

Aggregator Plugins

1. Background

In Maven 2.x we have a boolean mojo annotation `@aggregator` with the following effects on the mojo execution:

Execution	For mojos executed directly from the CLI, the mojo will only be executed once and not per each project in the reactor. For mojos bound to a lifecycle phase, the mojo will be executed for each project where the lifecycle binding is present.
Dependency Resolution	If an aggregating mojo is annotated with <code>@requiresDependencyResolution</code> , the core will resolve the dependencies for all projects in the reactor and not just for the current project.
Forking	The annotation <code>@execute</code> on an aggregating mojo will fork the requested goal/phase on all projects in the reactor.

Besides, aggregating mojos often use the parameter expression `reactorProjects` or `MavenSession.getSortedProjects()` to get hold of all the projects in the reactor for processing.

The current design has some problems, especially if aggregators are bound into a lifecycle phase, so let's step back and look what we want to support and how this might work.

2. Use Cases

While we currently have only one annotation to request aggregation, we have at least two different use cases for it. The differences in these use cases as outlined next contribute to the problems we currently encounter with aggregation and its use.

Pre- and Post-build Hooks

Given a (multi-module) build, users might want to perform tasks before the lifecycle of the first project starts or after the lifecycle of the last project has completed. For instance, imagine a build like this:

1. Pre-build hook
2. Project 1
 - a. validate
 - b. (...)
 - c. deploy
3. Project 2
 - a. validate
 - b. (...)
 - c. deploy
4. Post-build hook

It's assumed that build hooks are implemented as regular mojos (with special annotations) and are introduced to a build via plugin executions defined in the POM. However, the `<phase>` element of such a plugin execution would

have a slightly different meaning. Instead of saying "bind this mojo into lifecycle phase xyz" it should be interpreted as "if the build executes to phase xyz or beyond, register this mojo as a pre-/post-build hook".

A further mojo annotation could be introduced to enable the plugin author to control whether a post-build hook should be called regardless whether the build failed before, i.e. to provide some finally-like cleanup.

An example use for a pre-build hook could be an Enforcer rule that checks the build environment before any of the projects start to build.

Sub-module Summaries

A probably more common use case is to post-process output from child modules in order to produce some aggregated/summarized output. In terms of build steps, this would look like:

1. Child 1
 - a. validate
 - b. (...)
 - c. deploy
2. Child 2
 - a. validate
 - b. (...)
 - c. deploy
3. Aggregator
 - a. validate
 - b. (...)
 - c. (aggregating mojo bound to e.g. package phase)
 - d. (...)
 - e. deploy

The important difference of such a summary mojo compared to a post-build hook is the interaction with the regular lifecycle. A summary mojo bound to say the `package` phase would be executed during this phase such that later phases like `install` or `deploy` of the current project's build have access to the output of the summary mojo.

Finally note that for the summary mojo to be able to aggregate the output from the child modules, the aggregator project needs to run after the child modules.

A concrete example for this type of aggregation is the production of aggregated API docs or other assembly-like output that should be attached to a project and installed/deployed alongside the main artifact. An aggregated site with summary reports is another example.

Scope for Aggregation

Orthogonal to the scenarios outlined above, we have to distinguish what part of a reactor build should be subject to aggregation. Consider the following multi-module hierarchy where the projects marked with (X) associate an aggregating mojo with their lifecycle:

- Top-Level Aggregator POM T (X)
 - Second-Level Aggregator POM S1 (X)
 - Child A
 - Child B
 - Second-Level Aggregator POM S2 (X)
 - Child C
 - Child D

Running `mvn deploy` on the top-level aggregator POM could have the following effects:

1. Invoke the aggregating mojo in each project it is declared in. In detail the following three mojo executions would result for our example:
 - a. Invoke aggregator on second-level POM S1, aggregating output from child A and B
 - b. Invoke aggregator on second-level POM S2, aggregating output from child C and D
 - c. Invoke aggregator on top-level POM, aggregating output from A, B, S1, C, D and S2
2. Invoke the aggregating mojo only in the top-most project it is declared in. For the example given, this would mean only one mojo execution by suppressing any other executions of the mojo in sub modules (of any depth):
 - a. Invoke aggregator on top-level POM, aggregating output from A, B, S1, C, D and S2

Both styles have their supporting use cases. For a summary mojo that produces an aggregated assembly, the user might not want to skip this assembly step just because he invoked the build from a higher level of the project hierarchy where an even bigger assembly is produced. For a pre-build hook like a validation step on the other hand, it might be preferable to run only on the top-most project (e.g. for performance reasons).

To address this distinction in aggregation scope, we might start off with new mojo annotations like `@aggregator top-level|project` that plugin authors can use to indicate the desired operational mode. But it seems this ultimately demands a new POM element to enable the user to choose the mode that fits his intentions.

Compared to Maven 2.x, the first style of aggregation resembles somehow the current behavior, i.e. the aggregating mojo being executed in each project it is encountered. The major difference however is the order in which the individual projects are executed. For the common setup where the aggregator POM is also used as parent POM, it would be build ahead of the child modules in 2.x, making aggregation of child output impossible right now.

Also note that the second style of aggregation does not necessarily mean the aggregating mojo is only executed once per reactor build. Consider this variation of the above example where the aggregating mojo is only declared in S1:

- Top-Level Aggregator POM T
 - Second-Level Aggregator POM S1 (X)
 - Child A
 - Child B
 - Second-Level Aggregator POM S2
 - Child C
 - Child D

When running Maven on the top-level project, it seems unintuitive to invoke the aggregating mojo on the entire reactor just because the user ran the build from a higher level of the project hierarchy where however the aggregating mojo is not declared. This would extend the effect of the aggregator to modules that are no sub modules of its declaring project S1. This is exactly one of the problems we have in Maven 2.x where an aggregating mojo bound to a lifecycle phase causes dependency resolution for the entire reactor although some modules haven't been built yet.

3. Realization

All the different use cases outlined above are the things that we might want to support in future Maven versions. Yet we historically have only this single boolean `@aggregator` annotation that does not tell which use case a mojo is intended to serve. It appears though that the majority of aggregating mojos out there is meant to provide summary mojos. Hence I propose the following behavior of Maven core:

Project Ordering

A project with packaging `pom` can serve both as a parent POM and as an aggregator POM. Inheritance belongs to the construction of the effective model and happens long before we reach the lifecycle executor and as such does not care about project order. Aggregation in the sense of a summary mojo however imposes a constraint on the order namely that the project with the aggregating mojo needs to be built after its child modules. For this reason, the project sorter needs to be changed to mark an aggregator POM as a dependant of all its modules. This is contrary to the related article (1) and the current behavior of Maven 2.x. The hopefully few cases where users setup an aggregator POM to produce some artifact for consumption by sub modules would demand to restructure the build and move the production of the artifact to a sub module of the aggregator.

Dependency Resolution

A mojo flagged as `@aggregator` should no longer trigger dependency resolution for the entire reactor but only for the sub tree of the project hierarchy where the aggregating mojo is rooted. For a mojo invoked directly from the CLI, this effectively makes no differences compared to Maven 2.x. For mojos bound to the lifecycle, this prevents dependency resolution errors on modules that due to the project order can never be build in time for the aggregating mojo.

Forking

Just as with dependency resolution, an aggregating mojo should no longer fork the entire reactor but only the sub tree of the project hierarchy it is relevant for.

Project Retrieval

What remains unclear to myself is how to handle the `reactorProjects` parameter expression in aggregating mojos. I am tempted to believe that those mojos don't really want all reactor projects but again only the sub tree of the project hierarchy they operate in. If this assumption proves sensible, it would fit the bill to change the semantics of the `reactorProjects` expression to only deliver the projects from the sub tree of the project hierarchy, thereby being in sync with the changes for dependency resolution and forking.

The obvious alternative is to leave `reactorProjects` as is and introduce a new expression `subProjects` or similar that only delivers the current project and all its (transitive) sub modules.

Project Hierarchy Tree

Internally, the core will need to keep the tree of projects that forms the project hierarchy as determined by aggregation, i.e. via the `<modules>` section in the POM.

Pre-/Post Build Hooks

The details of this are left open for future design. Right now, I simply assume we will introduce new mojo annotations to mark those goals and distinguish them from the summary mojos that continue to use the existing `@aggregator` annotation.

4. Related Articles

1. [Atypical Plugin Use Cases](#)
2. [Deterministic Lifecycle Planning](#)