Moving forward with the POM data model

Introduction

The Maven POM format has been stable for several years. It is described by an XML schema ([http://maven.apache.org/xsd/maven-4.0.0.xsd](http://maven.apache.org/xsd/maven-4.0.0.xsd)) that makes no allowance for extension. Thus, tools that read POMs are entitled to expect POMs to follow the schema precisely.

For Maven to add features and resolve long-standing problems, we need new information in POMs. To cite one example, the `<scm/>` element is groaning under the burden of comprehensive support for git. However, simply publishing a new schema is risky. Even if the new structure is entirely backward-compatible, existing tools may malfunction when presented with unexpected XML elements. It's important to be very clear about this: even adding a new element to the POM with conventions that the old elements continue to contain the equivalent subset is unsafe. Informal XML parsers can malfunction when presented with new elements.

Hypothetically, the namespace rules of XML schema could permit POM extensions in other namespaces. However, this is no solution. For one thing, many XML parsers in the Maven ecosystem are not namespace-aware. For another, namespaces have problems of their own, and are not well-suited. It would be one thing if the POM had been designed around namespaces (c.f. Spring context XML files). But it was not.

Therefore, Maven needs a compatibility scheme that allows new versions to consume new POM formats, while leaving old versions, and other tools, consuming the old format(s). This document attempts to capture a design solution for this problem that has been discussed from time to time on the dev list.

In thinking about compatibility, it's important to distinguish two situations: using a POM from a published artifact in building the dependency tree, and building a project.

Once a POM is published in a release, it's very important that all reasonable versions of Maven will interpret it in the same way for dependency management.

However, it's not required that any arbitrary version of Maven be able to correctly build any project. This is the purpose of the `/project/prerequisites/maven`. An interesting question below is whether there should be a coupling between the POM version and new information in the POM.

Basic Design

The fundamental idea of this solution is to add a revision number to the artifact type of a POM. Right now, all POMs are of type 'pom'. In this design, you can think of 'pom' as equivalent to 'pom4', and anticipate that the next step is to have new artifacts of type 'pom5'.

On the surface, this seems simple and insufficient. Some new version of maven emits and consumes 'pom5' artifacts, and is thus completely disconnected from previous versions of Maven. That isn't going to help anyone. Compatibility requires more complexity, as follows:

- Down-conversion is required: It must be possible to derive a 'pom4' model from a 'pom5' model. While this process will inevitably lose information, the resulting 'pom4' model must at least permit a broad range of relatively ordinary projects to build.
- New Maven reads Old Pom: When looking for a POM in a repository, if the 'pom5' Maven fails to find a 'pom5', it must look for, and consume, a 'pom4'.
- Publish old for new: Installing or Deploying a 'pom5' must, in parallel, install or deploy the corresponding 'pom4'. Thus, ordinary deployment from the pom5 Maven produces an artifact consumable by old tools.

There has to be a way to refuse compatibility: The existing mechanism for specifying a minimal Maven version has to be clarified or extended. To the present reader, at least, it's unclear to me whether the `<prerequisites/>` specification for the Maven version applies only to builds, or also to consuming the POM to resolve transitive dependencies and other artifact information. There should be a way to say, 'I know that this POM is unsafe going backwards, don't publish pom4'. We hope that this is used rarely, but we cannot anticipate all conditions. For extra credit, we could live up the the intention of the doc and extend this to specifying minimal versions of arbitrary plugins.

Do we need round-trip?

In theory, repository managers, at most, read 'pom' artifacts and treat unknown types as opaque. Thus, the design above should work without any initial changes to any repository manager. What if, however, some repository manager refused to cooperate in storing 'pom5' artifacts? I hope that this question is completely hypothetical. If it isn't, we could consider some scheme of encoding 'pom5' data in special comments in 'pom4' files. This leads rapidly to a mare's nest of complexity, so I hope we don't have to go there.
Conventions for Extending the POM

What kinds of XML constructs should be used to make the POM more extensible? Here are some categories:

Attributes

Adding attributes, when they make sense, is a very effective way to make small additions that are backward-compatible. No sane piece of Java code is going to refuse to process a POM because it sees an unfamiliar attribute. Attributes are simply too common of a mechanism for extension and annotation.

In fact, the community seems to have a consensus to relax the annotation model before adopting 'pom5'. By changing an option to modello, we can get a schema that explicitly allow xs:anyAttribute on all the elements.

Since maven POMs carry a schema URI and URL, we can make this change without the whole mechanism above. New POMs will have the new URI, old ones, the 4.0.0 URI. Any tool which does validate will validate against the schema called out, and all will be well.

What sort of information makes sense via attributes? The example of the moment is to control how information flows down from parents to children. This is how attributes are already used in Maven for plugin configuration. Extending that approach to other elements (/project/distributionManagement/site/url, or /project/scm/connection, e.g.) is logical. It's also reasonable from a compatibility standpoint. These items are not part of dependency resolution, so adding to their semantics can't break published artifacts.

Imagine that we define something like <url inherit='false'> to mean 'Don't allow children to inherit this URL.' Perhaps this should implicitly set /project/prerequisites/maven to the version of Maven that first supported this, to avoid incorrect builds with old versions of maven? Except, of course, that this would require a time machine to apply to maven 2.2.1 and before.

There's a bit of a dilemma in the use of attributes. Many people complain about the sheer verbosity of Maven POMs. Using attributes is often far less verbose. On the other hand, it would be kind of awful if you had to guess whether some bit of information was an attribute or a child element. The inheritance control is a good example of attribute use, because it is specifying a fact about this particular element. Going down the path of, oh, <forkMode v='always'/> to reduce the verbosity is perhaps not a great idea.

Namespaces

Namespaces are a subject of some controversy in the world of XML. The decision of the HTML5 junta to run away with them is a reason to pause before diving into them willy-nilly. Nonetheless, I think that they make some sense in POMs.

The least controversial application of namespaces is 'someone else's data'. It would be really delux if other tools (m2e, e.g.) could mix their information into POMs, without fear of collision of distress, by putting it in their own namespace.

Sadly, there is a lot of namespace-unaware parsing going on in Maven itself and in plugins. Thus, I think that opening the doors to namespace usage has to be part of POM5. We would add some xs:any's modelled on those in the XML Schema schema itself, to allow 'foreign schema' content in every element.

A second possible use of namespace is to organize the POM itself into multiple namespaces. A model for this would be Spring IoC context XML files. Each lump of functionality lives in a namespace. Anyone who wants to elaborate this approach should read and reject the HTML5 rhetoric.

It would, on the other hand, be a bad idea to try to use namespaces as a versioning technique. Users will not have much fun trying to remember namespace prefixes for elements when the only organizing principle is chronology.

Properties, properties, properties

It's interesting to contrast the very tight control of the POM schema in general with the completely uncontrolled situation inside /project/build/plugins/plugin/configuration and it's several equivalents. Maven could have demanded that each plugin define a schema for it's configuration. That would have permitted more complex or tuned XML syntax for plugins – but it would also have forced all the plugin authors to cope with schema management or use modello. Instead, the <configuration/> element is the wild west. Anyone can put anything in there, and there's a gentle-plugin's agreement not to complain about unexpected content. This can be bad when users mis-spell and get no diagnosis.

What if we applied the same principle to the chunks of the POM like <scm/> or <distributionManagement/>? Each of these elements serves a dual purpose: it documents facts about the project, and it feeds configuration to a subsystem or a plugin (release, deploy, site). OK, if it's in the configuration business, let's give it a <configuration> element, with arbitrary properties inside. If git required 27 additional parameters to the scm provider for the release plugin to work, then we have a Map of properties from a <configuration> element to pass to the scm provider. If we need a 28th, no stress, no new POM version.

I don't know if it would be safe to do this before we do POM5, if we restrict it to 'non-dependency-tree' elements. It seems to be an experimental proposition: if old versions of Maven cheerfully ignore additional elements in these places when building the dependency tree, we could do it. There would always be some risk of some other tool tripping over this information and failing.