Artifact Resolution

Improving Artifact Resolution

jdcasey
NOTE: I'm not sure I've accurately captured some of the arguments in this discussion. If you know more about it, please add your remarks and/or correct mine!

Context

1. Artifact Transitivity
2. Dependency Exclusions
3. DependencyManagement
4. Artifact Scopes
5. Attachments and Transitive Resolution

1. Artifact Transitivity

In some cases - probably due to bad metadata on the repository - the dependency exclusion problem becomes so acute as to make it more productive to simply disable transitive artifact resolution. Unless/until we can come up with a strategy to clean up the metadata on the repository such that users will have access to it immediately, and maintain a higher standard of quality for this metadata (perhaps incorporating some sort of user comments/ratings in MRM per-artifact?), we will have some users who feel it would be much easier to build their projects without transitive artifact resolution.

2. Dependency Exclusions

Currently, dependency exclusions are specified as affecting a single dependency's transitive closure. However, because of a bug in Maven, any exclusion will affect **all** of the project's dependencies - the whole artifact resolution process for that build, in other words.

There are some who feel that it would be simpler to change the exclusion semantics (and syntax), such that excluding an artifact formally specifies that you're excluding it globally from that build. This will effectively formalize the process that's already in place (as I understand it), and would lead to dependency exclusions being specified directly within a dependencyManagement section, for instance, rather than within a single dependency declaration.

3. DependencyManagement

With the proposed changes to version conflict resolution, restricting the DependencyManagement section to only those dependencies specified directly in the POM inheritance hierarchy of the current project poses a subtle problem of consistency. That is, when the artifact version selected is not necessarily the nearest, it creates a situation where the DependencyManagement section clearly states one version/scope/etc. but another one is used because the dependency used is not the one that was declared directly within the POM.

> **jdcasey**
>
> I know there are other reasons to use DependencyManagement outside of the POM inheritance hierarchy (read: transitively), but I can't think of a single one. The concept that an external (to your own POM hierarchy) entity would be able to supply a version/scope/etc. for the dependency you've declared seems rather ridiculous to me, to be honest.
>
> Please, someone, shed some light on this for me!

4. Artifact Scopes

> **jdcasey**
>
> From my perspective, it's not clear whether the existing scoping mechanism is really fulfilling its purpose. In other words, I'm not clear on whether we need to "clean up" the existing rules for artifact scopes, or abandon scoping altogether in favor of some more flexible option.

There is a lot of confusion in the community (and even among Maven developers!) about how the different scopes imply one another, and what use case some of the more exotic scopes are meant to address. Compounding this, there are several pro/con arguments about whether certain scopes should be transitive. Also, it's become relatively clear that the system scope represents a design flaw in Maven 2.0. Finally, the scope vocabulary supported by Maven is sometimes not adequate to capture the different classes of dependencies used in the build process, and definitely cannot accommodate new dependency usages which might be encountered when using new build lifecycle mappings.

Provided Scope Debate

Some feel that the provided scope should be transitive. The argument is that the top-level consumer should have control over what artifacts are excluded as provided (or for any reason, for that matter). Others would say that the provided scope annotates an artifact as always meant to be provided by the platform, a fact that doesn't change when it's discovered transitively.

System Scope Outlier

The system scope almost seems to be a way of addressing a special case where provided scope + a specification-style dependency might be a better solution. The reason is that system-scoped dependencies are assumed to pre-exist on the build machine, configured and maintained externally WRT the build process. Things like the Javac compiler exist in different locations on different JDK implementations (or not at all on some implementations), and therefore cannot be adequately captured as a dependency by something as concrete as a systemPath (the operative part of a system-scoped dependency).

I think most Maven developers agree that there must be some better solution to this problem, which will accommodate situations like the above more elegantly.

Scope Vocabulary vs. Extensibility

The static scoping vocabulary supported by Maven 2.0 cannot address all of the various distinct use cases for dependencies in existing builds. For example, the integration-test lifecycle phase has no accompanying dependency scope, which means it must share some other scope with another phase - which may lead to classpath problems in

one or more phases of the build. Since Maven 2.0 supports custom lifecycle mappings via the <packaging/> element, this problem could literally explode in front of us. To extend the previous example, this would become a major concern if functional, system, or performance testing phases were added.

jdcasey
My feeling here is that there needs to be a more extensible mechanism for specifying dependency scopes and their heritability. Alternative lifecycles (or an evolving lifecycle with more sophisticated testing support) should have the ability to simply define a new scoping type, along with it the rules for what that scope inherits, or what scopes inherit from it.

Attachments and Transitive Resolution

We also need a way to distinguish the dependencies that apply when resolved transitively via an attached artifact, rather than the main one. For example, while the server-side component of an EJB may need several dependencies, the ejb-client aspect might only need one or two of those, just enough to fulfill the needs of the EJB interfaces, and give the client the ability to work with the data. However, when a dependency of type ejb-client is declared in a project, the entire dependency set is resolved transitively.

Problem

Dependency Scoping rules are hard-coded in Maven 2.0, which means we will have to design and implement a componentized solution to accommodate new scopes. Additionally, there is currently no way to specify a transitivity flag or list of project-global exclusions list, which means these solutions will also need to be designed and implemented.

Taken with the proposed changes to DependencyManagement, these new behaviors will almost certainly cause backward-compatibility problems, which is not an option. Therefore, the real problem becomes how to support these sorts of enhancements while still supporting legacy (2.0) builds.

Resources