Tutorial - Build Automation with Gradle

NOTE: This tutorial uses gradle 5.6 as the base version. Feel free to use higher versions.

Working with Build Tool: Gradle

As a project grows larger, there may be many commands that need to be run in order to compile, test, create jar files or other tasks. Running these commands manually after each change of the source code would quickly become both tedious and error prone. Instead, software projects use build tools that document and automate the commands to run. Some example of build tools are *Make*, *Ant*, *Ivy*, *Maven*, *Rake*, *Gant*, *Scons*, *SBT*, *Leinengen*, and *Buildr*.

In this tutorial, we will learn *Gradle*. It is a Java Virtual Machine (*JVM*) based build tool. It acknowledges and improves on the other build tools mentioned above. It expresses its build files in *Groovy*. *Groovy* is a dynamic language of the *JVM*, similar in many respects to Java, but with some important differences. Every *Gradle* build file is an executable *Groovy* script.

This tutorial is going to cover the following topics:

- *Gradle* core concepts
- *Gradle* installation
- Running different *Gradle* commands
- Building a java project with Gradle
- IDE for Gradle
- Gradle wrapper

Gradle core concepts

1. **Projects**: A project is typically some software you want to build.
2. **Tasks**: Tasks are actions that are required to build the project. A task could be compiling the source code, generating JavaDoc, zipping the compiled classes into a *JAR* file etc.
3. **Build Scripts**: A project typically has a build script in which its tasks are defined. The build script is used by the *gradle* command so it can know what tasks are defined for the project.

A Gradle project contains one or more tasks to execute in order to build the project. The build script is typically called *build.gradle* and is normally located in the root directory of the project you are building. When the *gradle* command is executed, the it looks for the build script in the directory where the command is executed from.

Gradle can do the following things:

- Automate common tasks like compiling, producing jar files, and running tests
- Support different languages by the use of plugins. There is, for example, a Java plugin. We will use this plugin in the later part of the tutorial.
- Incremental builds, i.e., avoid building things that are already up to date. For example, if you run a build, but did not change any source files since your last build, then no compilation or testing will be run.
- Automatic download of dependencies, i.e. appropriate versions of library files that your project depends on. These files are downloaded from global repositories like *Maven* and *Ivy*.

Install Gradle

Gradle runs on all major operating systems and requires only a Java JDK version 7 or higher to run. To check the java version in your machine, run the following command (for windows):

```
java -version
```

You should see something like this:

```
java version "1.8.0_181"
Java(TM) SE Runtime Environment (build 1.8.0_181-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.181-b13, mixed mode)
```

For Mac/linux:

```
java --version
```

You should see something like this:

```
java 9.0.4
Java(TM) SE Runtime Environment (build 9.0.4+11)
Java HotSpot(TM) 64-Bit Server VM (build 9.0.4+11, mixed mode)
```

To install Gradle in your own machine, follow these instructions.

Gradle Installation: find instructions here: **https://gradle.org/install/ (https://gradle.org/install/)**

Step 1. **Download** **(https://gradle.org/releases)** the latest Gradle distribution

Note that, the the distribution ZIP file comes in two flavors:

- Binary-only (bin)

- Complete (all) with docs and sources

Step 2. Unpack the distribution

Linux & MacOS users **(https://gradle.org/install/)**

Unzip the distribution zip file in the directory of your choosing, e.g.:

```
❯ mkdir /soft2412/gradle
❯ unzip -d /soft2412/gradle gradle-5.6-bin.zip
❯ ls /soft2412/gradle/gradle-5.6
LICENSE  NOTICE  bin  getting-started.html  init.d  lib  media
```


Microsoft Windows users

Create a new directory *C:\Gradle* with **File Explorer**.

Open a second **File Explorer** window and go to the directory where the Gradle distribution was downloaded. Double-click the ZIP archive to expose the content. Drag the content folder *gradle-5.6* to your newly created *C:\Gradle* folder.

Alternatively you can unpack the Gradle distribution ZIP into *C:\Gradle* using an archiver tool of your choice.

Step 3. Configure your system environment

For running Gradle, firstly add the environment variable *GRADLE_HOME*. This should point to the unpacked files from the Gradle website. Next add *GRADLE_HOME/bin* to your *PATH* environment variable. Usually, this is sufficient to run Gradle.

Linux & MacOS users

Configure your *PATH* environment variable to include the *bin* directory of the unzipped distribution, e.g.:

```
export PATH=$PATH:/soft2412/gradle/gradle-5.6/bin
```

Microsoft Windows users

In **File Explorer** right-click on the *This PC* (or *Computer*) icon, then click *Properties* → *Advanced System Settings* → *Environmental Variables*.

Under *System Variables* select *Path*, then click *Edit*. Add an entry for *C:\Gradle\gradle-5.6\bin*. Click OK to save.

Verifying installation

To test if Gradle is properly installed, open a command prompt and execute this command:

```
gradle -version
```

If *Gradle* is correctly installed, you should see an output similar to this:

```
Gradle 5.6
------------------------------------------------------------

Build time:   2019-07-16 08:14:03 UTC
Revision:     3245f748c70614f2dd4dc1849919198107f935a5

Kotlin DSL:   0.18.4
Kotlin:       1.3.31
Groovy:       2.5.4
Ant:          Apache Ant(TM) version 1.9.11 compiled on March 12 2019
JVM:          1.8.0_181 (Oracle Corporation 25.181-b13)
OS:           Windows 10 10.0 amd64
```


Reference commands to run Gradle

To run *Gradle* from the command line you must first have installed it correctly. When installed correctly you can run *Gradle* using this command line:

```
gradle
```

To run this command in a specific build script, you must be in the directory where the script is located.

Run a Task

The Gradle build script typically contains one or more tasks which you can execute. You execute a task in a build script by listing the task name after the gradle command on the command line. Here is how to run a Gradle task:

```
gradle <task>
```

This command runs the task named <task> in the build script located in the same directory as the command is executed from.

Run Multiple Tasks

Gradle can run multiple tasks with a single command by providing the names of all the tasks to run. The names of the tasks to run should be separated by a space. Here is an example of a *Gradle* command running multiple tasks:

```
gradle clean build
```

This command line will make *Gradle* first execute the task named *clean* and then the task named *build*. *Gradle* will also execute the tasks the named tasks depends on (if any).

Each task will only get executed once, regardless of how many times it is found as part of the listed tasks or the tasks the listed tasks depend on. Thus, this command will only execute the *clean* task once:

```
gradle clean clean
```


Task Name Abbreviation

You don't actually have to write the full task name of a task in order to execute it. *Gradle* just need enough of the task name to be able to distinguish it uniquely from other tasks in your *Gradle* build script. For example you can run the *build* task using this command:

```
gradle b
```

A *b* is enough to uniquely identify the task *build* in a Gradle build script if no other tasks start with the letter *b*. If other tasks start with a *b* too, you may have to provide more of the task name. These *Gradle* command examples show how that could look:

```
gradle bu
gradle bui
gradle buil
```

Excluding Tasks

You can exclude tasks from execution using the *-x* flag. Excluding tasks is mostly useful to exclude a task which another task depends on, typically to speed up the build process by excluding tasks you know you don't need to execute.

For example, the *build* task from the *Gradle* Java plugin depends on the tasks *test* and *testClasses* . Perhaps you have just executed the tests and know that they succeed. Therefore

you do not want to run the *test* and *testClasses* tasks again, but only build the project. You can do so with this *Gradle* command:

```
gradle build -x test
```

This will exclude the *test* task from execution. Since the *test* task depends on the *testClasses* task and the *test* task is now excluded, the *testClasses* is also implicitly excluded.

Quiet Mode

You can run *Gradle* in a "quiet mode" which means that the *Gradle* command leaves out a lot of the status messages it normally prints to the command line when executing. To run *Gradle* in quiet mode you add the command line switch *-q* to the command line. Here is an example showing how to run *Gradle* in quiet mode:


```
gradle -q
```

Or if you want to run a specific task:


```
gradle -q <task>
```


Listing Tasks in Build Script

You can list all tasks in a build script by passing the command line argument *tasks* to the *gradle* command, like this:


```
gradle tasks
```

This will list all the tasks found in the build script located in the directory in which you run this command.

If you use the *--all* flag you will get more information about each task. Here is an command line example:

```
gradle tasks --all
```

Specifying Build Script

By default, Gradle executes the build script located in the same directory in which the *gradle* command is executed. However, it is possible to choose to build another build script.

You specify another build script using the *-b* command line flag. Here is an example:

```
gradle -b subproject-dir/build.gradle build
```

As an alternative to specifying a different build script than the one found at the default location, you can also specify another project to build. A project is identified by the directory in which the project is located.

You run the build script of another project by using the *-p* flag followed by the directory of the project. Here is an example of specifying what project to build:

```
gradle -p subproject-dir build
```

This example will build the project found in the directory *subproject-dir*. Gradle will use the *build.gradle* file found in the project directory (*subproject-dir*) and will execute the *build* task in that build script.

Listing Subprojects

Gradle can list all subprojects of a project (as specified inside the Gradle build script for the project). You do so using the *projects* task. Here is how that looks:

```
gradle projects
```


Gradle Help

You can get a list of the *gradle* command's options by passing the *-h* flag to the *gradle* command, like this:

```
gradle -h
```

Build Failures

If a task in the build script fails during execution, *Gradle* will abort the whole build. This is done to save you time. Often, later tasks in the build script make no sense to execute if earlier tasks fail.

You can instruct *Gradle* to continue the build even if a task fails. This is done using the *--continue* flag, like this:

```
gradle build --continue
```

Gradle will continue executing all tasks where the tasks it depends on were executed successfully. Thus, if task B depends on task A and A fails, then Gradle will actually not execute B, even if you instruct Gradle to continue the build despite errors.

Dry Run

A "dry run" is a run where *Gradle* does not actually execute all the tasks you tell it to execute. Instead *Gradle* prints out information about what tasks that would have been executed in case you had run Gradle normally.

You signal to Gradle to make a dry run using the *-m* flag. Here is a *Gradle* dry run command example:

```
gradle -m build
```

The output from a dry run will look somewhat similar to this:

D:\data\projects\gradle-experiments>gradle -m build :compileJava SKIPPED :processResources SKIPPED :classes SKIPPED :jar SKIPPED :assemble SKIPPED :compileTestJava SKIPPED :processTestResources SKIPPED :testClasses SKIPPED :test SKIPPED :check SKIPPED :build SKIPPED BUILD SUCCESSFUL Total time: 14.869 secs D:\data\projects\gradle-experiments>

Verbose output

More verbose output of a gradle task execution can be obtained by typing the following command:


```
gradle <task> --console verbose
```

Building Java Projects with Gradle

In this part of the tutorial, you are going to build a java project with Gradle.

Create the Directory Structure

- Create a project directory named *demo*. You can run the following commands for Linux and Mac:

```
mkdir demo
cd demo
```

Building the Java Project

To build the project follow these steps:

- *cd demo* and run *gradle init*
- Choose "application", "java", "Groovy" and "JUnit Jupyter"
- Now check the newly created files and folder after the *init* task had run. You will see a build.gradle file (the build.script)

- Run the command *gradle build*.

  Check out the directory structure after the build command had run. You will see the following directory structure under build folder.

  - classes. The project's compiled .class files.
  - libs. Assembled project libraries (usually JAR and/or WAR files)

- Now run the *gradle run* command. You will see the following:

```
> Task :run
Hello world.
```


To Do:

- List the tasks of your project with the command you have learned. Read through the task explanations to have some idea about the different tasks that Java plugin has added to your project.

Exploring dependencies

Usually, a complex Java project will depend on some external JAR files. To reference these JAR files in the project, you need to define them in the *build.gradle* file. In Gradle, artifacts such as JAR files, are located in a repository. A repository can be used for fetching the dependencies of a project, or for publishing the artifacts of a project, or both. Here is an example for adding public Maven repository:

build.gradle

```
repositories {
    mavenCentral()
}
```

You can also add dependencies in your project as:

build.gradle

```
dependencies {
    compile group: 'commons-collections', name: 'commons-collections', version: '3.2.2'
    testCompile group: 'org.junit.jupiter', name: 'junit-jupiter-api', version: '5.5.1'
}
```

The above code declares that the production classes have a compile-time dependency on commons collections, and the test classes have a compile-time dependency on JUnit. Note down the syntax for adding the dependencies.

Basically, commons-collections is a repository that is a package that extends and augments the java collections framework. Find the documentation of commons collections here: **https://commons.apache.org/proper/commons-collections/javadocs/api-3.2.2/ (https://commons.apache.org/proper/commons-collections/javadocs/api-3.2.2/)** . You can find further repositories here: **https://mvnrepository.com/ (https://mvnrepository.com/)** including JUnit which you will be using next week for JUnit testing.

Now, let's add a new file called HelloWorld.java to use Google Guava's Joiner to join strings.

HelloWorld.java

```
import com.google.common.base.Joiner;
import com.google.common.collect.Lists;
import java.util.List;

public class HelloWorld {

	public static void main(String[] args) {

	List newList = Lists.newArrayList("Hello", "World");
	String result = Joiner.on(",").join(newList);
	System.out.println(result);

	  }
}
```


Make sure you change the mainClassName attribute in the build.gradle script to be "HelloWorld" so that gradle knows that the main class it should run would be "HelloWorld".

Also check if these lines exist in the dependencies block of the build.gradle script:

```
// This dependency is used by the application.
implementation 'com.google.guava:guava:29.0-jre'
```

Note that we have added two new packages from Google Guava. Now, if you try to build the project, using *gradle build* command, and you should see "Hello,World" as the output.

Tasks To Do:

- Run *gradle properties* to list the properties of a project.

IDE for Gradle

We spent some time in the above sections looking at gradle commands. However, its always great to use an IDE for java projects where running gradle is easier i.e. no command line :)

You can use *Eclipse/IntelliJ* as an *IDE* for your *Gradle* project.

- To use *Gradle* in *Eclipse*, you need to download Buildship tooling first. You can download Buildship toolong via Eclipse update manager or Eclipse Market place.
- For IntelliJ, you will have to download the IDE here:
https://www.jetbrains.com/idea/download/ (https://www.jetbrains.com/idea/download/)

Tasks To Do:

- After you finish setting up Gradle in Eclipse/IntelliJ, import the project you have created to the IDE.

Gradle wrapper

So far, we have understood how to install gradle and use it. Gradle wrapper helps automate the installation of a gradle version used by a project. The Wrapper is a script that invokes a declared version of Gradle, downloading it beforehand if necessary. As a result, developers can get up and running with a Gradle project quickly without having to follow manual installation processes saving the company time and money.

Another advantage of using gradle wrapper is that it standardises a project on a given Gradle version, leading to more reliable and robust builds.Different users can be provided with a different version of gradle, which is as simple as changing the wrapper definition.

First you need to generate the necessary gradle wrapper files in order to start making use of it. Run this command to generate these files.

```
gradle wrapper
```

Now, here's how you use it: Modify the following line in your gradle/wrapper/gradle-wrapper.properties file as follows (use an older version that is compatible with all collaborator's workstations, say 4.8):

```
distributionUrl=https://services.gradle.org/distributions/gradle-4.8-bin.zip
```

This way you have defined from where to install the gradle package. The location of installation is specified in "distributionPath" and "distributionBase".

This can also be automatically done without changing the above line by typing:

```
./gradlew wrapper --gradle-version 4.8
```

Now, you must use the “gradlew” program in the current folder to build your tasks under this new specified version. The version may be changed to another one, not necessarily 4.8

For more information on gradle wrapper, please refer to this link: **https://docs.gradle.org/current/userguide/gradle_wrapper.html (https://docs.gradle.org/current/userguide/gradle_wrapper.html)**

References:

1. Building and Testing with Gradle: Tim Berglund and Matthew McCullough, O'Reilly

2. Some of the contents adapted from Lund University

3. **http://tutorials.jenkov.com/gradle/gradle-tutorial.html (http://tutorials.jenkov.com/gradle/gradle-tutorial.html)**

4. **https://docs.gradle.org/current/userguide/tutorial_java_projects.html (https://docs.gradle.org/current/userguide/tutorial_java_projects.html/)**

5. **https://docs.gradle.org/current/userguide/installation.html (https://docs.gradle.org/current/userguide/installation.html)**

6. **https://docs.gradle.org/current/userguide/gradle_wrapper.html (https://docs.gradle.org/current/userguide/gradle_wrapper.html)**