</prerequisites>

<properties>
  <mavenVersion>3.6.3</mavenVersion>
  <mavenPluginToolsVersion>3.7.0</mavenPluginToolsVersion>
</properties>

<dependencies>
  <dependency>
    <groupId>org.apache.maven</groupId>
    <artifactId>maven-plugin-api</artifactId>
    <version>${mavenVersion}</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.apache.maven.plugin-tools</groupId>
    <artifactId>maven-plugin-annotations</artifactId>
    <version>${mavenPluginToolsVersion}</version>
    <scope>provided</scope>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-plugin-plugin</artifactId>
      <version>${mavenPluginToolsVersion}</version>
    </plugin>
  </plugins>
</build>
```

Notes:

- Do not try to support 10+ years of legacy. It is warmly recommended to just forget about ill fated Maven 3.0, so consider Maven versions from 3.1 and above. Still, even with Maven 3.1 you cover 10 years. Currently Apache Maven project defines "minimal baseline" (prerequisite) of 3.2.5, but only in case of "trivial" plugins like `maven-clean-plugin` is. Still, "latest stable" it too high constraint as well. Best middle ground is something like "latest patch release of one/two minor versions minus from current stable" or some similar policy (as of writing: 3.8.7 is latest stable, hence 3.6.3 is sane middle ground). Also, do not forget that Maven versions since 3.3 were all Java 7, just like current latest stable 3.8.7 is!
- This snippet shows **only entries recommended/required for Maven Plugins**! These are required for Maven Plugins, but as Maven Plugin is a Java Project, all the POM required elements, part of "best practices" for Java Project (parent POM, fixed plugin versions, etc) are not shown here!
- **Important: Maven 2.x was automatically "providing" `org.codehaus.plexus:plexus-utils` for plugins as well, or in other words, plugins did not had to to declare `plexus-utils` as dependency, it was "just there" (or they did, but in `provided` scope). Maven 3.x continued doing this as part of "backward compatibility" with Maven 2.x line, but this will not happen anymore with Maven 3.9 and 4.0! If your plugin uses `plexus-utils`, but does not have it declared as `compile` scoped dependency, it will fail in Maven 3.9.0 or later.**
- In case your plugin does a bit more than something trivial, you usually need to use more of Maven. In that case just add all the needed depenendencies in your POM as provided scoped. In general, it is true that all Maven artifacts having groupId org.apache.maven should be in provided scope. **Important: There is one unfortunate exception to this rule above: org.apache.maven:maven-archiver. This artifact is not part of Maven Core, and uses unfortunate groupId. Should be `org.apache.maven.shared` instead.**

Then you can enlist your other (non-Maven) dependencies as well.

Testing Maven Plugins

Testing Maven Plugins is not trivial thing, given they are Maven Components, but not plain Eclipse Sisu or Codehaus Plexus components, but rather the plugin descriptor needs to be "translated" and installed into (any of two) container, plus the existing components in Maven Plugin JAR, if any. Hence, simplest is to use some existing frameworks for testing Maven Plugins.

This section is more like a collection of links/pointers, so below is a list of some documentation and frameworks:

- the "old school" https://maven.apache.org/plugin-developers/plugin-testing.html
- nice related writeup/series https://dev.to/khmarbaise/series/8418 and https://khmarbaise.github.io/maven-it-extension/itf-documentation/background/background.html

Alternative frameworks supporting Junit5 or taking alternate approach:

- https://khmarbaise.github.io/maven-it-extension/
- https://github.com/takari/takari-plugin-testing-project