

Suppression, Ordering, and Replacement of Plugins and Mojos Bindings

[PROPOSAL]

Related Issues and Articles

JIRA Issues

1. [MNG-2804](#)
2. [MNG-2805](#)
3. [MNG-2806](#)

Wiki Articles

1. [Deterministic Lifecycle Planning](#): Summary of build-planning features, foundational to the improvements outlined here.

1. Background:

A. Accumulation of Mojo-Phase Bindings

Currently, Maven supports additive mapping of mojos to lifecycle phases using four basic sources:

i. Lifecycle mapping given by `<packaging/>`

This lifecycle phase mapping is defined in the component descriptor (components.xml) for the artifact that defines a particular packaging. In some cases, this artifact is in the core of Maven (in the case of 'jar' packaging, for instance). In other cases, the lifecycle phase mapping may be specified by a build extension, or even a plugin that has `<extensions/>` enabled (set to true). In any case, this lifecycle mapping specifies the skeletal set of mojo-to-phase mappings that defines the basic build process for that type of project.

ii. Mojo bindings accumulated via inheritance

These are simply the additional mojo bindings specified in ancestor POMs that have `<inherit/>` set to true (which is the default setting if nothing is specified in most cases). For each level of inheritance, the mojo bindings that are added are appended to the list of bindings already in that particular phase.

iii. Mojo bindings specified in the current POM

When the current POM is built, all mojos bound in the POMs ancestry are appended to the base lifecycle mapping given by the project's packaging. After this happens, any mojo bindings specified in the current POM are appended to

this cumulative list of phase bindings. This means that the current POM's mojo bindings will always be inserted into the build process after the bindings given by its packaging and ancestry.

iv. Mojo bindings injected via activated profiles

After a particular POM's inheritance effects are calculated, active profiles are injected. When a profile is activated, its modifications are added to the POM in which it was declared. Only after all active profiles at that level are injected is that POM eligible for use in inheritance calculations, as a parent POM. Mojo definitions injected by a profile are appended to the list of phase bindings, after the POM's "normal" mojo bindings have been added.

This means that mojo bindings specified in an active profile will always be inserted into the build process after the bindings given by the base POM (not to mention ancestry and packaging, as described in (iii.) above) in which it was declared.

B. Shortcomings of the Additive-Only Approach

Since the lifecycle map is additive via inheritance and profile injection, with a skeletal base that's defined in a binary artifact, gaining visibility to the complete build plan is nearly impossible. This means that in complex inheritance hierarchies with profiles involved, it can be nearly impossible to understand what steps happen in which phase, much less know how best to modify this plan with a new mojo in the current POM.

Additionally, with a large inheritance hierarchy, it's highly likely that the lifecycle mapping will wind up with multiple mojos bound into some phase or other. This is not a problem in itself, particularly if the mojos don't depend on one another's actions. However, when the project POM (the leaf of the inheritance hierarchy) tries to replace a mojo that is listed ahead of some other binding in a single phase which depends on the mojo, it quickly becomes obvious that this is an impossible task. Suppressing that first mojo can be done via a custom parameter, and the project POM can add a new mojo binding to the phase to substitute for the skipped mojo which provides compatible functionality within that build. However, this new mojo binding cannot REPLACE the original (first) mojo binding; the second original binding (that depends on the actions of the first mojo) will ALWAYS run ahead of the new binding, and fail because there is now a step out of order in the build process.

C. The Traditional Solution: New Phases

Traditionally, Maven has solved this problem by binding the new mojo to an earlier lifecycle phase. If a suitable phase didn't exist, new phases were introduced in later releases to address the issue. This led to a proliferation of 'pre-' and 'post-' phases, with each addressing a problem that had no good solution until the next release of Maven.

This approach to solving the ordering problem assumes that there are a finite and calculable number of steps (mojo bindings) in any given build process, such that each step could occupy its own lifecycle phase in the lifecycle for that build. However, as the number of attached artifacts increases to

accommodate new types of metadata, reporting, or advanced build capabilities (such as providing buildable project-source or patch artifacts), so does the number of new mojo phase bindings to resolve, produce, package, install, and deploy these attachments. If a project packaging has a relatively high utilization of the lifecycle phases by default, and is used in a large development environment with a deep inheritance hierarchy (even 2-3 levels can be enough to display it), the lifecycle phase utilization can easily become saturated, resulting in multiple mojo bindings in some phases. Inheritance, profile injection, and the ability to specify arbitrary lifecycle mappings for custom packaging types means that it's simply not possible to calculate how many phases will be enough.

Additionally, for each new phase created, the existing phases lose a little of their meaning as verbs in the build process. When you just have 'resources', 'compile', 'package', etc. it's easy to understand the types of steps that might execute in each. However, what steps would you expect to take place in 'pre-package' or 'post-integration-test'? What about 'pre-resources'? The meaning of these new phases would be a little more watered down than the original list. As we add more and more phases, the meaning of each phase becomes a little less clear. If the aim is to create one phase for each possible mojo binding in any given build, phase names become somewhat irrelevant (since the distribution of mojo bindings for a given build is somewhat arbitrary)...they just become named slots whose names may or may not indicate a sequence.

Since one critical advantage of Maven over Ant or other build systems is its universal, intuitive set of verbs used to build any project, it is imperative that we preserve the meaning of these verbs and not water them down with subtle variations and modifiers. Aggregation of build steps within some verbs is inevitable if we are to retain a simple set of build verbs; we should embrace this aggregation or else abandon the vocabulary approach altogether.

2. Problem

Maven does not provide any means of suppressing, replacing, or reordering the list of mojo bindings for a particular lifecycle phase. As a consequence, developers are forced into extensive research on the applicable lifecycle mapping given by their packaging, plus the POM inheritance hierarchy and any applicable profiles along the way in order to determine what steps constitute the build process for their project.

Given the finite set of lifecycle phases available for a given build using Maven, along with the likelihood of multiple mojos binding to a single phase in complex build environments, developers must have an intuitive, flexible mechanism for expressing the steps required for their builds.

3. Requirements

- Provide a mechanism for specifying an ordering hint for a new mojo binding. This hint can be relative, since most ordering concerns will deal with putting one mojo ahead of another specific mojo. Both prepending and appending a new binding after a specific existing binding is required here.
- Provide a mechanism for replacing one mojo binding with another, without

needing to know where that binding is. Since the binding targeted for replacement may be defined in any number of hidden places (components.xml within a binary archive, or POM deployed on a remote repository), replacement by binding id (execution id, which implies plugin id) is essential.

- Provide a mechanism for disabling the execution of a mojo binding. This addresses similar situations to the last item, to avoid the need for constant research into the lifecycle bindings for a change to be made. For convenience, it may make sense to allow disabling of the entire plugin, and individual mojo bindings (executions).
- Simple, intuitive syntax.
- No modification of existing semantics or syntax; only additions that preserve backward compatibility are allowed.
- Tools must be provided to help analyze the build process, including the build-order which gives a step-by-step list of the mojos (and executionId's) that will be executed during the build.

4. Proposed Solution

A. The Basics

This solution involves adding two new sections to the **<execution/>** element of the plugin declaration in a POM, and one new element to the main plugin declaration section. In the execution section, we add a new section called something like **<phaseOrdering>**, and a new element called **<disableExecution>**. In the main plugin section, we add a new element called **<disablePlugin>**. The meaning of these new sections and elements will be described below.

B. Execution-Level Modifications

Inside the execution section, the new **<phaseOrdering>** section handles the ordering of the new execution in relation to some existing mojo or execution. It should contain the groupId, artifactId, and optionally, version of the plugin that specifies the execution or mojo, along with the executionId, and optionally, mojo name, along with an ordering type. Using these elements, the lifecycle executor would locate the plugin, execution, and (optionally) the mojo to which this section refers, and apply the specified ordering operation for the new execution in relation to the target execution/mojo. For instance, if the type (operation) was "before", it would inject the new execution just in front of the target execution/mojo in the build process. If the type operation was "replace" it would remove the target execution/mojo, and insert the new execution in its place. Valid values would be **insert-before**, **insert-after**, and **replace**. If this section is not specified, existing phase-binding rules will be applied to preserve backward compatibility.

In addition to the new **<phaseOrdering>** section, the new **<disableExecution>** element makes it possible to respecify a plugin execution using the same executionId, and then disable it. Since plugin executions can be merged through inheritance and profile injection using the executionId as a merge key, simply specifying an empty execution with a particular executionId and **<disableExecution>true</disableExecution>** should be enough to disable an

existing execution with that executionId brought in from a parent POM, a profile, or even the base lifecycle mapping brought in by the project's packaging. To disable a mojo supplied in a lifecycle mapping, use the default executionId of 'default'.

C. Plugin-Level Modification: `<disablePlugin/>`

Inside the plugin declaration itself, the new `<disablePlugin>` element makes it possible to turn off all executions (including the default one) of that plugin in the lifecycle. This will include disabling a binding of that plugin given in the lifecycle mapping itself. To do this, simply specify the plugin using `groupId`, `artifactId`, and optionally, `version`, then set `<disablePlugin>true</disablePlugin>`. All instances of all mojos in the plugin will be removed from the build process by the lifecycle executor.

D. New Mojo: `help:build-steps`

Finally, to aid in debugging lifecycle issues, a new mojo will be added to the maven-help-plugin, called **build-steps** or similar. This mojo will output a complete list of mojo bindings (with executionId), their phase attachments, and their ordering within each phase, in order to enable users to research insertion points for new mojo bindings and debug problems that may arise from the replacement of inherited mojo bindings.

5. Relevant Conversations

1. [Jesse McConnell and John Casey on IRC, 31-Jan-2007](#)
2. [Jesse McConnell and John Casey on IRC, 01-Feb-2007](#)

6. Paraphrased Feedback from the Maven Developers' List