Maven 3.0.x

Maven 3.0 – Jason van Zyl

Discussion: need some answers to these before even pushing this to the list
TODO: Jesse and Greg spent a lot of time getting the async SSL working so a little description this work would be useful
TODO: architecture document about Mercury transport as the async HTTP/DAV client
TODO: example of user facing API for Mercury
TODO: architecture document and spec for mercury (largely in the wiki)
TODO: example of user facing API for maven-shared-model
TODO: architecture document on maven itself, plugin manager, lifecycle executor, profile construction
TODO: check with kenney to see if his work survived in substituting components or if it's his work that's actually making it work

Technical Preparation: not necessary before discussions can start but helpful to help guide how we concretely determine backward compat
TODO: get an explanation of the process Arnaud and Benjamin have for plugins. I started by capturing the log
TODO: document standard setup for Hudson we have so people can see the results of testing
TODO: setup hudson with emma for code coverage and ask VELO to help us setup coverage for integration tests

Architectural Goals

Backward Compatibility

We must ensure that plugins and reports written against the Maven 2.0.x APIs remain to work in 2.1. We don't want people to have to rewrite
their plugins. There are several plugins that are using the current artifact resolution code that will not be supported (please see the Mercury section below). The
ones that are in our control we can port over to use Mercury, and external users will have to deal with the major version change. Most people will not be affected and
Mercury will be a far better solution.

We must also ensure that POMs of version 4.0.0 are supported in 2.1.x along with the behavior currently experienced. We are relying heavily on our integrations tests right
now but as we move forward the work that Shane is doing on the project builder with maven-shared-model will help us to accommodate different versions of a POM, and different
formats we decide to support. The maven-shared-model code has no limitation to formats of XML, or any of format like YAML, script, or anything else anyone can dream up. These
implementations may find use outside of Maven. For example someone might build something with the Maven Embedder, JRuby, and Mercury to create a JRuby-based system. The
same could be done for Groovy, or Intercal.

Plugin and Extension Loading

Maven's mechanisms for loading plugins and build extensions has been refactored. You can find more information in the Maven 2.1 Plugin and Extension Loading Design document.

Lifecycle

Aggregator Mojo Handling

Aggregator mojos bound to the lifecycle have been deprecated. This practice can produce some very strange results, and isn't really the right solution for many of the problems it attempts to solve. I'm hoping to include some better options for bracketing the normal build - both before, and after, explicitly - to make aggregator mojos obsolete, but for now they've been deprecated to avoid disrupting backward compatibility.

Also, aggregator mojos that **are** bound to the lifecycle will only be allowed to execute at most once during the build, to limit redundant execution. These mojos are meant to act on all projects in the reactor at once, and binding them to one pom.xml file is dangerous in that it can produce different build results depending on whether that pom.xml is included. This is further complicated if two modules in a reactor configure the same aggregator mojo...in which case, it may run multiple times...or, when the aggregator is configured in the parent pom, where it will run for each descendant module.

Plugin API

Shading of plexus-utils causing a ClassCastException on plugin.getConfiguration() (MNG-3012)

The fact that plexus-utils is hidden from plugins in the newer releases of Maven means that plugin.getConfiguration() from maven-model can cause a ClassCastException, if used from within a mojo. The plan to fix this is basically just to import Xpp3Dom from the shaded plexus-utils version in maven-core within the plugin's classrealm. This should allow us to share the same instance of that class (only, shouldn't really affect other p-u classes) and preserve backward compatibility for existing plugin releases.

comment from kenney:
The problem with this is that it's a hack. If xpp3dom/plexus utils is updated and the plugin requires the new xpp3dom class, which has a new method for instance, this will break the plugin.

About this specific issue (MNG-3012): The best solution is to only share java., javax., and core maven api classes, so we can no longer export anything outside the plugin api (which includes maven-model, maven-project, maven-settings e.a.). This would require to phase out plugin.getConfiguration() and other model methods that return xpp3dom classes, and let them return interfaces present in the maven core api. Those interfaces would have an implementation class that implements both that interface and extends xpp3dom, which will be hidden for the plugin. Another solution could be to use xmlplexusconfiguration

There's one more solution to consider; using ASM to rewrite plugins as they're loaded. We could add code modifiers that workaround incompatibilities by detecting usage patterns, like (Xpp3Dom) plugin.getConfiguration(). An example could be to modify the code to wrap Xpp3DomParser.parse(new StringReader(String.valueOf(/plugin.getConfiguration()/))) around the call. This is even more of a hack though. Perhaps a mojo that scans for plugin incompatibilities using ASM is more feasible (no code modification).

So the basic problem we're up against is that there can be core api changes between major versions that pose incompatibilities for plugins written against an older version. The simplest solution would be to let plugins specify the maven versions they work against (which is partly present:
<requires><mavenVersion>2.0.6</mavenVersion></requires>. If this field supports a versionrange, or we'd default the version interpretation above to mean [2.0.6,2.1), we can detect plugins that won't run. The shading mentioned above is solving only one incompatibility problem, and there are bound to be more. Maybe we even need 2 versions of a plugin at some point, targeted toward different maven versions, though I'd really like to avoid that. But we cannot just assume our 2.0 plugin api will never change across 'major' (read: minor) releases.

Strategies for ensuring backward compatibility

- Plugin integration testing

Benjamin, Dennis, Arnaud, and Olivier have been improving the harness we have to for running integration tests on the plugins. So it will be easier to test a plugin set with a given installation
in Maven.

POM changes

There are many changes that users have requested in the POM, in addition to wholesale formatting changes. Acommodating these requests is a little tricky
because we need to support different versions simultaneously so that if projecta A builds with 2.0.x, project B can consume the project A POM using 2.1.x.
We just need some way to easy support multiple versions and support mediation between the different versions.

- Tags: loose categorization people to use to help categorize as they see fit
- Categories: more standard categories that form over time by using category structures that exist or common tags that are used so often they become categories
- Dendency excludes: being to transitively exclude a dependency
- Properties on dependencies
- Specification Dependencies
- Schematron/RelaxNG descriptor for each plugin – Bryon Jacob proposed a flexible model but XSD is hard to fight here so I'm not sure how far this will go

Embedding

Full embedding of the Maven core is a major feature of the 2.1.x line. The embedder was created primarily for IDE integration and is now being consumed by m2eclipse, Mevenide and IDEA,
but the embedder is also used by the Maven CLI to ensure parity between IDEs and the CLI as much as possible. To understand how the embedder work you can refer to
the Maven Embedder documentation.

Custom Components

As discussed in Substituting of Custom Components we now have two ways
to insert new components into the system.

- Using a directory and specifying it in the Classworlds configuration. Tycho simply has a special set of components that load first before the standard maven components and they override
the standard Maven components. Here's the example based on what Tycho is currently doing which allows custom components to be used.

```
main is org.apache.maven.cli.MavenCli from
plexus.core


set maven.home default ${user.home}/m2


[plexus.core]

load ${maven.home}/tycho/*.jar

load ${maven.home}/lib/*.jar
```

- The embedder has the ContainerCustomizer which allow you to inject new component descriptors. This is used in the IDE integration (m2ecipse, Netbeans) for adding
  custom artifact resolvers.

But what we ultimately need for Tycho is a way to dynamically pull in a set of components based on the packaging of a project. In our case with Tycho the packaging is
maven-osgi-bundle and that should kick in the set of components that do builds for OSGi bundles. We also have another use case in Tycho where we are building OSGi bundles
without a POM and actually using a manifest. In this case we need to somehow detect the manifest and then have the custom set of components kick in. In the case of Tycho
we need a different project builder, and artifact resolver.

Mercury

Mercury is a replacement for the current Maven Artifact subsystem, and a complete replacement for the HTTP and DAV portions of the existing transport.

The primary reasons for replacing the code are that it is unmaintainable and nearly impossible to navigate, it uses completely non-standard structures and libraries for
version calculations, the API is too hard for people to use, and it is not given to users to consume as a single componment to use. Users are forced to know how several
complicated components interact in order to implement a mechanism of retrieving artifacts from a repository. The entire mechanism needs to be replaced with something
that can be maintained and is reliable.

Mercury started as some fixes to Maven Artifact to first help with embeddability and error reporting for IDE integration. This was a direct result of all IDE integrators having to reimplement the current artifact resolver to provide decent feedback to users when errors occured. The artifact subsystem would just die and leave the IDE in an unusable state. Milos was the first to implement his own artifact resolver, and Eugene soon had to do the same in m2eclipse. Oleg and I were also trying to use the current artifact mechanism in an embedded mode for some Eclipse plugins and this also proved to be quite painful. After the first attempt of removing the fail-fast behavior, Oleg and I decided to make a break from the old codebase and attempt to create Mercury with the following goals in mind:

- Find the best people in the world to help create an awesome HTTP and DAV implementation. We did this by talking to Greg, Jan, and Jesse who are the Jetty folks and there just isn't anyone who knows HTTP better. Greg and Jan are awesome, and Jesse is Maven committer so we have some deep understanding of the issues involved. So what Oleg and I wanted to see was:
    - Easy SSL support where mucky with certificates in the default install is not required.
    - Connection pooling
    - Connection parallelization
    - Built in DAV client support for deployment
    - Atomic retrieval: we make sure absolutely everything is been safely transported to disk before we place it in the local Maven repository
    - Atomic deployment: in this case we could only support this using a special filter Greg created which blocks requests for any artifacts being deploy in the current set until the entire set land safely to disk. So it becomes impossible to ask for an artifact that refers to something else in the set before it is actually available.
    - Starting thinking about a client that can understand GeoIP. Given the recent spikes in traffic we are going to start needing to distribute the load.
- Find the best solution possible solution for dealing with version calculations, in particular ranges. For this we called on Daniel Le Berre and ask for some help in integrating his SAT4J library. We learned about the SAT4J library from the P2 project over at Eclipse.org at the last EclipseCon. SAT4J was deemed the best way forward by the P2 team in providing the most reliable, and most workable solution for doing version calculation. SAT4J provides ways to plug-in strategies to deal with our scopes, conflict resolution strategies and it is deadly fast. We felt we are in good company as we can call on Daniel and the P2 team and collaborate when difficult problems arise.
- Find the best people to help with with security. This might an SSL-based solution to secure the channel where the source is known to be safe, a PGP-based solution where the contents must be secured assuming a hostile channel, or a combination of the two. To that end I have contacted the folks at the Legion of the Bouncy Castle and asked them to provide us the expertise to implement a safe and correct solution. I have not persued any help on the SSL.

So in the end I believe it would be detrimental to use the Maven Artifact code in the 2.1.x tree and the change needs to be made to use Mercury before the first alpha ships. Oleg and I started this work, and Oleg has subsequently worked tirelessly on Mercury along with a great deal of help from Greg, Jan and Jesse. I think Oleg understands the requirements as he's seen Maven in action in one of the largest development environments in the world and watched how Maven can fail spectacularly.

Plugin API

- Symmetric output expressions
- Java5 Mojo annotations (Yoav Landman has this working already)
- Clean separation of plugins from reports. It's not good that those are the same thing in the Maven internals.
    - Not using concrete XML classes in the Plugin API (Xpp3Dom)

Core Refactorings

- Project Builder (architecture)
    - Maven shared model work: a new way of reading in the models for Maven that is not format dependent in any way i.e. XML, text, YAML, scripts, whatever.
    - Pluggable model readers: this could leverage different implementations provided by the shared model work, but we still need a way to detect the type and version of the model that we want to consume
    - A new terse format that uses attributes
    - Automatic parent versioning
    - New interpolation component (plexus-interpolation)
    - Dynamic build sections (MNG-3530)
    - Mixin support – allowing a paramterizable template which can be imported with one line.
- Remove the use of separate plugin repositories. We only need to pull resources from one repository. We started doing this but I've had a couple clients that want to separate the tools they use from the code they are developing/building.
- Decouple script-based Plugins from the core – we are a large part of the way here I need to summarize what was done.
- Remove Settings from the core and make it a user facing configuration (This is primarily done – jason)
- Have one configuration model for request
- Have one configuration model for session: session takes the request in the constructor and delegates
- Domain logging
- Plugin Manager
    - Removal of the Plugin Registry (done) – we moved in a direction where people lock down their versions and we've helped by putting default versions in the parent POM.
    - Load Plugin dependencies into a separate ClassRealm (done)
    - Plugin Execution Environment: Ability to run any version of a plugin where an environment is created which contains all the requirements for a particular version of the Plugin API
- Lifecycle Executor
    - Queryable Lifecycle
        - The most important change in the embedding environment. You can actually query Maven for the complete execution before it happens. We must know the entire model of execution before we execute.
- OSGi-like Classloading to support isolated execution environments

Java 5

Java5 annotations for plugins: we have two implementations that have now been merged in plexus-cdc. QDOX 1.7 has now been released so we may want to check the source level gleaning again. Jason Dillon has created a working class processing model. We need to deal with Plexus components and Maven plugins.

Integration and promotion of scriptable plugins

Toolchains

- Milos has implemented this and Shane had some feedback so this needs to be linked together

Reporting

- Report Execution Environment: Ability to run any version of a report where an environment is created which contains all the requirements for a particular version of the Report API.
- Decouple the reporting core. We need to get Doxia out of the core. Anything it needs to run should be isolated.

Other Use Cases to Integrate

Determining project type in Eclipse (Igor Fedorenko)

Support for "java" projects in Eclipse has certain overhead and it is desirable to only enable for projects that actually require it. More specifically, java maven projects have JRE classpath container, maven classpath container, have java-specific UI elements enabled and are offered in various java-related searches. Also, tools like WTP and AJDT treat (eclipse) java projects specially.

There is currently no direct way to tell if a (maven) project needs to be configured as java project in eclipse. The closest test condition I can think off is

1 the project ArtifactHandler language=java

and

2.1 ArtifactHandler addedToClasspath=true

or

2.2 MavenProject.getCompileSourceRoots().size() > 0

or

2.3 MavenProject.getTestCompileSourceRoots().size() > 0

(in other words the project is java and either itself is added to classpath or has sources to compile).

This test will return false negative for some WAR projects (not added to classpath and don't have any sources to compile). Also, compileSourceRoots and testCompileSourceRoots only become fully populated after running maven build lifecycle, which is expensive.

- Extensions
  - Different categories of extensions: providers vs packaging vs PMD/Checkstyle resources stuff in a JAR
  - Transparent Extension Loading
    - Any Wagon or SCM providers should get picked up automatically from SCM and distributionManagement URLs
    - Any extensions containing packaging/lifecycle related bits needs to be picked up automatically