

Artifact resolution and repository discovery

General description of current (2.x) resolution

(this is a super-simplified description that focuses on background relevant to the specific problem being analyzed)

Today Maven roughly follows the current recursive process to resolve dependencies:

Read and interpolate the current pom (or current dependency). Then pull out the list of dependencies and repositories from the pom. The repositories found are added to the existing list which comes from the superpom (central) and any in the settings.xml and any pulled from previous poms during this maven execution. These repositories are then used to resolve the next level of dependencies. Repeat.

If the artifact being resolved is a plugin, then only "pluginRepositories" are considered for resolution. Repositories and Plugin Repositories are filtered based on the declared snapshot/release flags and if the current artifact being located is a snapshot or release.

Benefits of the current approach

1. If dependencies of your build aren't contained in central, you can add a repository entry to your pom and Maven will find them. The practical benefit of this is two-fold
 - a. that someone with a normal maven install can checkout and build your code without knowing where your snapshots are, or where any of your dependencies might be located ahead of time. It just works.
 - b. that someone depending on your artifacts doesn't have to know where your downstream dependencies are. (Note: this may be a bad thing in some cases because of the repository pollution and non-deterministic resolution based on repos declared elsewhere)

Problems with current approach

1. Having repositories in the poms is usually bad. Released poms are immutable, but urls change over time. This eventually leads to poms that point to incorrect urls and dependencies can't be located automatically.
2. The repositories introduced by dependencies and transitive dependencies "pollute" the build. That is because once they get added to the list of repos, it stays their for the remainder of the build. Maven doesn't know which dependencies the pom author intended to introduce via the repository declaration. A very concrete problem is that there is no guarantee that a given artifact present in two different repositories are not the same. People take Apache artifacts and modify them so that if a different repository was used in a subsequent calculation you might get something different. Even with OSGi where a bundle can theoretically be sourced from any repository the behavior of that bundle is not guaranteed to be the same. So people want to know the set of artifacts they retrieved from a given set of repositories is more or less immutable. So here again you need a central authority to provide signatures if you want to absolutely guarantee this characteristic. We know from practice people change things all the time and it has dire effects on users.
3. The current implementation of pluginRepositories is troublesome and doesn't do what it intended. Currently the dependencies of a plugin are resolved via the regular repositories (since they themselves aren't a plugin). This has the practical effect that for every pluginRepo you introduce you must also introduce it as a repo. The valid use case that this attempted to solve is being able to separate the dependencies needed by the build from those needed for the build. You will often want different policies on things that are used by Maven plugins than you would for things allowed into your build. (think GPL artifacts)
4. The nature of walking the tree and discovering repositories as you go makes it difficult to do a bounded SAT range calculation. Each decision may introduce yet more repositories that may have affected the previous calculations. It's not that this is not solvable it just takes too much time to be practical by a system like Maven. If you had to wait the length of time it takes P2 to figure itself out that would be unacceptable in standard Maven CLI use.

Requirements of a final 3.x solution

1. maintain the ability for a user to checkout your code and run mvn install and have it work with no prior setup on their part.
2. be able to depend on some jar and not worry about any repositories required for transitive resolution (ie discover the repositories transitively as dependencies are processed) *(this is controversial and may be eliminated. First it contributes to the Problem #4 above in that SAT can't be done on a bounded list of repositories. It also doesn't work normally behind a repository manager because the list of repos is usually controlled in the repo manager and thus autodiscovery is intentionally blocked, usually via a mirrorOf * to circumvent the repos maven finds in the poms.)*
3. be able to separate the dependencies needed by maven plugins from those needed by the build. This means not only where they are resolved from, but also how they are stored locally to prevent cross-contamination.
4. Repository identification: at this point we are pretty much in agreement that the URL should be the unique identifier for a repository. People who care about what they are publishing either need to use canonical repositories like Maven central or need to guarantee the existence of the repositories or have decent pointers. In a fully distributed system the relocation mechanism we have does not work in a fully distributed system without a master to manage relocations.

Proposed Solution Details

To be determined and populated AFTER the benefits / drawbacks and **specifically** requirements are gathered.