Sonatype Blog: Latest Posts

Create a Customized Build Process in Maven

August 5, 2009 By John Casey

Maven's build process is driven by three key concepts: the build lifecycle, mojos, and the lifecycle mappings. If you're not familiar with these concepts, you can read up on them in Maven, The Definitive Guide, especially Chapter 1 and Chapter 10.

Maven's basic unit of work during the build is the Mojo (Maven POJO). Mojos are contained in plugins that group them together with similar functions. Additionally, Maven supplies a standard set of scaffolds – called *build lifecycles* – that provide an abstract, structured progression of the types of activities (*lifecycle phases*) typically found in a build. The standard build for a particular type of project is defined when the appropriate mojos are bound to the appropriate lifecycle phases using a *lifecycle mapping*.

Obviously, this is only a starting point; individual projects can further fine-tune their own builds by binding or reconfiguring mojos in their own POMs. However, the default build for a particular type of project – specified by the *packaging* element in the POM – is defined by its lifecycle mapping.

Maven provides mappings for many common project packagings out-of-the-box. But what if we have a custom project type? Maybe something that produces a particular type of artifact that only our internal systems know how to use? If we're building a large number of these projects, it may make sense to teach Maven to support a custom project packaging. This is a relatively easy task, if you know a few tricks.

The Custom Lifecyle Mapping

In many cases, we're interested in building a project artifact that is loosely based on the `jar` archive layout, with some critical details such as files added to `META-INF/` that turn it into more than a run-of-the-mill classpath entry.

To create a lifecycle mapping for this sort of packaging, we're really interested in creating something that's basically a *jar++* mapping. In our case, we're going to define a new packaging type for plugins of a mythical app, called 'WeatherMaker'. We'll call the packaging `wm-plugin`.

As the foundation for the `wm-plugin` lifecycle mapping, we'll start by copying the standard `jar` lifecycle mapping component from the `components.xml` file in Maven SVN (**NOTE:** This is a fairly large file, so you might want to search for `role-hint>jar`). The `jar` lifecycle-mapping component definition looks like this:

```
<component>
  <role>org.apache.maven.lifecycle.mapping.LifecycleMapping</role>
  <role-hint>jar</role-hint>
  <implementation>
    org.apache.maven.lifecycle.mapping.DefaultLifecycleMapping
  </implementation>
  <configuration>
    <lifecycles>
      <lifecycle>
        <id>default</id>
        <phases>
          <process-resources>
            org.apache.maven.plugins:maven-resources-plugin:resources
          </process-resources>
```

```
            <compile>
              org.apache.maven.plugins:maven-compiler-plugin:compile
            </compile>
            <process-test-resources>
              org.apache.maven.plugins:maven-resources-plugin:testResources
            </process-test-resources>
            <test-compile>
              org.apache.maven.plugins:maven-compiler-plugin:testCompile
            </test-compile>
            <test>
              org.apache.maven.plugins:maven-surefire-plugin:test
            </test>
            <package>
              org.apache.maven.plugins:maven-jar-plugin:jar
            </package>
            <install>
              org.apache.maven.plugins:maven-install-plugin:install
            </install>
            <deploy>
              org.apache.maven.plugins:maven-deploy-plugin:deploy
            </deploy>
          </phases>
        </lifecycle>
      </lifecycles>
    </configuration>
  </component>
```

You'll notice that this lifecycle mapping only addresses the `default` lifecycle, but not the `clean` or `site` lifecycles. Since Maven's default mapping for these two lifecycles is sufficient for most use cases, most lifecycle mapping components don't need to define them explicitly.

Basically, the lifecycle mapping above lists the phases to which particular mojo bindings are made, and each mojo binding within a phase is listed in terms of its plugin coordinate – the `groupId`, `artifactId`, and optionally, `version` of the plugin – along with the mojo's own name, or `goal`. The format for each mojo binding is: `$groupId:$artifactId:[$version:]$goal`. If the version is omitted, Maven will search the POM's `plugins` and `pluginManagement` sections, and ultimately default over to the latest released version of the plugin if no other version specification can be found.

To build on the standard `jar` format, we're going to be using a mojo called `generate-descriptor` from a plugin called `weathermaker-maven-plugin` with a groupId of `org.sonatype.example.plugins` to generate a plugin descriptor file for inclusion in the project artifact. Since this descriptor is generated, and is a classpath resource (not something to be compiled), it should be bound to the `generate-resources` lifecycle phase.

Our modified lifecycle mapping looks like this:

```
<component>
  <role>org.apache.maven.lifecycle.mapping.LifecycleMapping</role>
  <!-- The POM packaging for this type of project is 'wm-plugin' -->
  <role-hint>wm-plugin</role-hint>
  <implementation>
    org.apache.maven.lifecycle.mapping.DefaultLifecycleMapping
  </implementation>
  <configuration>
    <lifecycles>
      <lifecycle>
        <id>default</id>
        <phases>

          <!-- Generate the plugin descriptor marking this artifact
                    as a plugin for the weathermaker application. -->
          <generate-resources>
            org.sonatype.example.plugins:weathermaker-maven-plugin:\
              ${project.version}:generate-descriptor
          </generate-resources>
          <process-resources>
            org.apache.maven.plugins:maven-resources-plugin:resources
          </process-resources>
          <compile>
```

```
            org.apache.maven.plugins:maven-compiler-plugin:compile
          </compile>
          <process-test-resources>
            org.apache.maven.plugins:maven-resources-plugin:testResources
          </process-test-resources>
          <test-compile>
            org.apache.maven.plugins:maven-compiler-plugin:testCompile
          </test-compile>
          <test>
            org.apache.maven.plugins:maven-surefire-plugin:test
          </test>
          <package>
            org.apache.maven.plugins:maven-jar-plugin:jar
          </package>
          <install>
            org.apache.maven.plugins:maven-install-plugin:install
          </install>
          <deploy>
            org.apache.maven.plugins:maven-deploy-plugin:deploy
          </deploy>
        </phases>
      </lifecycle>
    </lifecycles>
  </configuration>
</component>
```

NOTE: The backslash ('\') above indicates a continued line. This is very important!

We've made two important changes in the lifecycle mapping above: we changed the role-hint to match the new POM packaging, and we added the mojo binding to generate a weathermaker plugin descriptor. Again, the POM `packaging` element is keyed to this change:

```
<!-- The POM packaging for this type of project is 'wm-plugin' -->
<role-hint>wm-plugin</role-hint>
```

The customized mojo binding is added to the `generate-resources` lifecycle phase like this:

```
<!-- Generate the plugin descriptor marking this artifact
as a plugin for the weathermaker application. -->
<generate-resources>
  org.sonatype.example.plugins:weathermaker-maven-plugin:\
    ${project.version}:generate-descriptor
</generate-resources>
```

(Watch that backslash! It indicates a continued line!)

Also, notice the use of the expression `${project.version}` in our custom lifecycle binding. This isn't critical for now, but if we're releasing the custom lifecycle mapping as part of the weathermaker application, as part of a developer pack with the `weathermaker-maven-plugin`, it might make sense to key the lifecycle in to the particular version of the plugin with which it was released. At build time for this lifecycle mapping project (not projects that use the mapping), we'll filter a concrete value into the component definition in place of this expression, allowing it to stay up to date with successive application releases.

Of course, we could choose to add or subtract any number of mojo bindings to our custom lifecycle mapping. However, since the jar build process is fairly well-understood, it's generally good practice to change as little as possible to avoid unwanted side effects. At this point, we can include this new component in a `components.xml` file, wrapped in the following XML:

```
<?xml version="1.0"?>
<component-set>
  <components>
    <!-- lifecycle mapping component goes here -->
  </components>
</component-set>
```

Packaging The Lifecycle Mapping

Now, we can include this in a skeletal Maven project, inside the standard resources directory structure:

```
|- weathermaker-plugin-lifecycle/
|  |- pom.xml
|  |- src/
|  |  |- main/
|  |  |  |- resources/
|  |  |  |  |- META-INF/
|  |  |  |  |  |- plexus/
|  |  |  |  |  |  |- components.xml
```

…and add a basic POM:

```
<?xml version="1.0"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 \
                   http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.examples.weathermaker</groupId>
    <artifactId>weathermaker-parent</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <artifactId>weathermaker-plugin-lifecycle</artifactId>
  <build>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
        <filtering>true</filtering>
      </resource>
    </resources>
  </build>
</project>
```

(Again, backslashes indicate line continuations.)

Note that this isn't *quite* your run-of-the-mill POM. We're redefining the default resources directory to be filtered, so the weathermaker project version will be filtered into the lifecycle-mapping component, and tie our custom mojo bindings to a particular version of the weathermaker-maven-plugin.

Once we build this project, we'll have a jar artifact that we can reference as a build extension in our project POMs, so we can use the new lifecycle mapping.

It may look like our job it finished, but we're not quite finished yet. Read on!

Additional Details: The ArtifactHandler Component

As long as we're fine with having our new weathermaker plugin artifacts deployed into the maven repository with the extension `.wm-plugin`, and we're not concerned with having these plugin artifacts added to the classpath when they're specified as dependencies in other projects, we're basically done. However, since we're interested in creating weathermaker plugins that depend on other weathermaker plugins (even if they're expected to be provided by the core weathermaker application), this isn't a great solution for us. Additionally, since wm-plugin artifacts are really just jars with some additional special sauce, it might be a lot less confusing to name them with the standard `.jar` extension.

To achieve these goals, we need to define a custom `ArtifactHandler` component to accompany our new lifecycle mapping. Artifact handlers provide Maven with some basic handling instructions for artifacts (and dependencies) that match a particular *type*.

The handler component may map the given type to a different extension; specify that it is self-contained and includes its own dependencies; define the language in which the artifact is implemented; or tell Maven whether the artifact is intended to be used as part of a classpath, maybe for use in compiling the current project's source code. Fortunately, the `ArtifactHandler` component definition is simple and straightforward.

Again, we can base the handler for our `wm-plugin` artifacts on a pre-existing handler. However, this time we'll use the `ArtifactHandler` for `ejb` artifacts as our starting point (from Maven SVN):

```
<component>
  <role>org.apache.maven.artifact.handler.ArtifactHandler</role>
  <role-hint>ejb</role-hint>
  <implementation>
    org.apache.maven.artifact.handler.DefaultArtifactHandler
  </implementation>
  <configuration>
    <type>ejb</type>
    <extension>jar</extension>
    <language>java</language>
    <addedToClasspath>true</addedToClasspath>
  </configuration>
</component>
```

Just as with the lifecycle mapping, this component's `role-hint` element maps it to the POM `packaging` element. Additionally, the same `role-hint` element also maps the handler to the `type` element used when specifying a `dependency` in the POM. As you can see, there is a redundant `type` element in the configuration for this component; this element may be vestigial, but we'll specify it correctly just to be safe:

```
<component>
  <role>org.apache.maven.artifact.handler.ArtifactHandler</role>
  <!-- We want to look this up by dependency-type
            and POM packaging 'wm-plugin' -->
  <role-hint>wm-plugin</role-hint>
  <implementation>
    org.apache.maven.artifact.handler.DefaultArtifactHandler
  </implementation>
  <configuration>
    <!-- This should always be consistent with the role-hint,
                to be safe. -->
    <type>wm-plugin</type>
    <extension>jar</extension>
    <language>java</language>
    <addedToClasspath>true</addedToClasspath>
  </configuration>
</component>
```

As you can see, the `extension` is left with the standard value, `jar`. Aside from this, we're telling Maven that artifacts of this type are implemented in Java, and are meant to participate in classpaths.

Again, our customizations basically center on the role-hint:

```
<!-- We want to look this up by dependency-type
       and POM packaging 'wm-plugin' -->
<role-hint>wm-plugin</role-hint>
```

However, we change the `type` element to maintain consistency:

```
<!-- This should always be consistent with the role-hint,
        to be safe. -->
<type>wm-plugin</type>
```

Now, we simply add our new `ArtifactHandler` component definition to the same `src/main/resources/META-INF/plexus/components.xml` file that houses our custom lifecycle mapping. At this point, we **should** be able to rebuild the `weathermaker-plugin-lifecycle` project and be able to install, deploy, and resolve weathermaker-plugin artifacts using the filename extension

`.jar`. **However**, due to a couple known bugs in Maven versions prior to 2.2.1 (which is quickly nearing release), our new artifact handler will not quite work as expected for the install and deploy steps. To make that part work, we need to work around the bugs.

Working Around Known Issues

The most difficult and persistent problems in Maven mostly arise from timing issues, or what's commonly referred to as chicken-and-egg scenarios. In this particular case, we need to understand a little about how Maven works internally in order to understand why project builds don't use our new ArtifactHandler.

Maven builds project instances before it does almost anything else. It does so by collapsing the inheritance hierarchy of a particular POM, performing steps like expression resolution and some other intermediate processing, then initializing the project's main artifact.

Only after Maven has created project instances for each POM in the current build will it load any build extensions that are specified, and start executing the build lifecycle itself. The reason extension loading takes place so late in the process is because all configuration in Maven hinges on the project instance.

Any properties specified in profiles within the `settings.xml` file are injected into project instances; likewise, build extensions and plugins that are treated as build extensions are specified in the POM. Since these plugin specifications may need additional information from ancestral POMs, they cannot be loaded safely until the project instance is fully calculated.

What does this mean for us? Remember where the project artifact is constructed; when this happens, Maven uses all available ArtifactHandler definitions, and creates ArtifactHandler instances for any POM `packaging` or dependency `type` it encounters that doesn't already have a handler defined. As I mentioned above, the whole point of our custom ArtifactHandler was to escape the vagaries of these on-the-fly ArtifactHandlers. Since our custom ArtifactHandler isn't available when the project artifact is created, it's not used, and the project uses the on-the-fly version to install or deploy the project's build output.

To work around this shortcoming, we need to inject some custom behavior that executes at the beginning of the build, before the build has really done anything. This custom behavior will forcibly lookup all ArtifactHandler components that – for whatever reason – haven't been connected to the ArtifactHandlerManager component that Maven uses when resolving artifacts. It will also attempt to reset the ArtifactHandler instance used by the project artifact, forcing Maven to use our custom ArtifactHandler. The simplest way to inject this sort of code is by creating a custom Mojo and binding it to the `initialize` phase of the default build lifecycle.

The resulting mojo code is too long to list here, but you can see an example in the app-lifecycle-maven-plugin, which is used to build plugins for Sonatype applications such as Nexus.

Once we have the mojo, we still need to add it to our lifecycle mapping:

```
<component>
  <role>org.apache.maven.lifecycle.mapping.LifecycleMapping</role>
  <role-hint>wm-plugin</role-hint>
  <implementation>
    org.apache.maven.lifecycle.mapping.DefaultLifecycleMapping

  </implementation>
  <configuration>
    <lifecycles>
      <lifecycle>
        <id>default</id>
        <phases>
          <!-- Inject our custom ArtifactHandler into the project's main artifact,
                    and fixup any dependency artifacts that may be wrong. -->
          <initialize>
            org.sonatype.example.plugins:weathermaker-maven-plugin:\
              ${project.version}:inject-artifact-handler
          </initialize>
          ...
```

(Once again, remember that the backslash above denotes a line continuation.)

At this point, we've defined a custom lifecycle mapping to define the set of mojos that take part in building our custom weathermaker plugins, along with configuring where in the build they execute. We've defined a custom ArtifactHandler component to tell Maven that even though the POM `packaging` and dependency `type` elements say `wm-plugin`, it should actually be dealing with files that have an extension of `.jar`.

Finally, we've added a custom mojo to work around some known bugs in Maven that prevent project artifacts from taking advantage of custom ArtifactHandler components. Technically, we're done. We can rebuild the `weathermaker-maven-plugin` and `weathermaker-plugin-lifecycle` projects, configure our projects to use them, and we're ready to go.

But we could still streamline this a bit more to make it easier to use.

Packaging Revisited

Right now, we have two custom artifacts that must be available to build our weathermaker plugins: one jar filled with mapping component definitions, and a plugin artifact which the mapping

artifact references. Both of these artifacts have to be present for the build to work, and they will never be used separately.

So, why not combine them?

As you may have noticed if you read closely, there are two ways to define custom extensions in Maven: the `extensions` section, and using the `extensions` flag inside of a `plugin` definition. Up to this point, we've sort of assumed that we'd be using the `extensions` section of the POM, located directly inside the `build` section. This is the purest approach, in the sense that it's the natural choice if your artifact provides only lifecycle mappings, artifact handlers, and the like.

However, if your lifecycle mapping references a plugin that must be present for the lifecycle to run, then it makes a lot more sense to include that lifecycle mapping inside the plugin itself, and use the `extensions == true` flag in your POM's plugin configuration itself. This makes the plugin much more self-contained, and reduces the risk of one or the other artifact going missing for some reason (e.g. a network outage).

To include our mapping components inside the weathermaker-maven-plugin, we simply copy the resources directory over to the plugin's directory structure and update the plugin POM to filter those resources:

```
<build>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <filtering>true</filtering>
    </resource>
  </resources>
</build>
```

Once we've copied the resources and added the above POM configuration, we can scrap the weathermaker-plugin-lifecycle project altogether, and start using the weathermaker-maven-plugin exclusively.

Configuring Your Project POM

Yes, but…how exactly do we do that?

In the POM for our weathermaker plugin project, we simply add the following plugin configuration:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.sonatype.example.plugins</groupId>
      <artifactId>weathermaker-maven-plugin</artifactId>
      <version>1.0-SNAPSHOT</version>
      <!-- This is the magic configuration! -->
      <extensions>true</extensions>
    </plugin>
  </plugins>
</build>
```

Inclusion of this plugin with the `extensions` flag set to `true` will cause Maven to load our custom lifecycle-mapping and artifact-handler components and use them to execute our customized build process for the project.

Congratulations! You're done.

Resources

- Maven, The Definitive Guide
  - Chapter 1, Introducing Apache Maven
  - Chapter 10, The Build Lifecycle
- `maven-core` components.xml, from Maven 2.2.0 SVN (including the `jar` lifecycle mapping)
- `maven-artifact` components.xml, from Maven 2.2.0 SVN (including the `ejb` ArtifactHandler)
- inject-artifact-handlers mojo, from the `app-lifecycle-maven-plugin`, used to build Nexus plugins

Categories: Everything Open Source, Nexus Repo Reel
Tags: advanced maven, How-To, lifecycle, Maven

CLM

Nexus

About

Resources

Contact
General Inquiry
Report a Security Issue

Connect