Chapter 9. The task graph

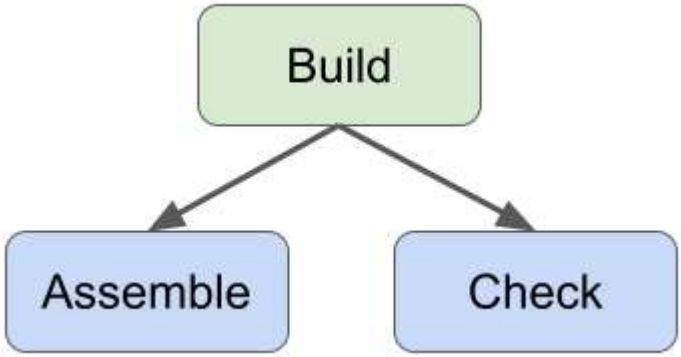
In this chapter, you'll learn in more detail what the Gradle task graph is, what it looks like in a Java project, and how you can make the best use of it in your day-to-day work.

9.1. Task graph essentials

We already discussed in *Introduction to Gradle* how tasks can have dependencies on other tasks. A dependency on another task means that the other task gets run first.

NOTE	Task dependencies are a completely separate concept to the dependencies you setup for compiling your code, such as JUnit.

Let's use as an example some of the tasks added by the Java plugin. The *build* task depends on the *assemble* and *check* tasks.



This means that when you run `./gradlew build`, *assemble* and *check* get run first. We haven't used any of these tasks yet in this book, but don't worry because in a moment you'll see them in action.

The interesting thing is that the *build* task itself doesn't do anything when it's run It's only there to *aggregate* both the *assemble* and *check* tasks together.

- **assemble** handles building your project including creating artifacts such as jar files.
- **check** handles testing your project.

The *build* task, using task dependencies, makes sure that both *assemble* and *check* get run. That's really nice, as it means if you ever need to build and test a whole project, you can just run a single task, *build*, to do everything.

9.2. Java plugin task graph

That was just 3 of the tasks added by the Java plugin. There are lots more, so let's look at the complete task graph.

Yes, there's quite a lot to look at here. Don't worry as we're going to step through this bit-by-bit

The lines that connect a task on the left to a task to the right imply a dependency on the task to right. so for example, *build* depends on *check*.

You already have practical experience of some of these tasks, including *jar*, *compileJava*, *processResources*, and *test*. These tasks are highlighted in green, showing tasks which actually perform an action when they're run.

For example:

- **jar** creates a jar file.
- **compileJava** compiles classes.
- **processResources** well…processes resources.
- **test** runs tests.

Let's take a look at the *classes* task, highlighted in blue. This task doesn't perform any action itself when it's run, it merely aggregates *compileJava* and *processResources* using task dependencies. Blue indicates aggregate tasks

Moving up the tree, *assemble* is another aggregate task, which when run causes the jar task to run, which in turn runs classes, which runs *compileJava* and *processResources*. You get the idea.

Test tasks

Let's focus now on the tasks under the *test* task. You'll notice we have the same *compileJava*, *processResources*, and *classes* tasks. This ensures that when we run the test task, our main application code has been properly compiled and resources processed

These 3 tasks are duplicated again under *compileTestJava*. *compileTestJava* itself is the task which compiles all your test code and similarly *processTestResources* processes all your test resources.

Above all these we have:

- *testClasses*, another aggregate task which depends on *compileTestJava* and *processTestResources*.
- *test*, which ensures that both the main code and test code are built, before running the tests themselves.
- *check*, another aggregate task which just depends on *test*.

Finally, at the very top level is *build*, an aggregate task which builds and tests everything in your project.

Thankfully you don't need remember all this information in its entirety.

Important tasks

Highlighted with a star are five key Gradle tasks to remember which will help when using Gradle in your day-to-day work.

- **build** is used when you just need to build and test the whole project.
- **assemble** and **jar** you can think of as equivalent, and are used when you only need to assemble the project without testing it.
- **check** and **test** you can think of as equivalent, and are used to test your code without assembling it.

But why you wouldn't just run *build* all the time? Well, medium to large projects can take a while to build, so picking the specific Gradle task for what you need to accomplish can save time.

For example, if you just need to test your code change then you can run *check* rather than *build*. This could be quicker, since *check* doesn't depend on the *jar* task, so no *jar* is created.

Conversely, if you just need to *build* your project into a *jar* file, without testing your changes, you could run the *assemble* task.

Let's now jump back into the project as we left it in the *Building a Java project with Gradle* practical chapter, to demonstrate a few of these important Gradle tasks.

9.3. Practical demonstration

Before we get started, we need to run a task which we haven't talked about yet called *clean*.

This simple task deletes the *build* directory. Normally you don't run this very often, but we'll run it now to ensure we start in a fresh state and can see how each of the other tasks changes the directory structure.

So let's run `./gradlew clean`.

```
$ ./gradlew clean

BUILD SUCCESSFUL in 1s
```

```
1 actionable task: 1 executed
```

You can see the *build* directory was deleted

```
$ ls build
ls: cannot access 'build': No such file or directory
```

assemble task

The first important task to run is *assemble*, which compiles your code, processes resources, and builds a jar file.

So let's run `./gradlew assemble`.

```
$ ./gradlew assemble

BUILD SUCCESSFUL in 2s
3 actionable tasks: 3 executed
```

Let's take a look in the newly generated *build* directory.

```
build
|-- classes
|   `-- java
|       `-- main ①
|           `-- com
|               `-- gradlehero
|                   `-- languageapp
|                       `-- SayHello.class
|-- generated
|   `-- sources
|       |-- annotationProcessor
|       |   `-- java
|       |       `-- main
|       `-- headers
|           `-- java
|               `-- main
|-- libs
|   `-- get-going-with-gradle-practical.jar ③
|-- resources
|   `-- main ②
|       |-- en.txt
|       `-- es.txt
`-- tmp
    |-- compileJava
    |   `-- previous-compilation-data.bin
    `-- jar
```

```
        `-- MANIFEST.MF
```

① Compiled classes are in the *classes/java/main* directory.

② Resources are in the *resources/main* directory.

③ A *jar* file was generated in the libs directory.

So *assemble* is great for generating all your build artifacts in one go. Notice that we don't have any test reports generated as *assemble* doesn't run the tests.

Let's delete the *build* directory again by running `./gradlew clean`.

```
$ ./gradlew clean

BUILD SUCCESSFUL in 1s
1 actionable task: 1 executed
```

check task

The next important task to run is *check*. This task compiles the main code, processes the main resources, compiles test code, processes test resources, and actually runs the tests too

Let's run `./gradlew check`.

```
$ ./gradlew check

BUILD SUCCESSFUL in 2s
4 actionable tasks: 4 executed
```

Let's take a look in the *build* directory.

```
build
|-- classes
|   `-- java
|       |-- main ①
|       |   `-- com
|       |       `-- gradlehero
|       |           `-- languageapp
|       |               `-- SayHello.class
|       `-- test ①
|           `-- com
|               `-- gradlehero
|                   `-- languageapp
|                       `-- SayHelloTest.class
|-- generated
|   `-- sources
|       |-- annotationProcessor
|       |   `-- java
```

```
|       |       |-- main
|       |       `-- test
|       `-- headers
|           `-- java
|               |-- main
|               `-- test
|-- reports
|   `-- tests
|       `-- test ③
|           |-- classes
|           |   `-- com.gradlehero.languageapp.SayHelloTest.html
|           |-- css
|           |   |-- base-style.css
|           |   `-- style.css
|           |-- index.html
|           |-- js
|           |   `-- report.js
|           `-- packages
|               `-- com.gradlehero.languageapp.html
|-- resources
|   `-- main ②
|       |-- en.txt
|       `-- es.txt
|-- test-results
|   `-- test
|       |-- TEST-com.gradlehero.languageapp.SayHelloTest.xml
|       `-- binary
|           |-- output.bin
|           |-- output.bin.idx
|           `-- results.bin
`-- tmp
    |-- compileJava
    |   `-- previous-compilation-data.bin
    |-- compileTestJava
    |   `-- previous-compilation-data.bin
    `-- test
```

① We have both *classes/java/main* and *classes/java/test* because Gradle has compiled main and test classes.

② In the *resources* directory are only main resources, but if we had test resources they'd appear here too.

③ There's a reports directory containing a test report, since our tests were also executed.

Note that we don't have a *libs* directory or jar file, as the *check* task doesn't depend on *jar*.

So *check* is the task to run when you just need to run the tests without assembling everything in your project.

Let's delete the build directory once more with `./gradlew clean`.

```
$ ./gradlew clean

BUILD SUCCESSFUL in 1s
1 actionable task: 1 executed
```

build task

The final important task to try is *build*, which depends on the two tasks we've just run, *assemble* and *check*.

Let's run `./gradlew build`.

```
$ ./gradlew build

BUILD SUCCESSFUL in 2s
5 actionable tasks: 5 executed
```

Let's have a look in the *build* directory.

```
build
|-- classes
|   `-- java
|       |-- main ①
|       |   `-- com
|       |       `-- gradlehero
|       |           `-- languageapp
|       |               `-- SayHello.class
|       `-- test ①
|           `-- com
|               `-- gradlehero
|                   `-- languageapp
|                       `-- SayHelloTest.class
|-- generated
|   `-- sources
|       |-- annotationProcessor
|       |   `-- java
|       |       |-- main
|       |       `-- test
|       `-- headers
|           `-- java
|               |-- main
|               `-- test
|-- libs
|   `-- get-going-with-gradle-practical.jar ③
|-- reports
|   `-- tests
|       `-- test ④
|           |-- classes
```

```
|           |   `-- com.gradlehero.languageapp.SayHelloTest.html
|           |-- css
|           |   |-- base-style.css
|           |   `-- style.css
|           |-- index.html
|           |-- js
|           |   `-- report.js
|           `-- packages
|               `-- com.gradlehero.languageapp.html
|-- resources
|   `-- main ②
|       |-- en.txt
|       `-- es.txt
|-- test-results
|   `-- test
|       |-- TEST-com.gradlehero.languageapp.SayHelloTest.xml
|       `-- binary
|           |-- output.bin
|           |-- output.bin.idx
|           `-- results.bin
`-- tmp
    |-- compileJava
    |   `-- previous-compilation-data.bin
    |-- compileTestJava
    |   `-- previous-compilation-data.bin
    |-- jar
    |   `-- MANIFEST.MF
    `-- test
```

① We have compiled main and test code.

② We have the processed resources.

③ We have a jar file in the libs directory.

④ We have test reports showing that the tests ran.

So *build* is the task to run when you just need to assemble your whole project and run all the tests.

NOTE	The *build* directory also contains *generated*, *test-results*, and *tmp* directories, used internally by Gradle. You can ignore these directories.

9.4. Chapter summary

There was a lot of information in the last few pages, so here's a summary of the key takeaways.

1. Some tasks actually perform an action and some are just aggregate tasks.
2. Aggregate tasks combine multiple tasks together for convenience.
3. *assemble* and *jar* both compile code and build a jar file.
4. *check* and *test* both test your code.

5. *build* does both an *assemble* and a *check*.

You now have a clear understanding of the different tasks used by Gradle in Java projects and how they relate to each other.