Introduction to the Build Lifecycle

Table Of Contents

Build Lifecycle Basics

Maven 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 (./introduction-to-the-pom.html) 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 is defined by a different list of build phases, wherein a build phase represents a stage in the lifecycle.

For example, the default lifecycle comprises of the following phases (for a complete list of the lifecycle 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.
- `verify` - run any checks on results of integration tests to ensure quality criteria are met
- `install` - install the package into the local repository, for use as a dependency in other projects locally
- `deploy` - done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

These lifecycle phases (plus the other lifecycle phases not shown here) are executed sequentially to complete the `default` lifecycle. Given the lifecycle 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, package the binaries (e.g. jar), run integration tests against that package, verify the integration tests, install the verified package to the local repository, then deploy the installed package to a remote repository.

[top] (./introduction-to-the-lifecycle.html).

Usual Command Line Calls

In a development environment, use the following call to build and install artifacts into the local repository.

```
mvn install
```

This command executes each default life cycle phase in order (`validate` , `compile` , `package` , etc.), before executing `install` . You only need to call the last build phase to be executed, in this case, `install` :

In a build environment, use the following call to cleanly build and deploy artifacts into the shared repository.

```
mvn clean deploy
```

The same command can be used in a multi-module scenario (i.e. a project with one or more subprojects). Maven traverses into every subproject and executes `clean` , then executes `deploy` (including all of the prior build phase steps).

[top] (./introduction-to-the-lifecycle.html).

A Build Phase is Made Up of Plugin 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 plugin goals bound to those build phases.

A plugin goal represents a specific task (finer than a build phase) which contributes to the building and managing of a project. It may be bound to zero or more build phases. A goal not bound to any build phase could be executed outside of the build lifecycle by direct invocation. The order of execution depends on the order in which the goal(s) and the build phase(s) are invoked. For example, consider the command below. The `clean` and `package` arguments are build phases, while the `dependency:copy-dependencies` is a goal (of a plugin).

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

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

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: In Maven 2.0.5 and above, multiple goals bound to a phase are executed in the same order as they are declared in the POM, however multiple instances of the same plugin are not supported. Multiple instances of the same plugin are grouped to execute together and ordered in Maven 2.0.11 and above*).

[top] (./introduction-to-the-lifecycle.html).

Some Phases Are Not Usually Called From the Command Line

The phases named with hyphenated-words (`pre-*` , `post-*` , or `process-*`) are not usually directly called from the command line. These phases sequence the build, producing intermediate results that are not useful outside the build. In the case of invoking `integration-test` , the environment may be left in a hanging state.

Code coverage tools such as Jacoco and execution container plugins such as Tomcat, Cargo, and Docker bind goals to the `pre-integration-test` phase to prepare the integration test container environment. These plugins also bind goals to the `post-integration-test` phase to collect coverage statistics or decommission the integration test container.

Failsafe and code coverage plugins bind goals to `integration-test` and `verify` phases. The net result is test and coverage reports are available after the `verify` phase. If `integration-test` were to be called from the command line, no reports are generated. Worse is that the integration test container environment is left in a hanging state; the Tomcat webserver or Docker instance is left running, and Maven may not even terminate by itself.

[top] (./introduction-to-the-lifecycle.html).

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 via the equally named POM element `<packaging>` . Some of the valid packaging values are `jar` , `war` , `ear` and `pom` . If no packaging value has been specified, it will default to `jar` .

Each packaging contains a list of goals to bind to a particular phase. For example, the `jar` packaging will bind the following goals to build phases of 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 (/ref/current/maven-core/default-bindings.html); however, some packagings handle them differently. For example, a project that is purely metadata (packaging value is `pom`) only binds goals to the `install` and `deploy` phases (for a complete list of goal-to-build-phase bindings of some of the packaging types, refer to the Lifecycle Reference).

Note that for some packaging types to be available, you may also need to include a particular plugin in the `<build>` section of your POM and specify `<extensions>true</extensions>` for that plugin. One example of a plugin that requires this is the Plexus plugin, which provides a `plexus-application` and `plexus-service` packaging.

[top] (./introduction-to-the-lifecycle.html).

Plugins

The second way to add goals to phases is to configure plugins in your project. Plugins are artifacts that provide goals to Maven. Furthermore, a plugin may have one or more goals wherein each goal represents a capability of that plugin. For example, the 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 code.

As you will see in the later sections, plugins can contain information that indicates which lifecycle phase to bind a goal to. Note that adding the plugin on its own is not enough information - you must also specify the goals you want to 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 binds by default its goal `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>`:

```
...
 <plugin>
   <groupId>org.codehaus.modello</groupId>
   <artifactId>modello-maven-plugin</artifactId>
   <version>1.8.1</version>
   <executions>
     <execution>
       <configuration>
         <models>
           <model>src/main/mdo/maven.mdo</model>
         </models>
         <version>4.0.0</version>
       </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 inheritance 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:

```
...
 <plugin>
   <groupId>com.mycompany.example</groupId>
   <artifactId>display-maven-plugin</artifactId>
   <version>1.0</version>
   <executions>
     <execution>
       <phase>process-test-resources</phase>
       <goals>
         <goal>time</goal>
       </goals>
     </execution>
   </executions>
 </plugin>
...
```

[top] (./introduction-to-the-lifecycle.html).

Lifecycle Reference

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

Clean Lifecycle

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

Default Lifecycle

`validate`	validate the project is correct and all necessary information is available.
`initialize`	initialize build state, e.g. set properties or create directories.
`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
`process-test-classes`	post-process the generated files from test compilation, for example to do bytecode enhancement on Java classes. For Maven 2.0.5 and above.
`test`	run tests using a suitable unit testing framework. These tests should not require the code be packaged or deployed.
`prepare-package`	perform any operations necessary to prepare a package before the actual packaging. This often results in an unpacked, processed version of the package. (Maven 2.1 and above)
`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`	execute processes needed prior to the actual project site generation
`site`	generate the project's site documentation
`post-site`	execute processes needed to finalize the site generation, and to prepare for site deployment
`site-deploy`	deploy the generated site documentation to the specified web server

[top] (./introduction-to-the-lifecycle.html).

Built-in Lifecycle Bindings

Some phases have goals bound to them by default. And for the default lifecycle, these bindings depend on the packaging value. Here are some of the goal-to-build-phase bindings.

Clean Lifecycle Bindings

`clean`	`clean:clean`

Default Lifecycle Bindings - Packaging `ejb` / `ejb3` / `jar` / `par` / `rar` / `war`

process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
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 - Packaging `ear`

generate-resources	ear:generate-application-xml
process-resources	resources:resources
package	ear:ear
install	install:install
deploy	deploy:deploy

Default Lifecycle Bindings - Packaging `maven-plugin`

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

Default Lifecycle Bindings - Packaging `pom`

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

Site Lifecycle Bindings

site	site:site
site-deploy	site:deploy

References

The full Maven lifecycle is defined by the `components.xml` file in the `maven-core` module, with associated documentation (/ref/current/maven-core/lifecycles.html) for reference.

In Maven 2.x, default lifecycle bindings were included in `components.xml`, but in Maven 3.x, they are defined in a separate `default-bindings.xml (https://git-wip-us.apache.org/repos/asf?p=maven.git;a=blob;f=maven-core/src/main/resources/META-INF/plexus/default-bindings.xml)` descriptor.

See Lifecycles Reference (/ref/current/maven-core/lifecycles.html) and Plugin Bindings for default Lifecycle Reference (/ref/current/maven-core/default-bindings.html) for latest documentation taken directly from source code.

[top] (./introduction-to-the-lifecycle.html).
