Maven 3.x Class Loading

This article outlines the class loader hierarchy employed by Maven 3.x and is intended to assist plugin authors in understanding what classes/resources are accessible to their plugins at runtime.

Bootstrap Class Loader

This class loader provides the classes that make up the Java Runtime Environment, e.g. the classes from the packages `java.*`. It is the ultimate parent of all other class loaders. Bootstrap classes cannot be overriden by plugin dependencies.

System Class Loader

When executing Maven from the command line, the system class path as reported by the system property `java.class.path` contains only a single JAR named `plexus-classworlds.jar`. This utility JAR provides a class loader framework that Maven employs to create its class loader hierarchy. When Maven is embedded into an IDE or CI server, the system class path will likely look differently, e.g. contain JARs required by the IDE. In general, the system class loader supports the current Java application and not necessarily the components that this application embeds.

For these reasons, plugin authors should not make any assumptions about the system class loader unless the plugin documentation instructs the user to take specific actions for setting up the system class path (e.g. by loading a Java agent). In particular, plugins must not try to programmatically hack the system class loader.

Maven Core Class Loader

The JARs contained in the directory `${maven.home}/lib` of a Maven installation constitute the class path used to run the Maven core itself. This core class loader hosts both the public API of Maven, i.e. classes like `MavenProject`, and internal utility classes.

The core class loader is usually chained together with other class loaders by means of a parent-child relationship with the core class loader being the parent. In this constellation, the child class loader can only access the public Maven API from the core classs loader, i.e. roughly only classes from the package `org.apache.maven` and its subpackages are available to the child class loader.

Extension Class Loader

Inside their `<build>` section, projects can declare `<extension>` elements or `<plugin>` elements with `<extensions>true</extensions>` to introduce components that should be used by the Maven core or shared with other plugins. When build extensions refer to the same group id, artifact id, version and dependencies, they will use the same extension class loader. Otherwise, each extension gets its own class loader.

When two or more modules in a reactor build refer to the same extension, they do not only share the same class loader for this extension but also the same pool of components contributed by the extension. In other words, if an extension provides a stateful component implemented as a singleton, all modules in the reactor using this extension will share this singleton and its state.

Extension class loaders are also wired to the Maven core class loader and hence can access the Maven API.

Project Class Loader

In order to aggregate the various build extensions of a project, a project can be associated with a project class loader. In case a project uses no build extensions and hence has no need for a dedicated project class loader, this class loader will technically not be created.

The project class loader is wired to each extension class loader required for the project and also the core class loader. Just like the core class loader imposes restrictions on the classes that it exports to the project class loader, authors of build extension can do the same to prevent leakage of their internal utility classes into other plugins. To declare the packages exported by an extension, an extension descriptor as shown below needs to be present at the path `META-INF/maven/extension.xml` in the main artifact of the build extension:

```
<?xml version="1.0" encoding="UTF-8"?>
<extension>
  <!-- the packages to share with build plugins -->
  <exportedPackages>
    <exportedPackage>org.something.myextension</exportedPackage>
  </exportedPackages>

  <!-- the group & artifact ids to exclude from the class realms of build plugins because the extension
provides them -->
  <exportedArtifacts>
    <exportedArtifact>org.company:myextension</exportedArtifact>
  </exportedArtifacts>
</extension>
```

Only packages (and their sub packages) given by `<exportedPackage>` will be imported into the project class loader and are eventually available to plugins of the project. All other packages will be kept private to the extension class loader.

The `<exportedArtifacts>` declaration defines which dependencies of the build extension actually contribute the exported packages. Those artifacts will be excluded from the plugin class loaders to prevent duplicate classes and components in the plugin. These exclusions are not transitive, i.e. if "myextension" is declared as an exported artifact, only this specific artifact will be excluded from the plugin class loader, the transitive dependencies of "myextension" will not be excluded.

The extension descriptor can be empty or even missing as is the case with most existing Maven plugins that provide extensions. In absence of information about the exported packages/artifacts, no packages/artifacts will be exported.

The project class loader must not be confused with the compile/runtime/test classpath of the project. This class loader supports the build plugins that run in the context of the project, it has no relation to the dependencies required to compile or run the project itself.

Plugin Class Loader

For each plugin used by a project, a plugin class loader is created as a child of the project class loader. If the project has no extensions and as such no project class loader, the plugin class loader will be a direct child of the Maven core class loader instead.

The class path of a plugin class loader is derived from the dependencies of the plugin as defined in the POM of the plugin artifact and optionally by user-defined plugin dependencies listed in the `<plugin>` declaration of the project that uses the plugin. The plugin class loader does neither contain the dependencies of the current project nor the compiled main/test classes of the project. Plugins that need to load classes from the compile/runtime/test class path of a project need to create a custom `URLClassLoader` in combination with the mojo annotation `@requiresDependencyResolution`.

To sum up, a plugin can access classes/resources from the Java bootstrap class path, from the dependencies of the plugin, from the Maven API and from the system class loader. To load other classes, a custom class loader needs to be created.