Artifact Identity

Improving Artifact Identity in Maven 2.1

Context

1. Alternative Versioning Schemes
2. Accommodating Integrator / Vendor Builds
3. Platform-specific Artifacts
4. Encoding Build Options into Artifact Identity
5. Alternative Artifact Resolvers and Repository Layouts

1. Version Schemes

In some ways, Maven has a built-in version scheme that it supports (or, should support). Version ranges and snapshot artifacts are two examples of this built-in concept. With version ranges, it's established that 1.0-SNAPSHOT predates 1.0-beta-1, which predates 1.0 (or, it should be). With snapshots, there is a concrete file name pattern that must be obeyed for updates to happen normally.

All of this is problematic if your company/project uses a different set of labels to track artifacts through the development cycle. If you use 0.1, 0.2, 0.3, ... to signify pre-releases of 1.0, then version ranges are useless. Even if your policy is to use 1.0b1 to signify the first beta release of the project, Maven's current (or planned) version range mechanism will not work for you.

2. Integrator Support

Projects are often rebuilt by system integrators and vendors, to ensure they get certain patches applied (not to mention any number of other artifact-stability reasons). In many cases, these artifacts need to be compatible (from an artifact identity standpoint) with the mainstream releases. Therefore, they should be considered as revision releases, and (depending on the update policy for the user performing the build) should be eligible to be handled as an automatic update of the artifact.

Going beyond the discussion about version schemes and range handling, integrator builds add a new level of complexity to version ranges and update policies. Currently, only snapshots can be updated in a slipstream fashion. However, automatically updated builds of a particular version of artifact may be desirable when you're paying someone to support (and, implicitly, patch) the releases of an OSS project. Also, it's important to know that an integration build of 1.0-1.myco (or whatever) is between 1.0 and 1.0.1 when merged with the mainstream OSS release stream of that project.

3. Platform-specific Artifacts

Some programming languages (most, in fact) produce platform-specific binaries. Even if the programming language itself supports portable binaries, certain other aspects of the platform (think EJB server, or JNI) still force the production of binaries that are targeted at a particular platform. In order to support binary builds for these projects, users will require a way of capturing platform information for later querying, such that another build attempting to use one of these binaries as a dependency will find the right one for its own target platform. In some cases, this platform information may be provided by the user, either as an override for the target, or when the platform information cannot be detected automatically. In other cases, platform information may be more accurate if detected directly from the running system.

4. Capturing and Encoding Build Options

Associated with the problem of capturing platform information as part of the artifact's identity is the issue of capturing the build options used to product the binary. In many cases, conditional compilation can produce dramatically different resulting binaries. For example, in a C project the locations of dependency libraries on the system will often be compiled directly into the binaries (called rpath, I believe). If someone attempts to install or use the resulting binary on a system where one of the dependencies is in a different location, the binary will fail.

This principle extends all the way out to cases where the build may need a copy of the artifact with debugging symbols enabled. In fact, there are so many possible build options in an average C project that it's not practical to encode this information in a simple filename. Yet it's important that the build system be able to retrieve the most appropriate artifacts given the project's declared dependencies.

5. Alternative Artifact Resolvers and Repository Layouts

There have been several requests from non-traditional Maven users (linux distribution managers, IDE/platform developers, etc.) to allow a means of artifact resolution which is dramatically different from that currently supported. For instance, when building C packages on Fedora, it would

often be useful to specify a dependency which can be resolved from the list of installed in the RPM database, and use that dependency for compile-time flags. Also, when building Eclipse plugins or features, it would be useful to have an artifact resolver which could use the plugins/fragments that are already present in the Eclipse instance. Also included in these sort of exotic examples might be libraries contained within the JDK itself, like tools.jar, or in the standard extensions dir, like xerces or something.

Problem

Maven currently supports only one concept of artifact identity, with a single version scheme. Snapshot builds are the only type of artifact that can vary without varying the declared dependency in a POM, and snapshots have a concrete version format when deployed to a remote repository. Artifacts are uniquely identified by:

Resources

1. Improving Maven2 Dependency Resolution
2. Extending Maven 2.0 Dependencies
3. Support for other languages