Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Writing Build Scripts With Gradle

Even simple build scripts show the power and flexibility of the emerging new build tool for JVM projects.

July 15, 2014
URL:http://www.drdobbs.com/open-source/writing-build-scripts-with-gradle/240168648

In the first installment of this series on Gradle, I compared existing Java build tools with Gradle and described Gradle's compelling features. In this article, I show how to write and run Gradle's build scripts.

Installing Gradle

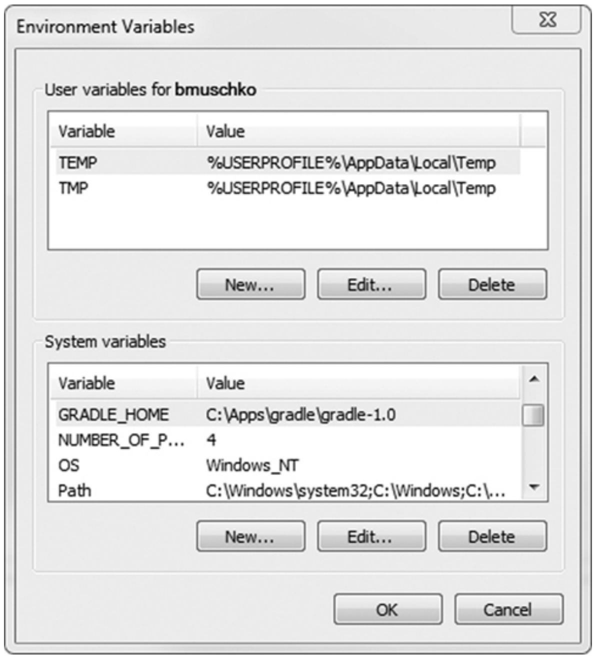
Before installing Gradle, make sure you've already installed the JDK with a version of 1.5 or higher. Even though some operating systems provide you with an out-of-the-box Java installation, make sure you have a valid version installed on your system. To check the JDK version, run `java -version`.

To get started with Gradle, download the distribution directly from the Gradle homepage. As a beginner to the tool, it makes sense to choose the ZIP file that includes the documentation and a wide range of source code examples to explore. Unzip the downloaded file to a directory of your choice. To reference your Gradle runtime in the shell, you'll need to create the environment variable `GRADLE_HOME` and add the binaries to your shell's execution path:

- Mac OS X and *nix: To make Gradle available in your shell, add the following two lines to your initialization script (for example, `~/.profile`). These instructions assume that you installed Gradle in the directory `/opt/gradle`:

  ```
  export GRADLE_HOME=/opt/gradle
  export PATH=$PATH:$GRADLE_HOME/bin
  ```

- Windows: In the dialog environment variable, define the variable `GRADLE_HOME` and update your path settings (Figure 1).


Environment Variables

User variables for bmuschko

Variable	Value
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

System variables

Variable	Value
GRADLE_HOME	C:\Apps\gradle\gradle-1.0
NUMBER_OF_P...	4
OS	Windows_NT
Path	C:\Windows\system32;C:\Windows;C:\...

Figure 1: Updating variable settings on Windows.

You'll verify that Gradle has been installed correctly and is ready to go. To check the version of the installed runtime, issue the command `gradle -v` in your shell. You should see meta-information about the installation, your JVM, and the operating system. The following example shows the version output of a successful Gradle 1.7 installation.

```
$ gradle -v
```

```
------------------------------------------------------------
Gradle 1.7
------------------------------------------------------------

Build time:   2013-08-06 11:19:56 UTC
Build number: none
Revision:     9a7199efaf72c620b33f9767874f0ebced135d83

Groovy:       1.8.6
Ant:          Apache Ant(TM) version 1.8.4 compiled on May 22 2012
Ivy:          2.2.0
JVM:          1.6.0_51 (Apple Inc. 20.51-b01-457)
OS:           Mac OS X 10.8.4 x86_64
```

Setting Gradle's JVM Options

Note that 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"
```

Although, the preferred way to do that is to add the variable to the Gradle startup script under `$GRADLE_HOME/bin`.

Now, let's implement a simple build script with Gradle. Even though most of the popular IDEs provide a Gradle plugin, all you need now is your favorite editor. Gradle plugin support exists for IDEs such as IntelliJ IDEA, Eclipse, and NetBeans.

Getting started with Gradle

Every Gradle build starts with a script. The default naming convention for a Gradle build script is `build.gradle`. When executing the command `gradle` in a shell, Gradle looks for a file with that exact name. If it can't be located, the runtime will display a help message.

Let's set the lofty goal of creating the typical "Hello world!" example in Gradle. First, create a file called `build.gradle`. Within that script, define a single atomic piece of work. In Gradle's vocabulary, this is called a task. In this example, the task is called `helloWorld`. To print the message “Hello world!” make use of Gradle's primary language, Groovy, by adding the `println` command to the task's action `doLast`. The method `println` is Groovy's shorter equivalent to Java's `System.out.println`:

```
task helloWorld {
   doLast {
      println 'Hello world!'
   }
}
```

Give it a spin:

```
$ gradle -q helloWorld
Hello world!
```

As expected, you see the output “Hello world!” when running the script. By defining the optional command-line option `quiet` with `-q`, you tell Gradle to only output the task's output.

Without knowing it, you already used Gradle's DSL. Tasks and actions are important elements of the language. An action named `doLast` is almost self-expressive. It's the last action that's executed for a task. Gradle allows for specifying the same logic in a more concise way. The left shift operator << is a shortcut for the action `doLast`. The following snippet shows a modified version of the first example:

```
task helloWorld << {
   println 'Hello world!'
}
```

Printing "Hello world!" only goes so far. I'll give you a taste of more-advanced features in the example build script shown in Listing One.

```
task startSession << {
    chant()
}
def chant() {
    ant.echo(message: 'Repeat after me...')
}
3.times {
    task "yayGradle$it" << {
        println 'Gradle rocks'
    }
}
yayGradle0.dependsOn startSession
yayGradle2.dependsOn yayGradle1, yayGradle0
task groupTherapy(dependsOn: yayGradle2)
```

Listing One: Dynamic task definition and task chaining.

You may not notice it at first, but there's a lot going on in this listing. You introduced the keyword `dependsOn` to indicate dependencies between tasks. Gradle makes sure that the depended-on task will always be executed before the task that defines the dependency. Under the hood, `dependsOn` is actually a method of a task.

A feature I talked about in the first installment in this series is Gradle's tight integration with Ant. Because you have full access to Groovy's language features, you can also print your message in a method named `chant()`. This method can easily be called from your task. Every script is equipped with a property called `ant` that grants direct access to Ant tasks. In this example, you printed out the message "Repeat after me" using the Ant task `echo` to start the therapy session.

A nifty feature Gradle provides is the definition of dynamic tasks, which specify their name at runtime. Your script creates three new tasks within a loop using Groovy's `times` method extension on `java.lang.Number`. Groovy automatically exposes an implicit variable named `it` to indicate the loop iteration index. You're using this counter to build the task name. For the first iteration, the task would be called `yayGradle0`.

Now running `gradle groupTherapy` results in the following output:

```
$ gradle groupTherapy
:startSession
[ant:echo] Repeat after me...
:yayGradle0
Gradle rocks
:yayGradle1
Gradle rocks
:yayGradle2
Gradle rocks
:groupTherapy
```

As shown in Figure 2 Gradle executed the tasks in the correct order. You may have noticed that the example omitted the `quiet` command-line option, which gives more information on the tasks run.

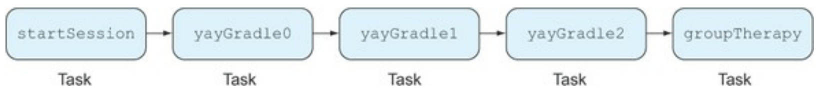


Figure 2: Task dependency graph.

Now, let's get more accustomed to Gradle's command line.

Using the Command Line

You just executed the tasks `helloWorld` and `groupTherapy` on the command line, which is going to be your tool of choice for learning. Even though using an IDE may seem more convenient to newcomers, a deep understanding of Gradle's command-line options and helper tasks will make you more efficient and productive in the long run.

Listing Available Tasks of a Project

In the previous section, I explained how to run a specific task using the `gradle` command. Running a task requires you to know the exact name. Wouldn't it be great if Gradle could tell you which tasks are available without you having to look at the source code? Gradle provides a helper task named `tasks` to introspect your build script and display each available task, including a descriptive message of its purpose. Running `gradle tasks` in quiet mode produces the following output:

```
$ gradle -q tasks
------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------
Build Setup tasks
-----------------
setupBuild - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]
Help tasks
----------
dependencies - Displays the dependencies of root project 'grouptherapy'.
dependencyInsight - Displays the insight into a specific dependency in root
 project 'grouptherapy'.
help - Displays a help message
projects - Displays the sub-projects of root project 'grouptherapy'.
properties - Displays the properties of root project 'grouptherapy'.
tasks - Displays the tasks runnable from root project 'grouptherapy' (some of
 the displayed tasks may belong to subprojects).
Other tasks
-----------
groupTherapy
To see all tasks and more detail, run with --all.
```

There are some things to note about the output. Gradle provides the concept of a task group, which can be seen as a cluster of tasks assigned to that group. Out of the box, each build script exposes the task group `Help tasks` without any additional work from the developer. If a task doesn't belong to a task group, it's displayed under `Other tasks`. This is where you find the task `groupTherapy`.

You may wonder what happened to the other tasks that you defined in your build script. On the bottom of the output, you'll find a note that you can get even more details about your project's tasks by using the `--all` option. Run it to get more information on them:

```
  $ gradle -q tasks --all
------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------
Build Setup tasks
-----------------
setupBuild - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]
Help tasks
----------
dependencies - Displays the dependencies of root project 'grouptherapy'.
help - Displays a help message
projects - Displays the sub-projects of root project 'grouptherapy'.
properties - Displays the properties of root project 'grouptherapy'.
tasks - Displays the tasks runnable from root project 'grouptherapy' (some of
 the displayed tasks may belong to subprojects).
Other tasks
-----------
groupTherapy
    startSession
    yayGradle0
    yayGradle1
    yayGradle2
```

The `--all` option is a great way to determine the execution order of a task graph before actually executing it. To reduce the noise, Gradle is smart enough to hide tasks that act as dependencies to a root task. For better readability, dependent tasks are displayed indented and ordered underneath the root task.

Task Execution

In the previous examples, we told Gradle to execute one specific task by adding it as an argument to the command `gradle`. Gradle's command-line implementation will in turn make sure that the task and all its dependencies are executed. You can also execute multiple tasks in a single build run by defining them as command-line parameters. Running `gradle yayGradle0 groupTherapy` would execute the task `yayGradle0` first and the task `groupTherapy` second.

Tasks are always executed just once, no matter whether they're specified on the command line or act as a dependency for another task. Let's see what the output looks like:

```
$ gradle yayGradle0 groupTherapy
:startSession
[ant:echo] Repeat after me...
:yayGradle0
Gradle rocks
:yayGradle1
Gradle rocks
:yayGradle2
Gradle rocks
:groupTherapy
```

No surprises here. You see the same output as if you'd just run `gradle groupTherapy`. The correct order was preserved and each of the tasks was only executed once.

Task Name Abbreviation

One of Gradle's productivity tools is the ability to abbreviate camel-cased task names on the command line. If you wanted to run the previous example in the abbreviated form, you'd just need to type `gradle yG0 gT`. This is especially useful if you're dealing with very long task names or multiple task arguments. Keep in mind that the task name abbreviation has to be unique to enable Gradle to identify the corresponding task. Consider the following scenario:

```
task groupTherapy << {
   ...
}

task generateTests << {
   ...
}
```

Using the abbreviation `gT` in a build that defines the tasks `groupTherapy` and `generate-Tests` causes Gradle to display an error:

```
$ gradle yG0 gT
```

```
FAILURE: Could not determine which tasks to execute.

* What went wrong:
Task 'gT' is ambiguous in root project 'grouptherapy'. Candidates are:
  'generateTests', 'groupTherapy'.

* Try:
Run gradle tasks to get a list of available tasks.

BUILD FAILED
```

Excluding a Task from Execution

Sometimes you want to exclude a specific task from your build run. Gradle provides the command-line option `-x` to achieve that. Let's say you want to exclude the task `yayGradle0`:

```
$ gradle groupTherapy -x yayGradle0
:yayGradle1
Gradle rocks
:yayGradle2
Gradle rocks
:groupTherapy
```

Gradle excluded the task `yayGradle0` and its dependent task `startSession`, a concept Gradle calls *smart exclusion*. Now that you're familiar with the command line, let's explore some other helpful functions.

Command-line Options

In this section, I explore the most important general-purpose options, flags to control your build script's logging level, and ways to provide properties to your project. The `gradle` command allows you to define one or more options at the same time. Let's say you want to change the log level to INFO using the `-i` option *and* print out any stack trace if an error occurs during execution with the option `-s`. To do so, execute the task `groupTherapy` command like this: `gradle groupTherapy–is` or `gradle groupTherapy -i -s`. As you can see, it's easy to combine multiple options. To discover the full set, run your build with the `–h` argument. I won't go over all the available options, but the most important ones are:

- `-?, -h, --help:` Prints out all available command-line options including a descriptive message.
- `-b, --build-file:` 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`).
- `--offline:` Often your build declares dependencies on libraries only available in repositories outside of your network. If these dependencies were not stored in your local cache yet, running a build without a network connection to these repositories would result in a failed build. Use this option to run your build in offline mode and only check the local dependency cache for dependencies.
- `-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`).
- `-i, --info:` In the default settings, a Gradle build doesn't output a lot of information. Use this option to get more informative messages by changing Gradle's logger to INFO log level. This is helpful if you want to get more information on what's happening under the hood.
- `-s, --stacktrace:` If you run into errors in your build, you'll want to know where they stem from. The option `–s` prints out an abbreviated stack trace if an exception is thrown, making it perfect for debugging broken builds.
- `-q, --quiet:` Reduces the log messages of a build run to error messages only.
- `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.

Gradle Daemon

When using Gradle on a day-to-day basis, you'll find yourself running your build repetitively. This is especially true if you're working on a Web application. You change a class, rebuild the Web application archive, bring up the server, and reload the URL in the browser to see your changes being reflected. Many developers prefer test-driven development to implement their application. For continuous feedback on their code quality, they run their unit tests over and over to find code defects early on. In both cases, you'll notice a significant productivity hit. Each time you initiate a build, the JVM has to be started, Gradle's dependencies have to be loaded into the class loader, and the project object model has to be constructed. This procedure usually takes a couple of seconds. The Gradle daemon can help here.

The daemon runs Gradle as a background process. Once started, the `gradle` command will reuse the forked daemon process for subsequent builds, avoiding the startup costs altogether. Let's go back to the previous build script example. On my machine, it takes about three seconds to successfully complete running the task `groupTherapy`. Let's try to improve the startup and execution time. It's easy to start the Gradle daemon on the command line: simply add the option `--daemon` to your `gradle` command. You may notice that we add a little extra time for starting up the daemon as well. To verify that the daemon process is running, you can check the process list on your operating system:

- Mac OS X and *nix: In a shell run the command `ps | grep gradle` to list the processes that contain the name gradle.
- Windows: Open the task manager with the keyboard shortcut Ctrl+Shift+Esc and click the Processes tab.

Subsequent invocations of the `gradle` command will now reuse the daemon process. Give it a shot and try running `gradle groupTherapy --daemon`. You'll find you got the startup and execution time down to about one second! Keep in mind that a daemon process will only be forked once even though you add the command-line option `--daemon`. The daemon process will automatically expire after a three-hour idle time. At any time you can choose to execute your build without using the daemon by adding the command-line option `--no-daemon`. To stop the daemon process, manually run `gradle --stop`. That's the Gradle daemon in a nutshell. For a deep dive into all configuration options and intricacies, refer to the Gradle online documentation.

Conclusion

Existing tools can't meet the build needs of today's industry. Improving on the best ideas of its competitors, Gradle provides a build-by-convention approach, reliable dependency management, and support for multi-project builds without having to sacrifice the flexibility and descriptiveness of your build.

In this series, we explored how Gradle can be used to deliver in each of the phases of a deployment pipeline in the context of continuous delivery. We ran some simple build scripts and found out how easy it is to define task dependencies using Gradle's DSL. Knowing the mechanics of Gradle's command line and its options is key to becoming highly productive. Gradle offers a wide variety of command-line switches for changing runtime behavior, passing properties to your project, and changing the logging level. As you explore Gradle further, I'm quite sure you'll conclude it's an important step forward for building Java projects. Enjoy!

Benjamin Muschko is the author of Gradle in Action*, published by Manning, from which this article is excerpted and adapted with permission. A special discount code has been set up for* Dr. Dobb's *readers for this book. Use* ***ddgrad*** *for a 42% discount off retail price (in all formats).*

Related Article

Why Build Your Java Projects with Gradle Rather than Ant or Maven?