

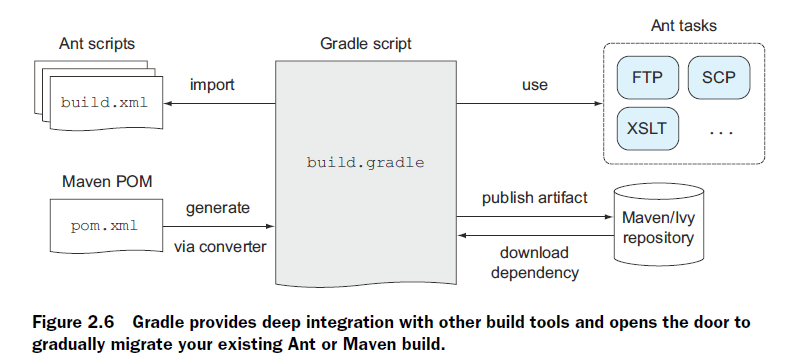
* Out-of-the-box Gradle provides you with two configuration blocks for your build script that allow you to define the dependencies and repositories that you want to retrieve them from. If the standard DSL elements don’t fit your needs, you can even introduce your own vocabulary through Gradle’s extension mechanism.
* A good place to start is the Gradle Build Language Reference Guide at http://www.gradle.org/docs/current/dsl/index.html.

Gradle supports incremental builds by specifying task inputs and outputs. It reliably figures out for you which tasks need to be skipped, built, or partially rebuilt. The same concept translates to multimodule projects, called partial builds. Because your build clearly defines the dependencies between submodules, Gradle takes care of rebuilding only the necessary parts. No more running clean by default!

Gradle supports parallel test execution.

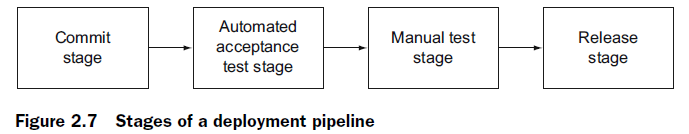
Gradle is going to support distributing test execution to multiple machines in a future version

Developers run builds many times during development. That means starting a new Gradle process each time, loading all its internal dependencies, and running the build logic. You’ll notice that it usually takes a couple of seconds before your script actually starts to execute. To improve the startup performance, Gradle can be run in daemon mode. In practice, the Gradle command forks a daemon process, which not only executes your build, but also keeps running in the background. Subsequent build invocations will piggyback on the existing daemon process to avoid the startup costs. As a result, you’ll notice a far snappier initial build execution.



For a deep dive on continuous delivery and all of its aspects, I recommend *Continuous Delivery: Reliable*

*Software Releases through Build, Test, and Deployment Automation* by Jez Humble and David Farley (Addison Wesley, 2010).



The concrete tasks we’re going to look at are

■ Compiling the code

■ Running unit and integration tests

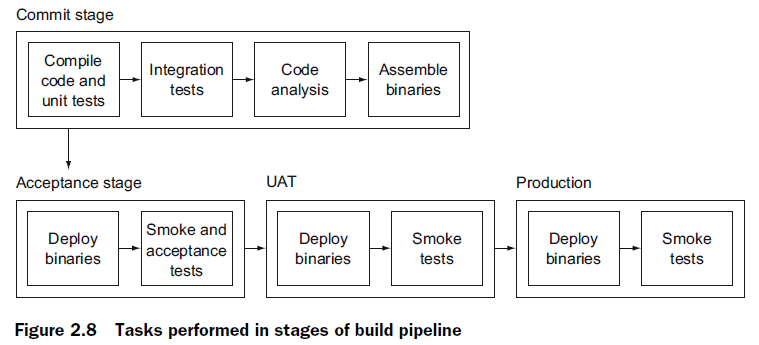
■ Performing static code analysis and generating test coverage

■ Creating the distribution

■ Provisioning the target environment

■ Deploying the deliverable

■ Performing smoke and automated functional tests



Like every other Java application, Gradle shares the same JVM options set by the environment

variable JAVA\_OPTS. If you want to pass arguments specifically to the Gradle

runtime, use the environment variable GRADLE\_OPTS. Let’s say you want to increase

the default maximum heap size to 1 GB. You could set it like this:

GRADLE\_OPTS="-Xmx1024m"

The preferred way to do that is to add the variable to the Gradle startup script under

$GRADLE\_HOME/bin.

Gradle provides a helper task named tasks to introspect your build script and display

each available task, including a descriptive message of its purpose

$ gradle -q tasks

Gradle tasks –all

Sometimes you want to exclude a specific task from your build run. Gradle provides

the command-line option –x to achieve that

The default naming convention for Gradle build script is build.gradle. Use this option to execute a build script with a different name (for example, gradle –b test.gradle).

*-D, --system-prop*: Gradle runs as a JVM process. As with all Java processes, you

can provide a system property like this: –Dmyprop=myvalue.

■ *-P, --project-prop*: Project properties are variables available in your build

script. You can use this option to pass a property to the build script directly

from the command line (for example, -Pmyprop=myvalue).

*tasks*: Displays all runnable tasks of your project including their descriptions.

Plugins applied to your project may provide additional tasks.

■ *properties*: Emits a list of all available properties in your project. Some of these

properties are provided by Gradle’s project object, the build’s internal representation.

Other properties are user-defined properties originating from a property

file or property command-line option, or directly declared in your build script.