Transitive Dependencies Filtering

Issues

Users have expressed the desire to turn off transitive dependencies and/or filter out some dependencies obtained transitively. This design attempts to allow the latter.

There is one contentious issue: should these filters be transitive?

Arguments in favour:

- less poms affected (only needs to specified to 1 level, not N)
- no need to know anything about the dependencies of your dependencies
- can be specified at a per dependency level, so less potential to be harmful (eg, you block jdbc knowing you don't need it from p-v, but some other dep introduces it too and really does need it)
- more of the information is likely to be in the repository, so we can report on it (more mid poms than end poms in the repo)
- doesn't make the assumption that an optional dependency is wrong
- its more like what users want, which means less re-education

Arguments against:

- potential to be harmful (p-v blocks jdbc, but down the foodchain someone uses velocity via p-v and triggers jdbc code)

Use cases

velocity's inclusion of jdbc

velocity requires jdbc for it's jdbc resource loader. If you are not using the resource loader, you don't need jdbc.

plexus-velocity depends on velocity - should it expose the jdbc resource loader (and hence jdbc dep) to dependees, or should it hide that, and make dependees introduce a velocity dep if they use the jdbc resource loader?

jpox inclusion of oracle's jdbc driver

jpox requires this for clob support. modello generates a jpox store but does not use clobs, so that code will never be touched. Should all modello dependees filter out ojdbc, or should modello-plugins-jpox filter it out?

betwixt's inclusion of beanutils

betwixt requires digester to do its parsing. In some instances, it also requires beanutils. However, I recently changed some code in the m1 project loading and realised the portion of betwixt I am using did not need beanutils at all. If mevenide was to depend on the m1 project loading, it would have no knowledge of the use of betwixt - should it receive the beanutils dep?

Design where filters are transitive

```
<dependency>
  <groupId>...</groupId>
  ...
  <exclusions>
    <exclusion>
      <groupId>jdbc</groupId>
      <artifactId>jdbc</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Reasoning:

- makes a nice artifact filter for use when transitively resolving that dependency
- do it per dependency so it is clear where it is coming from and doesn't accidentally filter out elsewhere
- works for both Maven and Ant tasks
- Can easily produce a report from the repository showing how it is used/abused
- normally transitive because if you don't use it, then those depending on you should declare it explictly if they need it, as they're dependency on you only implies they need what you need + what they say they need
- use it in dependencies to make it transitive. If you want a non-transitive version, it is also available in dependencyMgmt

Design where filters are not transitive

```
<dependencyManagement>
  <exclusions>
    <exclusion>
      <groupId>jdbc</groupId>
      <artifactId>jdbc</artifactId>
    </exclusion>
  </exclusions>
</dependencyManagement>
```

Viewpoints

NOTE: This section was created in parallel to the above modifications to the original page. Therefore, its structure may be at odds with the above content, and some of the arguments may be a bit redundant. However, I'm leaving it as-is for now, since it expresses my own opinion on this issue inside a framework that I think will help others know where/how to contribute theirs.

I'm creating this section simply as an area for each of us to weigh in on this issue, with the hopes that:

-we can avoid another multi-hour design session without resolution
-we can track this discussion over time, for the sake of documenting the reasons for our eventual design decision

The main point of contention here (I think) is the behavior and location of dependency-exclusion specifications. Specifically, can intermediary projects specify dependency exclusions that will be effective for dependents of these projects, or must each project decide for itself which declared upstream dependencies it can safely live without?

John Casey

My tendency was to think about these alternatives as the effects of two goals of the repository which were at odds: enabling user productivity vs. correctly specifying project information. From this perspective, it is correct from a principles argument to say that subsequent dependents on a project shouldn't have the right to second-guess that project's release manager regarding the project's dependencies. The viewpoint is in favor of making a project's pom.xml the sole authority for that project, and assumes that all dependencies are required for all functionality in the project.

The other view rendered by this perspective is that release managers who poorly specify their requirements compromise the net utility of maven as a whole. It only takes some critical number of poorly specified projects to push users away from maven and toward competitor products. This view is in favor of limiting the effects of "bad" poms in an effort to make users more productive (avoiding trial-and-error discovery of useless or blocker transitive dependencies) and to reduce the noise generated in the repository (as multitudes of poms are deployed containing the same exclusion). This is the practical side of the debate.

This perspective is a polarizing one, since it pits the goal of usability against the goal of correctness and (wince) purity. However, I believe there is an important third argument to be made, requiring a slight perspective shift. I believe that these two goals are not diametrically opposed. I'll explain, but first I want to lay some foundations.

Definitions

First, what do we mean by usability? Since this is a build tool, and in transitive dependency inclusion we're trying to read the implications of a dependency set and "know what the user means" it is important that we strive to provide the most accurate interpretation of the user's wishes. By and large, the user should be shielded from making elaborate and exceptional modifications to his POM; it should be literally only a couple of lines long (amplified by a couple of lines per dependency, obviously). In keeping with this, we have different scopes to allow users to specify, for example, that JUnit doesn't belong in his end assembly. This alleviates what was an uncontrollable problem with the first implementation of transitive dependencies: severe bloat. Bloat is as important a problem to the end user as dependency-closure specification once was.

Second, what do we mean by correctness of the repository? It means that the POM for a project is a complete, comprehensive expression of what that project provides, in what form, and what assumptions it makes in providing it. In other words, it is supposed to say exactly what it does, and exactly what it needs to do it. The build section is an exception here, in that it is essentially specific build instructions about how the project's

artifact has been provided. From a correctness standpoint, it **seems** logical to assume that only the development team and end users are qualified to make changes to this information. The development team, because they developed it, and should know better than anyone else; and the end user, because even if they are wrong, their modifications will not affect others (signified by the qualifier "end"). If the POM is an authoritative provider of information, from where could others derive the authority to override it?

Example

To address this third viewpoint, I need an example. I chose jpox, for reasons that will become apparent. The jpox-1.4 POM specifies a dependency on Oracle's ojdbc jar, version 14. For clarity, this dependency can only be useful to those users who are using jpox functionality on Oracle, and probably serves to allow jpox to use Oracle's blob implementations and such.

For the sake of argument, say I have a db-oriented project that uses jpox, but only for part of its functionality. The other part consists of code that requires more performance. For this, I use jdbc core features like PreparedStatement and such. These are bundled with the JDK, so there is no issue related to that functionality in and of itself. However, one implication of this "other half" is that I have to construct SQL statements using non-jpox code...and damn it, Oracle's just too hard!  So, I don't support it. It should be noted that this is **not** an application, merely a library. Since the project's functionality is cohesive, there is no good reason for me to split the jpox-based parts from the jdbc-based parts.

The Third View

This is an interesting case, since it shows how the two previous viewpoints can be made compatible. From a usability standpoint, including an implicit dependency on ojdbc is misleading and a nontrivial nuisance for my users. If they want to avoid bloat (ojdbc 14 is 1.4 megs - and I thought the '14' was a version  they'll have to add their own exclusions or else trim out the oracle stuff by hand. From a correctness standpoint, my project doesn't even support oracle, so it's absolutely incorrect for me to export a transitive dependency on an oracle library (which may make it into their assemblies and deployments). There is no possible way that they could use ojdbc.jar without causing problems with my library. Therefore, it should be possible for me to specify this anti-requirement in my POM to keep them from thinking they can use my project with Oracle based on the presence of this dependency. Even though jpox has functionality that depends on ojdbc.jar, I do not. Therefore, I need to have the ability to specify, on behalf of my users, that we will **not** depend on ojdbc.jar. In this case, absence of such an ability is in clear violation of the correctness goal.

While it should be noted that I wholeheartedly agree with the critical mass argument - namely, that we are in a delicate position where we might start driving away users - I support propagated exclusions primarily to enable the case above. I believe this illustrates that not all propagated exclusion is an abusive exploit of another project's position of power between another project and the end user. Since there is a valid use case - and I would argue, not an altogether exotic use case at that - I believe this is something we should allow. All of our actions in making maven are subject to misuse, but just because I **could** specify a dependency on jdk-1.0.1 doesn't mean that we should disallow users from specifying dependencies.