

Shared Build Context for Components and Plugins

1. Background

A. Shared Context in Systems of Interchangeable Components

Most large, component-oriented systems provide a shared context for their components, so alternative component implementations are not dependent on a single, restrictive (or, even worse, all-anticipating) API to pass information across multiple subsystems. Such parameter-passing leads to bloated or inadequate APIs, since the API designer can choose either to try to accommodate every possible implementation for a component, or he can restrict the API to the minimum necessary to do the job with the original implementation. Resources like database connection pools are a good example of items that live in this shared context. Components that require such resources simply look them up, or are wired to use them by a composing container, instead of forcing all APIs in between the connection pool initializer and consumer to pass the pool instance around.

B. Relevance to Maven

Maven is a large, component-oriented application built on top of a container (Plexus) which is capable of this sort of composition. However, Maven's component implementations are currently restricted to the APIs used to pass information around the system. They don't have ready access to a shared context for reading build-time information. Also, they cannot coordinate build activities with one another. This lack of shared context leads to an unnecessarily restrictive set of component implementations and inefficient build processes. Maven component implementations - including, but not restricted to, plugins - must have the ability to retrieve as much information as possible about the current build environment.

2. Problem

A. No Support for Querying Dynamic, Build-Time State

Currently, Maven provides a mechanism for injecting static information about the current build into plugins. It also supports wiring plugins and other components with component requirements, which allow them to retrieve further information about the current build, though this information currently is not cached for successive querying in a threadsafe way. It does not provide a mechanism for retrieving dynamic build-time state or progress for custom components and plugins to use.

B. Embeddable Property-Passing

The simple act of passing System properties in an embeddable way (where "System" properties need to vary across multiple threads in a single Maven instance) introduces major API changes, since these properties, which are initialized at the beginning of the build, must be passed throughout the system until they reach the appropriate components. In some cases, these

properties must propagate through five or six layers of Maven APIs before they can be used. Yet in many ways, the System properties represent a shared context (though too widely shared, in the embedded scenario) by which component-specific configurations can be communicated from the application entry point to the appropriate component.

C. Plugins in Silos

Additionally, there is no way for plugins to mark progress in a build, or signal that they have detected a state where the build has not failed but also should not proceed. Multiple mojos within a single plugin can pass information by way of a context map, but this context map restricts the consuming mojos to usage scenarios that also involve their counterpart mojos, since they cannot function properly without their counterparts in place, regardless of whether the functions of these counterparts could be fulfilled in other ways.

3. Proposed Solution

A. The Build Context

As a solution to the problem of a build-time shared context, a new project called maven-build-context should be created. This project would provide a formal mechanism for storing and retrieving structured data from the Plexus container's Context map. By using a set of tools to access the shared context, it can be scoped appropriately (per-project, per-thread, etc.) in such a way as to avoid the problems encountered when using System properties in an embedded environment. By using structured data, it's possible to avoid datatype-conversion problems (coercing a String to an int, for example).

B. Plugin Cooperation

By using a shared datatype with a controlled vocabulary, it's possible for plugins to cooperatively determine whether to execute, based on the actions of other plugins that were registered in the shared context. Plugins that go further in detecting build configurations beyond that explicitly laid out in the POM could inject this detected configuration for use by other plugins. Such advanced state might include the source or method used to resolve each project dependency, so that dependencies from certain sources can be handled differently when the project is deployed (imagine building certain dependencies from source, then deploying those built dependencies alongside the main project artifact automatically).

C. Baking it into the Container

Ideally, these shared-context objects should be wired in as part of the composition process for dependent components, and injected into mojos along with things like `localRepository` or `project`. The act of querying the application for this build-time information makes it much more difficult to chart dependencies between components that produce and consume such information. Therefore, it's important to specify this information as declaratively as possible, in order to catch potential problems ahead of time, before the build is in progress. Rudimentary specification of a `<dependency/>`

element in each component's POM that points to the shared datatype is a good start, but still doesn't enable a more advanced validation that producer and consumer of this shared datatype are present and aligned properly.

4. Potential Drawbacks

A. Inter-Plugin Dependencies

Obviously, the biggest concern here is the ability to form inter-plugin dependencies by specifying a shared datatype in one plugin, and developing another plugin that depends on that datatype. However, such a dependency doesn't preclude a new plugin from specifying a binary (not lifecycle) dependency on the original plugin (the one with shared the datatype), then acting as an alternative producer. An even better approach would be to specify the shared datatype in its own project, then provide dependencies on this new project for each of the three plugins. By doing this, a developer effectively creates an open protocol for an infinite array of plugins to communicate with one another through the shared context of the build.

5. Relevant Conversations

A. Transcripts

1. [Jesse McConnell and John Casey \(transcript from IRC\)](#)

B. Paraphrased Comments from Maven Developers' List

from: Wendell Beckwith

It would be useful to allow reading Maven project instances from sources other than XML files, so we could create them from Manifest.mf/plugin.xml files for Eclipse plugins. Alternately, it would be useful to allow modification of the sorted project list in the reactor (possibly by way of a build-context data element) to allow a plugin to perform a second pass at project-building, and inject those built from Manifests, etc.

from: Brian Fox

It would be nice if Maven could detect whether any action was taken in a previous lifecycle phase, and avoid later actions if they don't need to be performed. Big example here is the jar plugin skipping repackaging if the compile/resources phases didn't execute.

from: Carlos Sanchez

This could be useful in passing along toolchain configurations from plugin to plugin, or component to component as necessary.