Conflict Resolution

Improving Artifact Conflict Resolution

jdcasey
NOTE: This page describes the best of my knowledge on this topic. To be absolutely certain about my facts (in the Context, etc), I'd need to verify by looking at the source code for maven-artifact and maven-artifact-manager.

Context

Currently, Maven supports the resolution of artifact versions by way of nearest-wins. That is, for any set of dependencies that share the same groupId:artifactId:type[classifier] in a particular artifact closure, the one declared nearest to the current project in the dependency tree will be selected for use. This can lead to subtle effects, where upgrading a particular version of one dependency in your POM can lead to a transitive dependency being declared nearer than previously, and therefore changing the version used for that transitive dependency.

While this approach to conflict resolution is probably at least as good as any at solving the problem generally, it's important to recognize that certain situations may call for very different conflict resolution methods.

The following scenarios describe times when specific conflict resolution techniques may be used to address specific problems.

Testing and Production Builds

In this scenario, users will likely want to know if there is any disagreement on dependency versions. Any disagreement could be used to spot-check the full application for features that may express incompatibility with the chosen version of the artifact.

In some cases, these builds might be allowed to proceed with simple warnings about version conflicts. In others (particularly when the application is thought to be ready for a production rollout), it would be highly desirable to have the build fail if a conflict is detected, since any conflicts should have been resolved manually in the POM during testing or earlier.

Alternatively, it's possible that a testing team wouldn't want to adjust upstream dependency declarations, and wouldn't want to impact the purity of the application POM's dependency specification. In these cases, they want the dependency list to converge on a set of "blessed" versions without allowing the build system any leeway to decide (with a potential to decide incorrectly) on its own.

Nightly / Integration Builds

Nightly and integration-style builds will likely take place on a CI server, and should sometimes incorporate a build of all the latest artifacts which match the specified dependency list. This way it's possible to report on possible migration/upgrade paths for POMs in the build that are not using the latest matching version.

Corporate Policy-compliance Builds

Many companies keep tight control over which external dependencies they allow for the software they produce. This can take many forms, from auditing and approving individual artifacts as they are added to a corporate repository, to

auditing builds during testing, etc. In these situations, it might be useful to impose a conflict resolution technique to enforce the usage of a blessed set of artifact versions in a given build. This would allow other groups in the company to vet each artifact version before putting it on the blessed list.

In this case, non-compliance would likely result in a broken build.

Out-and-out Version Overrides

In some special cases, it may be useful to override the version resolution process altogether, and instead use some alternate artifact version. One such scenario might involve generating migration reports for each of a set of projects. These reports might let developers know what sorts of problems they'd be likely to confront if they tried to move their application from its existing dependency-version set to the latest-and-greatest set. To make these more useful, it might be good to restrict the version override to a single dependency at a time, to isolate the errors related to that dependency upgrade.

Problem

Maven's artifact-handling components have the current version conflict resolution strategy hard-coded in place. The current strategy has to be componentized and abstracted to an interface so that we can switch strategies.

Beyond this, we need to provide a mechanism for configuring the desired conflict resolution technique. For most purposes, it might be best to configure a Maven *instance* to use a particular technique, assuming that testing users, pre-production users, and CI systems will always look for the same type of conflict resolution technique.