introduction-to-the-lifecycle

Introduction to the Build Lifecycle

Table Of Contents

Build Lifecycle Basics

Maven 2.0 is based around the central concept of a build lifecycle. What this means is that the process for building and distributing a particular artifact (project) is clearly defined.

For the person building a project, this means that it is only necessary to learn a small set of commands to build any Maven project, and the POM will ensure they get the results they desired.

There are three built-in Build Lifecycles: default, clean, and site. The default lifecycle handles your project deployment, the clean lifecycle handles project cleaning, while the site lifecycle handles the creation of your project's site documentation.

A Build Lifecycle is Made Up of Phases

Each of these build lifecycles are defined by a different list of build phases, wherein a build phase represents a stage in the lifecycle.

For example, the default lifecycle has the following build phases (for a complete list of the build phases, refer to the Lifecycle Reference):

- `validate` - validate the project is correct and all necessary information is available
- `compile` - compile the source code of the project
- `test` - test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
- `package` - take the compiled code and package it in its distributable format, such as a JAR.
- `integration-test` - process and deploy the package if necessary into an environment where integration tests can be run
- `verify` - run any checks to verify the package is valid and meets quality criteria
- `install` - install the package into the local repository, for use as a dependency in other projects locally
- `deploy` - done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.

These build phases (plus the other build phases not shown here) and executed sequentially to complete the default lifecycle. Given the build phases above, this means that when the default lifecycle is used, Maven will first validate the project, then will try to compile the sources, run those against the tests, packages the binaries (i.e jar), run integration tests against that package, verifies the packaging, install the verifed package to the local repository, then depoy the installed package in a specified environment.

To do all those, you only need to call the last build phase to be executed, in this case, deploy.

```
mvn deploy
```

That is because if you call a build phase, it will execute not only that build phase, but also every build phase prior to the called build phase. Thus, doing

```
mvn integration-test
```

Will do every build phase before it (validate, compile, package), before executing integration-test.

There are more commands that are part of the lifecycle, which will be discussed in the following sections.

It should also be noted that the same command can be used in a multi-module scenario (i.e. a project with one or more subprojects). For example;

```
mvn clean install
```

This command will traverse into all of the subprojects and run `clean`, then `install` (including all of the prior steps).

A Build Phase is Made Up of Goals

However, even though a build phase is responsible for a specific step in the build lifecycle, the manner in which it carries out those responsibilities may vary. And this is done by declaring the goals bound to those build phases.

A goal represents a specific task (finer than a build phase) which contributes to the building and managing of a project. It may bound itself to zero or more build phases. And a goal not bound to any build phase executes outside of the build lifecycle. The order of execution depends on the order in which the goal(s) and the build phase(s) are invoked. For exmaple, the in command below. The `clean` and `package` arguments are build phases. While the `dependency:copy-dependencies` is a goal.

```
mvn clean dependency:copy-dependencies package
```

If this were to be executed, the `clean` phase will first be executed (meaning it will run all preceeding phases, plus the `clean` phase itself), and then the `dependency:copy-dependencies` goal, before finally executing the `package` phase (and all its preceeding build phases).

Moreover, if a goal is bound to one or more build phases, that goal will be called in all those phases.

Furthermore, a build phase can also have zero or more goals bound to it. If a build phase has no goals bound to it, that build phase will not execute. But if it has one or more goals bound to it, it will execute all those goals (_Note: as of maven-2.0.5, multiple goals bound to a phase are executed in the same order as they are declared in the POM_).

Setting up your Project to Use the Build Lifecycle

The build lifecycle is simple enough to use, but when you are constructing a Maven build for a project, how do you go about assigning tasks to each of those build phases?

Packaging

The first, and most common way, is to set the `packaging` for your project. Some of the valid `packaging` values are `jar`, `war`, `ear`, and `pom`. If no packaging value have been specified, it will default to `jar`.

Each packaging contains a list of goals to bind to a particular phase. For example, a JAR will bind the following build phases to the default lifecycle.

`process-resources`	`resources:resources`
`compile`	`compiler:compile`

process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar
install	install:install
deploy	deploy:deploy

This is an almost standard set of bindings; however, some packages handle them differently. For example, a project that is purely metadata (packaging value is pom) only binds the install and deploy phases (for a complete list of build-phase-to-goal binding of some of the {{packaging}}s, refer to the Lifecycle Reference).

Note that for some packaging types to be available, you may also need to include a particular plugin in your build section of your POM (as described in the next section). One example of a plugin that requires this is the Plexus plugin, which provides a plexus-application and plexus-service packaging.

Plugins

The second way to add goals to phases is to configure plugins in your project. Plugins are artifacts that provides goals to Maven. Furthermore, a plugin may have one or more goals wherein each goal represents a capability of that plugin. For example, the maven-compiler-plugin has two goals: compile and testCompile. The former compiles the source code of your main code, while the latter compiles the source code of your test codes.

As you will see in the later sections, plugins contain information that indicate which lifecycle phase to bind each goal to. Note that adding the plugin on its own is not enough information - you must also specify the goals you want run as part of your build.

The goals that are configured will be added to the goals already bound to the lifecycle from the packaging selected. If more than one goal is bound to a particular phase, the order used is that those from the packaging are executed first, followed by those configured in the POM. Note that you can use the executions element to gain more control over the order of particular goals.

For example, the Modello plugin always binds modello:java to the generate-sources phase (Note: the modello:java goal generates java source codes). So to use the Modello plugin and have it generate sources from a model and incorporate that into the build, you would add the following to your POM in the plugins section of build:

pom.xml

```
...
 <plugin>
   <groupId>org.codehaus.modello</groupId>
   <artifactId>modello-maven-plugin</artifactId>
   <executions>
     <execution>
       <configuration>
         <model>maven.mdo</model>
         <modelVersion>4.0.0</modelVersion>
       </configuration>
       <goals>
         <goal>java</goal>
       </goals>
     </execution>
   </executions>
 </plugin>
...
```

You might be wondering why that executions element is there. That is so that you can run the same goal multiple times with different configuration if needed. Separate executions can also be given an ID so that during inheritence or the application of profiles you can control whether goal configuration is merged or turned into an additional execution.

When multiple executions are given that match a particular phase, they are executed in the order specified in the POM, with inherited executions running first.

Now, in the case of `modello:java`, it only makes sense in the `generate-sources` phase. But some goals can be used in more than one phase, and there may not be a sensible default. For those, you can specify the phase yourself. For example, let's say you have a goal `display:time` that echos the current time to the commandline, and you want it to run in the `process-test-resources` phase to indicate when the tests were started. This would be configured like so:

pom.xml

```
...
 <plugin>
   <groupId>com.mycompany.example</groupId>
   <artifactId>maven-touch-plugin</artifactId>
   <executions>
     <execution>
       <phase>process-test-resources</phase>
       <goals>
         <goal>timestamp</goal>
       </goals>
     </execution>
   </executions>
 </plugin>
...
```

Lifecycle Reference

The following lists all build phases of the default, clean and site lifecycle, which are executed in the order given up to the point of the one specified.

Clean Lifecycle

`pre-clean`	executes processes needed prior to the actual project cleaning
`clean`	remove all files generated by the previous build
`post-clean`	executes processes needed to finalize the project cleaning

Default "build" Lifecycle

`validate`	validate the project is correct and all necessary information is available.
`initialize`	initializes the build process
`generate-sources`	generate any source code for inclusion in compilation.
`process-sources`	process the source code, for example to filter any values.
`generate-resources`	generate resources for inclusion in the package.

`process-resources`	copy and process the resources into the destination directory, ready for packaging.
`compile`	compile the source code of the project.
`process-classes`	post-process the generated files from compilation, for example to do bytecode enhancement on Java classes.
`generate-test-sources`	generate any test source code for inclusion in compilation.
`process-test-sources`	process the test source code, for example to filter any values.
`generate-test-resources`	create resources for testing.
`process-test-resources`	copy and process the resources into the test destination directory.
`test-compile`	compile the test source code into the test destination directory
`test`	run tests using a suitable unit testing framework. These tests should not require the code be packaged or deployed.
`package`	take the compiled code and package it in its distributable format, such as a JAR.
`pre-integration-test`	perform actions required before integration tests are executed. This may involve things such as setting up the required environment.
`integration-test`	process and deploy the package if necessary into an environment where integration tests can be run.
`post-integration-test`	perform actions required after integration tests have been executed. This may including cleaning up the environment.
`verify`	run any checks to verify the package is valid and meets quality criteria.
`install`	install the package into the local repository, for use as a dependency in other projects locally.
`deploy`	done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.

Site Lifecycle

pre-site	executes processes needed prior to the actual project site generation
site	generates the project's site documentation
post-site	executes processes needed to finalize the site generation, and to prepare for site deployment
site-deploy	deploys the generated site documentation to the specified web server

Furthermore, some phases have goals binded to it by default. And for the default lifecycle, the bindings depends on the `packaging` value. Here are some of the build-phase-to-goal bindings.

Lifecycle Default Bindings Reference

Clean Lifecycle Bindings

`clean`	`clean:clean`

Default Lifecycle Bindings - EJB / EJB3 / JAR / PAR / RAR / WAR Packaging

`process-resources`	`resources:resources`
`compile`	`compiler:compile`
`process-test-resources`	`resources:testResource`
`test-compile`	`compiler:testCompile`
`test`	`surefire:test`

package	ejb:ejb *or* ejb3:ejb3 *or* jar:jar *or* par:par *or* rar:rar *or* war:war
install	install:install
deploy	deploy:deploy

Default Lifecycle Bindings - EAR Packaging

generate-resources	ear:generateApplicationXml
process-resources	resources:resources
package	ear:ear
install	install:install
deploy	deploy:deploy

Default Lifecycle Bindings - maven-plugin Packaging

generate-resources	plugin:descriptor
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResource
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar *and* plugin:addPluginArtifactMetadata
install	install:install *and* plugin:updateRegistry
deploy	deploy:deploy

Default Lifecycle Bindings - POM Packaging

package	site:attach-descriptor
install	install:install
deploy	deploy:deploy

Site Lifecycle Bindings

site	site:site
site-deploy	site:deploy