

Dynamic POM Build Sections

Dynamic POM Build Sections: Supporting Change in MavenProject State

Context

Related JIRA Issues

- [MNG-3530](#)
- [MNG-3355](#)
- [MNG-2223](#)
- [MNG-2124](#)
- [MNG-1995](#)
- [MNG-1927](#)

Related Maven 2.0.x Feature Branch

- <https://svn.apache.org/repos/asf/maven/components/branches/john-2.0.x-dynamicBuild>

Discussion

Maven builds often include plugins that make changes to the project's internal state. In fact, this is one of the two main activities executed by most plugins - with the other being modification of project files themselves. Whether this involves setting the file in the project's main artifact, re-setting the build-directory paths, or modifying and introducing POM properties, these changes in project state must be reflected in the configurations used for subsequent plugin executions.

Problem

As plugins modify the internal state of the MavenProject instance during a build, these changes must be reflected in the configurations handed to subsequent plugin executions. If a project has a build directory in the POM of "foo", then MojoA calls `project.getBuild().setDirectory("bar")`, then this implies two things for plugins executing after MojoA:

- MojoB, defining a parameter as `outputDirectory`, and with a configuration in the POM of `<outputDirectory>${project.build.directory}/output</outputDirectory>`, should use the location `bar/output` when it executes.
- MojoC, defining a parameter for injecting the current MavenProject instance (using the `${project}` expression), and using the result of `project.getBuild().getDirectory()`, should use the location `bar` when it executes.

Currently (as of Maven 2.0.9), all expressions in the POM are resolved when the project instance is constructed. This means that expressions like `${project.build.directory}` are no longer dynamic, and therefore are locked down to their original values. Previously, this was the case with everything except build-path expressions like `project.build.directory` - for instance, a change in a POM property `<myProperty>originalValue</myProperty>` to `<myProperty>newValue</myProperty>` will still result in a plugin with a declared configuration of `<output>${myProperty}</output>` using the concrete value of `originalValue`.

The real problem here has nothing to do with build-path expressions, and everything to do with preserving the dynamism available in the POM as it is read from the filesystem, with expressions intact. If the state of the project

instance changes, the values of these expressions must change accordingly. At the same time, it's not enough to preserve the project instance in an unresolved form, and only do resolution when a parameter is injected, since plugins may opt to inject the whole project instance and navigate its state using the MavenProject and Model APIs. For example, this API-orientation becomes a factor if a plugin attempts to process the set of Resource instances declared within the current project. To illustrate, consider what happens if the POM declares the following:

```
<project>
  [...]
  <properties>

  <myResources>conf/resources</myResources>
  </properties>
  [...]
  <build>
    <resources>
      <resource>

      <directory>${myResources}</directory>
      </resource>
    </resources>
  </build>
</project>
```

If absolute dynamism is preserved by leaving expressions unresolved in the project instance itself, and the plugin retrieves the Resources from this MavenProject instance, the value for the Resource's directory, available through the API, will be the uninterpolated value:

```
${myResources}
```

While dynamism for expressions used in the build section of the POM is a good thing, it is not appropriate for everything in the POM. For instance, dynamically injecting new dependencies is absolutely not a good thing, since it affects Maven's ability to understand what external libraries a project depends on - in that scenario, the only way to get the full complement of project dependencies is to first run the plugin, then check the project dependency list.

From this, we can guess that there are certain parts of the POM that need to be variable in order to support things like forked-mojo execution, and other parts that must be as deterministic as possible. When forking a Clover

process, project classes must be instrumented, compiled, and tested without interfering with the project classes that are destined to be a part of the resulting project artifact; therefore, the build paths for the project must be variable. By the same token, these build paths must be interpolated using up-to-date project state, so build resources (the `<resources/>` section) reflect this information. Resources are object instances that contain fields like `directory`, `targetPath`, etc. and will be injected into plugins as complex objects. Unfortunately, Modello's generated I/O utilities do not allow them to be interpolated in isolation (without an owning Model instance), so the whole model must be re-interpolated to incorporate the most up-to-date information into the build section for use in plugin parameters. At the very least, such complex build objects should be updated to reflect changes in the overall project state before the next plugin is executed.

Since the information contained within the project's build state must remain mutable, it's not enough to simply copy the unresolved build section and interpolate it prior to each plugin execution. Because these plugins may modify the project's build state, all changes must be propagated back to the original, unresolved build section after the plugin completes its work. This will allow subsequent plugins to "see" the updated information. At the same time, any existing values that were interpolated for that plugin's execution - but remained unchanged - must be returned to their uninterpolated values, to accommodate re-interpolation to incorporate changes in the values these expressions reference.

Proposed Solution

One potential solution is to make exceptions of the `resources` and `testResources` sections of the build, and readjust them each time they - or the project instance itself - is injected into a plugin. We could then mask out the build section from interpolation, and preserve plugin configurations for just-in-time interpolation, prior to plugin execution. When plugin execution completes, any changes in the build section of the POM will be back-propagated its dynamic alter-ego, preserving any pre-existing expressions for later reinterpolation. In order to restore unchanged values to their uninterpolated state, all information in the build section must be tracked using two pieces of information: the dynamic, uninterpolated state, and the interpolated state that was originally injected into the plugin that just executed. When combined with the project state /after/ the plugin's execution, the original interpolated elements can serve as guides for reinjecting the dynamic expressions wherever project build state /hasn't/ changed.

To summarize, this solution involves masking out the build section of the POM before interpolating it at project-construction time, then re-interpolating the project's build state prior to each plugin execution, to calculate the concrete values that will be used in that execution. After the plugin executes, the project should be returned to its dynamic build state by restoring any unchanged elements to their original POM expressions, and incorporating any new or changed elements - copied from the build section used in the plugin execution, back to the "master" build section, for subsequent re-interpolation before the next plugin execution.

Notes

While this solution is rather heavy, it allows us to preserve a dynamic relationship between expressions used to configure plugins and current state of the project instance as successive plugins manipulate it. Obviously, a much simpler solution can be had by making the `MavenProject` and `Model` classes immutable - maybe even just interfaces that define accessor methods but not mutators - the providing a plugin-facing, public API for manipulating build state. This API could then track any inbound changes and modify only those sections of the project instance that need changing, without forcing the concrete resolution of any unrelated project state. However, these changes would require massive refactoring of core Maven APIs that plugins currently depend on heavily, and this in turn would trigger a cascade of plugin releases that are not backward compatible, just to comply with the new API introductions. While this may be an option in the Maven 2.1, 2.2, or 2.x timeframes, it simply isn't appropriate for the Maven 2.0.x code branch.