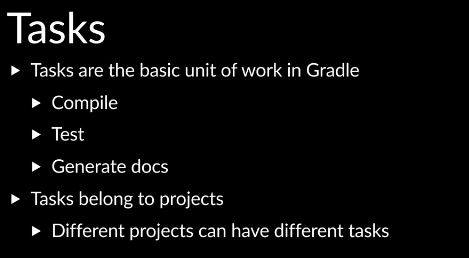
https://dpeuniversity.gradle.com/courses/012de84f-fcd3-45d4-9c4c-284382eb3f3f/activities/65b23cd1-fe72-4f0d-95d2-57d88eed40f2

Tasks overview – 10:36 minute

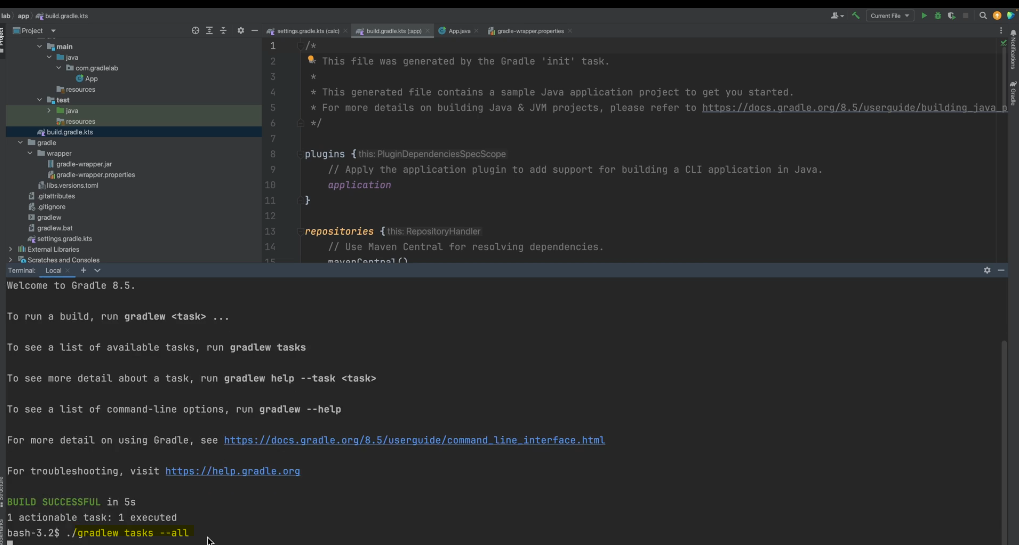
Let’s dive into tasks and explore how they are structured, how to execute them view the output. We will cover the following:

* The structure of tasks
  + have inputs
  + execute an action
  + can generate outputs
* View all available tasks
* Executing tasks
* Using outcome labels

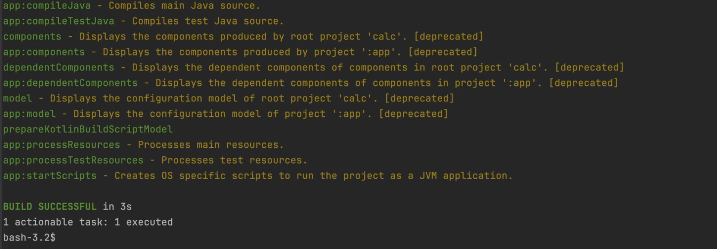
Tasks are the basic unit of work in Gradle Build Tool. Examples include those to compile and test your project. It’s important to understand that tasks belong to projects or sub projects and each can have different tasks. There may be different plugins applied, hence they would have different tasks. Let’s run a few tasks.



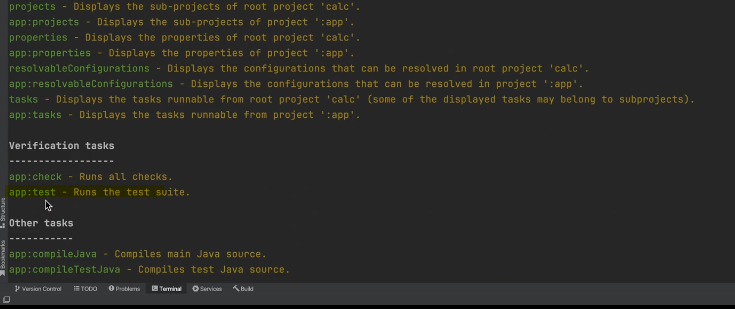
One of the first tasks you would often want to run on a project is tasks, which displays the available tasks.



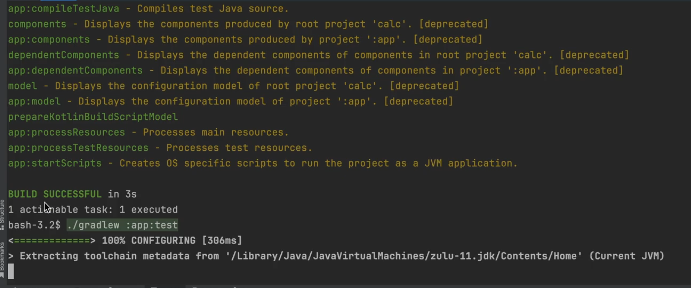
We want to pass the dash dash ( - - ) all argument to display all available tasks, otherwise Gradle build tool will display only a subset of tasks.



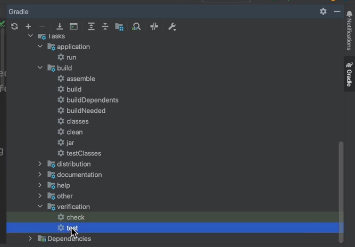
You can see it displays tasks available on the app subproject, which prefixes the name of the subproject followed by a colon. As well as tasks which can be run on the root project, which have no prefix, we can see there is a task to run tests on the app subproject. Let’s run this task.



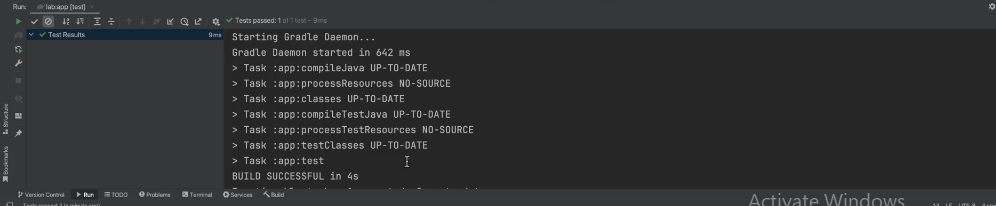
We can specify the name of the subproject we want to run the task on, and then the task.



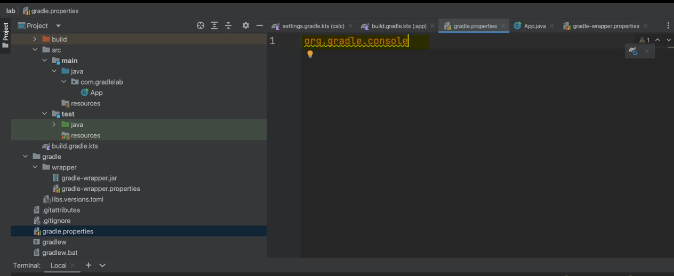
We can see the test ran successfully. In many editors, there’ll be a Gradle toolbar. In IntelliJ, it can usually be found on the right. Here you can see all the available tasks that can be run on the root project, as well as those that can be run on the app subproject.



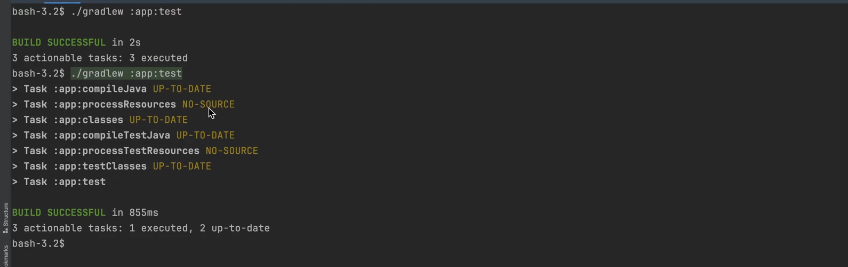
We can also run the tasks from the toolbar. Notice here the output contains more information.



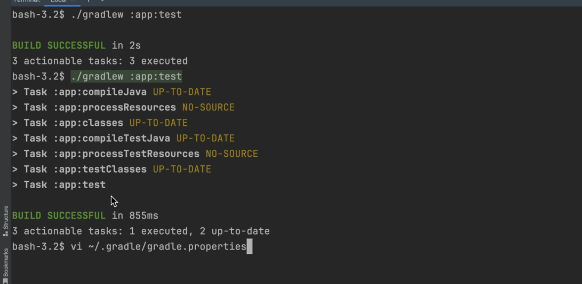
It displays the prerequisites tasks that had to run before the test task could run. Whereas in our terminal, this information was not displayed. We can add a property so it shows up in the terminal as well, so those who prefer using the terminal can get the same level of output detail. We can create a file on the top level directory called Gradle.properties. And in here we can add the property org.gradle.console and set it to the value verbose.



Now when we run the test task on the app subproject again,

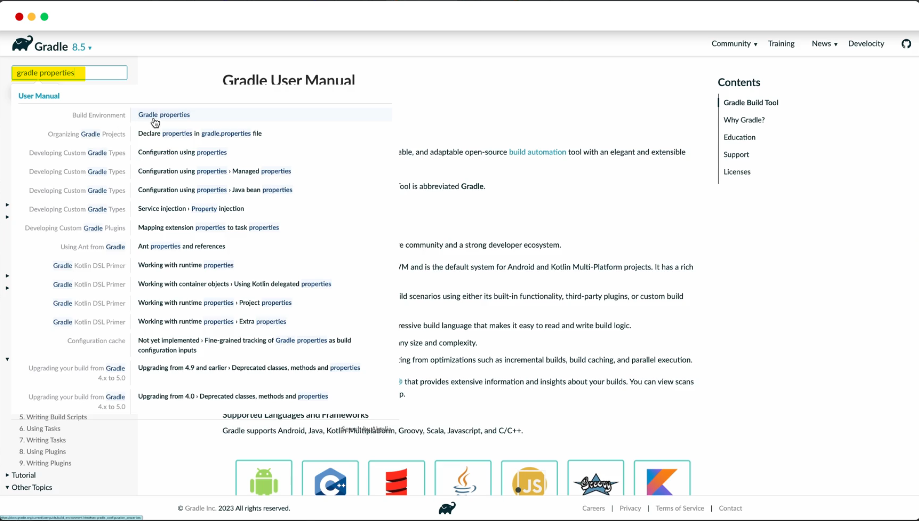


we can see the additional information. Alternatively, you can create a gradle.properties file in the Gradle home folder and put the logging property in there.

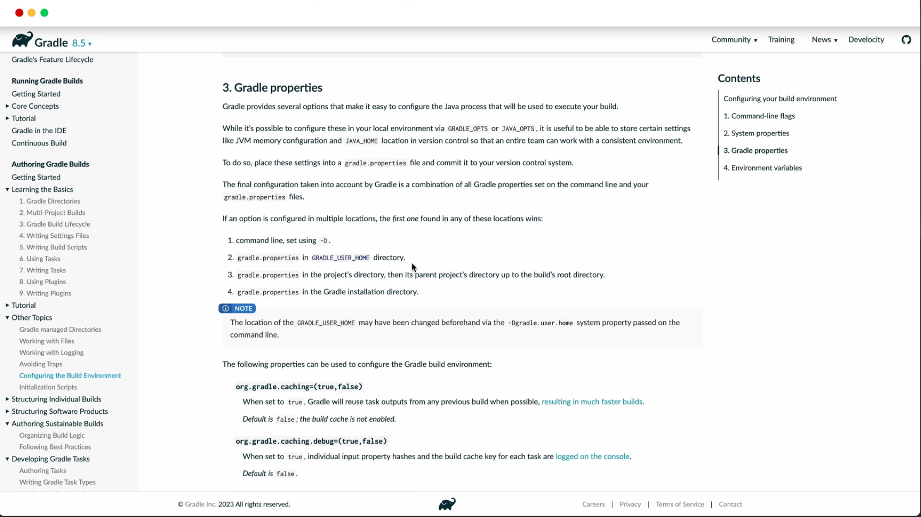


This will then apply the verbose logging to all projects you’re working on. Till you’re more familiar with the tasks and their dependencies, it may be helpful to use this verbose logging.

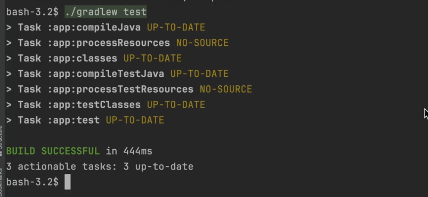
In the Gradle build documentation, you can search for Gradle properties



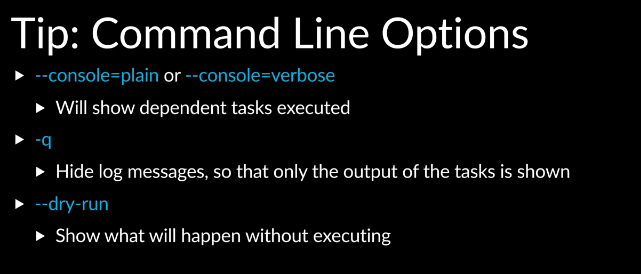
and click on the result build environment. Here you can learn more about how to use Gradle properties to configure options for the build. There are a number of places where properties can be defined. The documentation goes over these and lists the order of precedents, if properties are defined in multiple locations. Properties passed through the command line precedents. Then those defined in the Gradle properties file in the Gradle home directory.



Then those defined in the Gradle properties file in a subproject directory. Then those defined in the Gradle properties file in the root project directory. As you can see, the Gradle properties file can be added in multiple locations. Most of the time, you’ll have one in the root project directory for project specific options that everyone working on the project should have. And then you can have your own options defined in the Gradle properties in the Gradle home folder that will take precedence over the root project directory properties. You can also reference the documentation on other properties that can be defined. We can also run the test task on the root project.



When I do this, Gradle build tool will run the test task on all subprojects that have the task. Most tasks behave this way when we run on the root project. However, not all tasks do. Some tasks when run on the root project will only run on the root project. Therefore, it is good to get into the habit of being specific with regards to which subproject you want to run a task on. There are command line options that will affect either the logging or behavior or running tasks. The dry run argument will display all the tasks that would run without actually running them. This may be useful to look at before running the task.



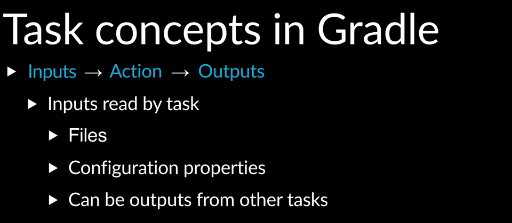
Conceptually; it’s important to understand that tasks have inputs, they may execute an action, and they produce outputs.



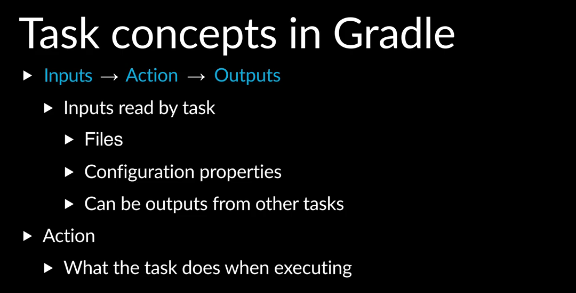
Inputs, action, outputs. Inputs can be files such as source code.



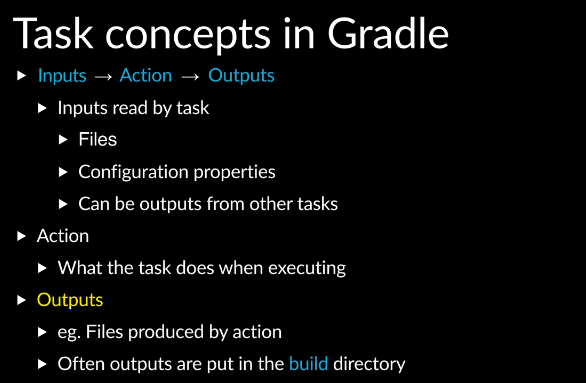
However, inputs can also be other things such as configuration options.



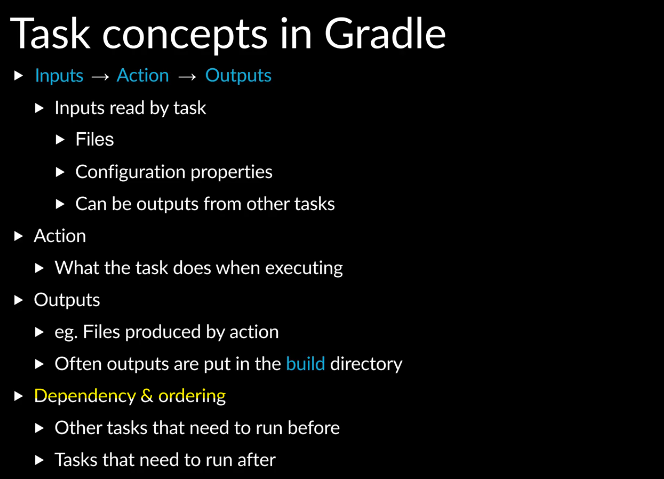
Also, the output of one task can be the input to another task.



The action is what a task does when executing.

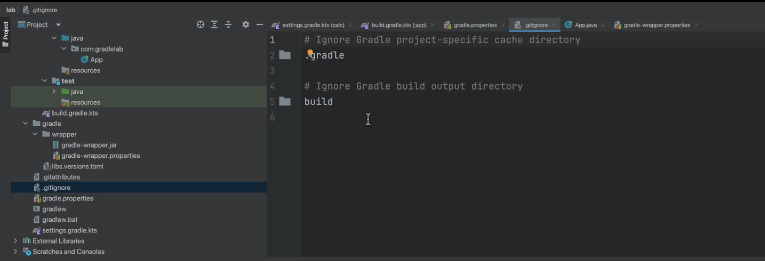


Output files produced are typically put in the build directory under the top level directory.

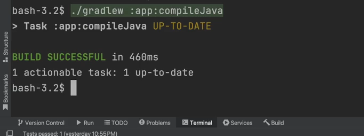


And as we have already seen, tasks can have dependencies on other tasks and ordering. We can see the build directory where the outputs of tasks are.

For example, the output of the compiled task can be found class.java. We can see the output of the test task as well as reports created by the test task. The outputs created by tasks should not be checked into version control.



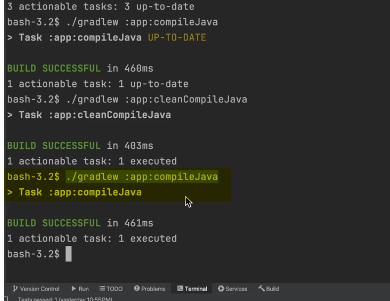
Gradle init created a gitignore file and put the build directory as something that should be excluded from checking into version control. In the terminal output, we can see string labels next to the tasks. These are called the outcome labels, and they tell us what happened during the action execution. If there is no outcome label, that means the task action executed successfully. If there is an up to date outcome label, Gradle build tool is telling us that it noticed the output for the task was already in the build directory, and none of the inputs changed since the outputs were created. So, Gradle build tool reused the existing outputs and skipped executing the action. The task is up to date, and the action does not need to be executed. If we run the compileJava task on the app subproject,



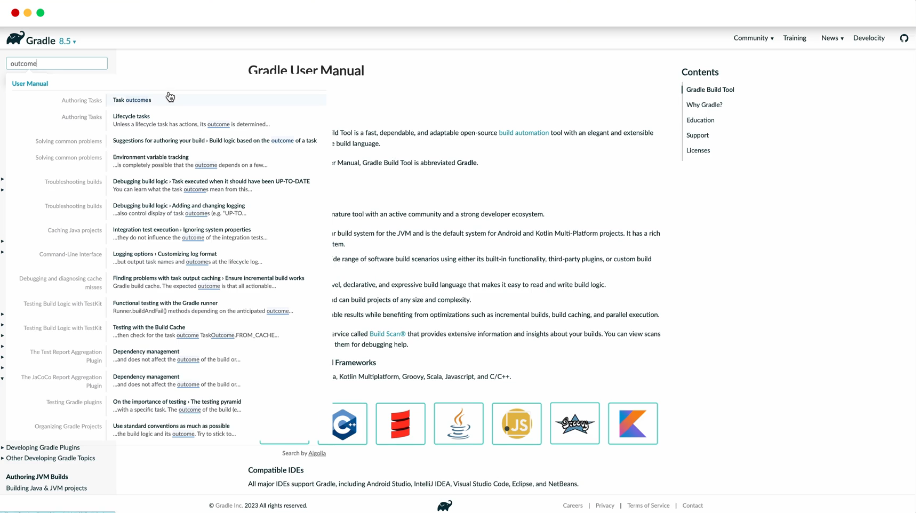
we can see the task is up to date. This is because the compiled output already exists, and we did not change any of the inputs.



If we clean the output of the task, which we can do by prefixing the task with the string clean as follows, we can see the output no longer exists.



Now if run the task, we can see there was no outcome label. That means the compile action executed and it was a success. We can see the output once again. This is a performance feature in Gradle build tool known as incremental build support, and it applies to tasks that are deterministic. That is, tasks for which the same inputs will always produce the same outputs, which most tasks are. We dive further into incremental build support and build caching in other training session so you can check those out too. There are other outcome labels, which you can refer to in the docs.



As you use Gradle build tool and encounter them, you can read up and learn more about what each one means.

Completed…