<https://dpeuniversity.gradle.com/courses/012de84f-fcd3-45d4-9c4c-284382eb3f3f/activities/2056ed60-84fc-4566-9c91-5c4ea4f3af1f>

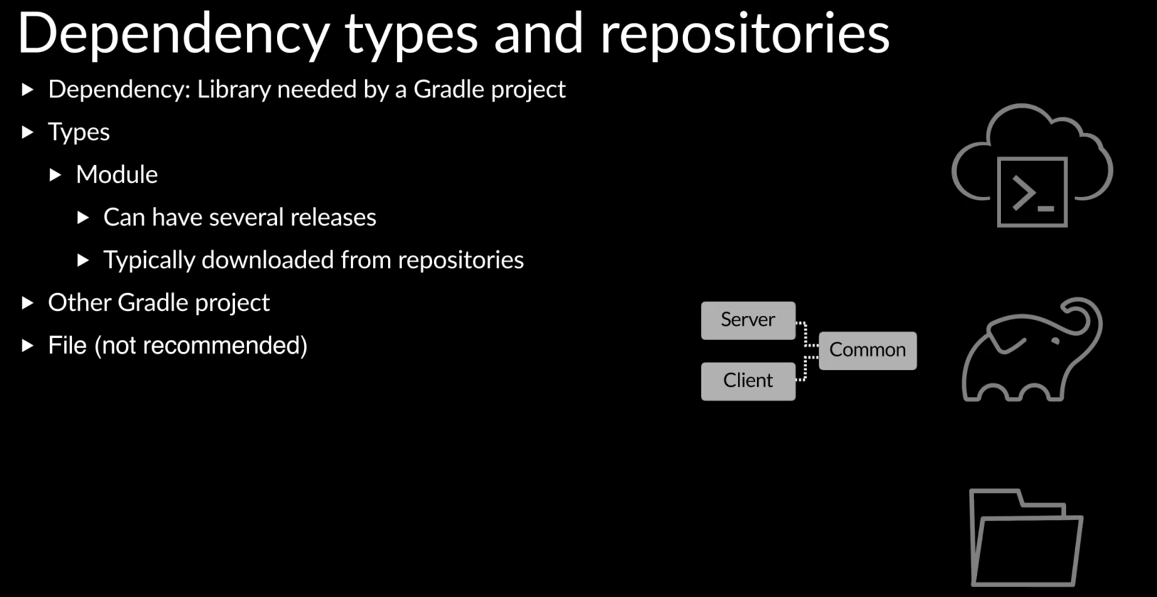
Overview – 10:38

Let’s go over the basics of dependency management with Gradle build tool. In this section we will cover:

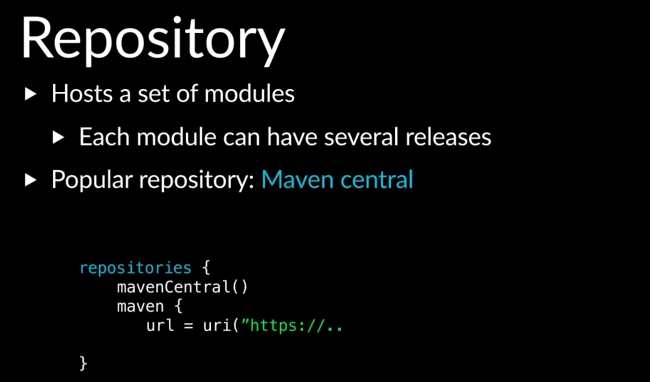
* What the configuration for defining dependencies looks like
  + Module ID
  + Dependency configuration
  + Module version
* How to inspect dependencies in more detail

Dependency Management Overview

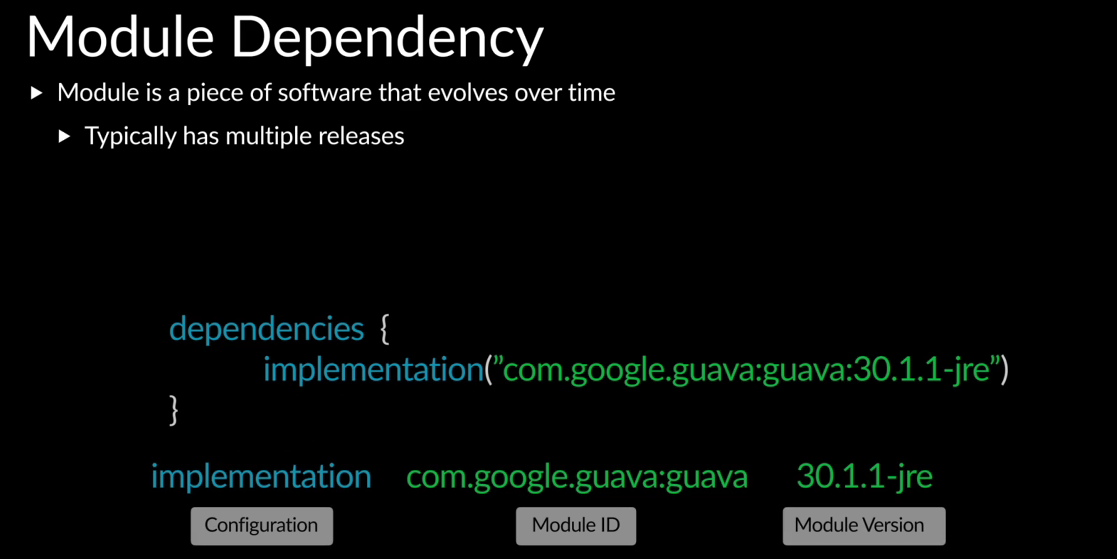
A dependency is another library needed by a Gradle project, and they can be declared in the configuration. There are three types of dependencies.



Modules are libraries that change over time and can have many versions available. When you declare a module dependency, you also specify what version or acceptable versions are and Gradle build tool find and fetch a matching version. Modules are typically downloaded from trusted sites called repositories. A dependency can also be to another subproject. For example, in a chat application where there are three subprojects, a Server, Client and Common, the Server and the Client will have a dependency on the Common subproject. Finally, one can have libraries already downloaded and available locally as files. Historically, people would have a directory where all the dependencies would be put, and sometimes, this directory would even be checked into version control. Although file dependencies are supported, it is recommended to let Gradle build tool manage dependencies and not manually download files. We will focus on module dependencies in this training.



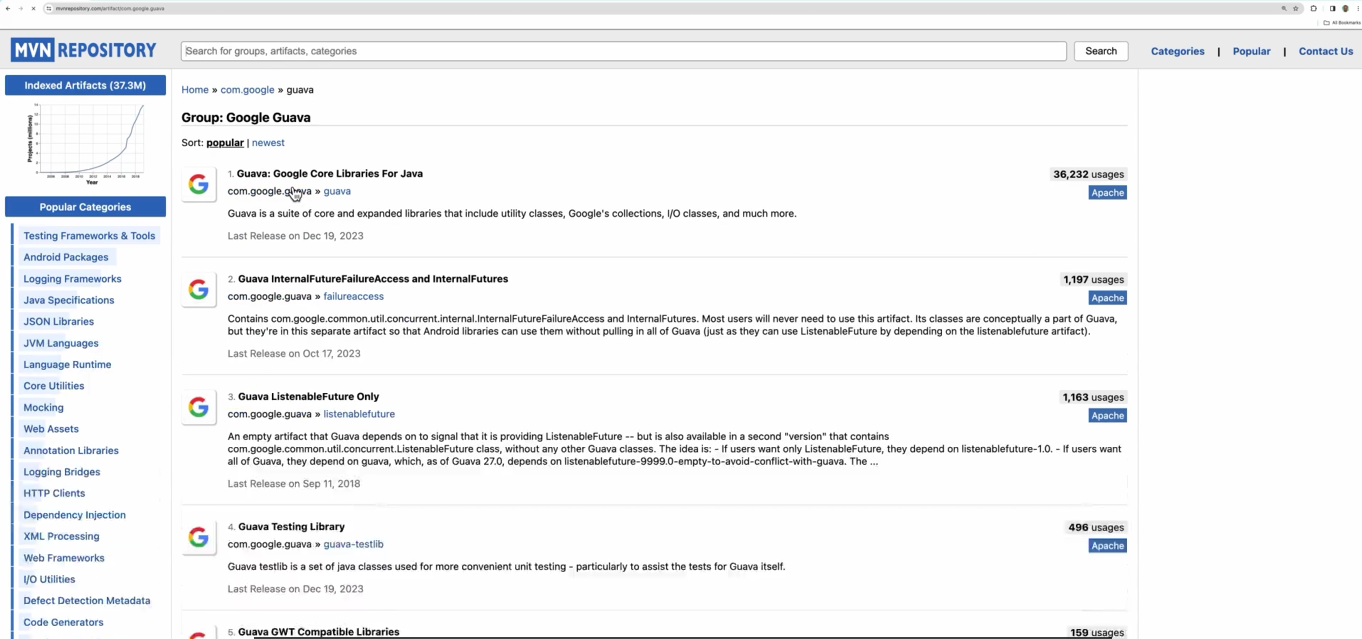
A repository can host a set of modules and each module can have several versions available. You can declare repositories in the repositories configuration. The Defactor repository in the JVM community is mavenCentral. You can also declare additional repositories using their URLs.



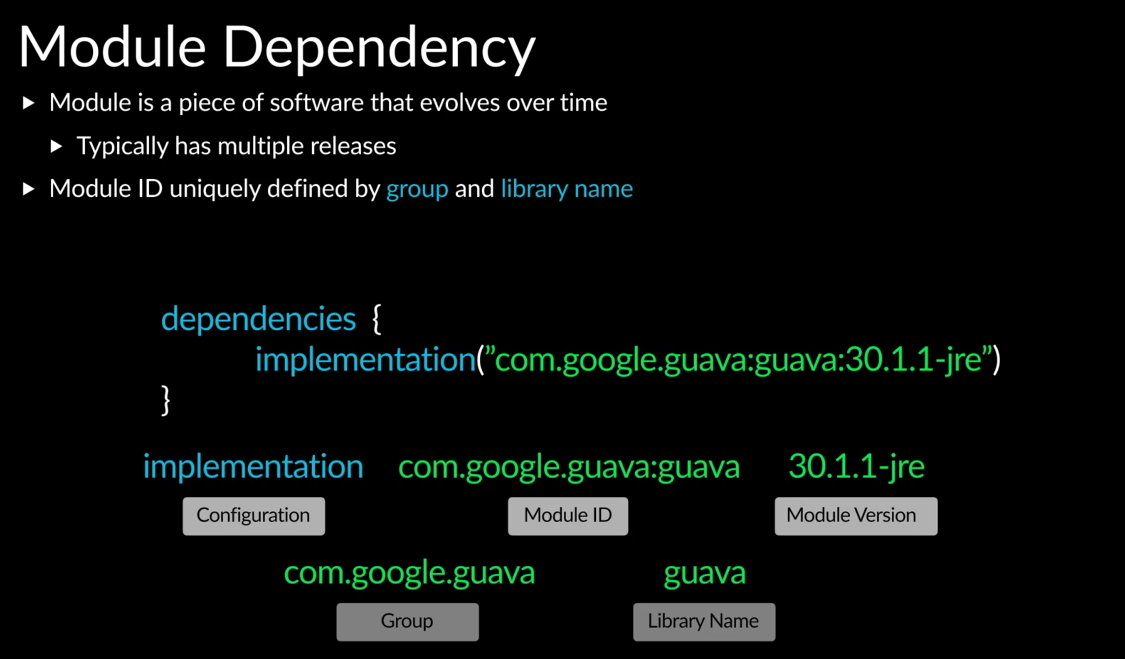
As we mentioned, a module is a library that evolves over time and can have several versions available. When declaring a module dependency, there are three things needed, the module ID, the version, and the dependency configuration.



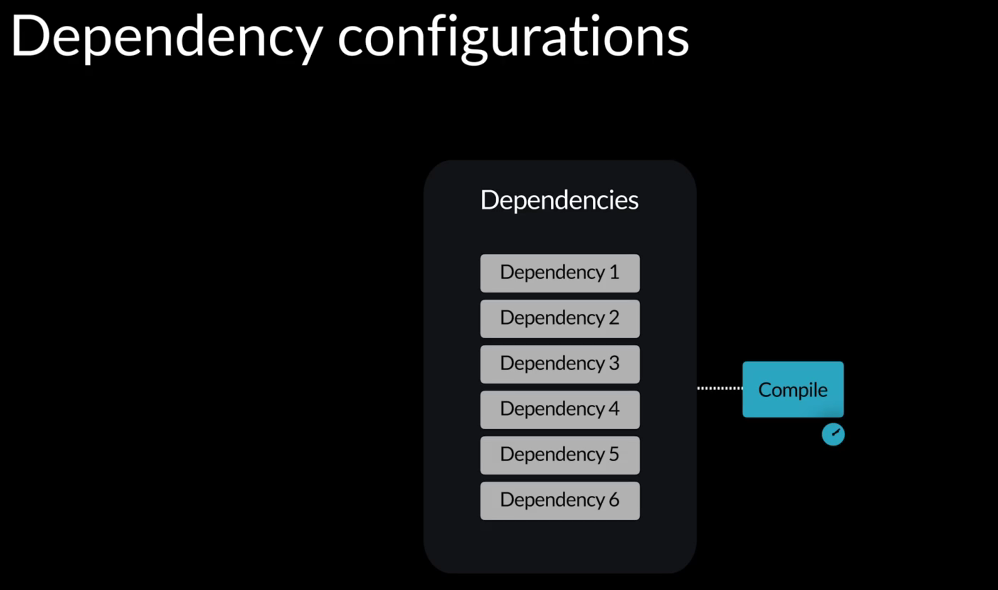
Module IDs are uniquely identified by a group and library name.



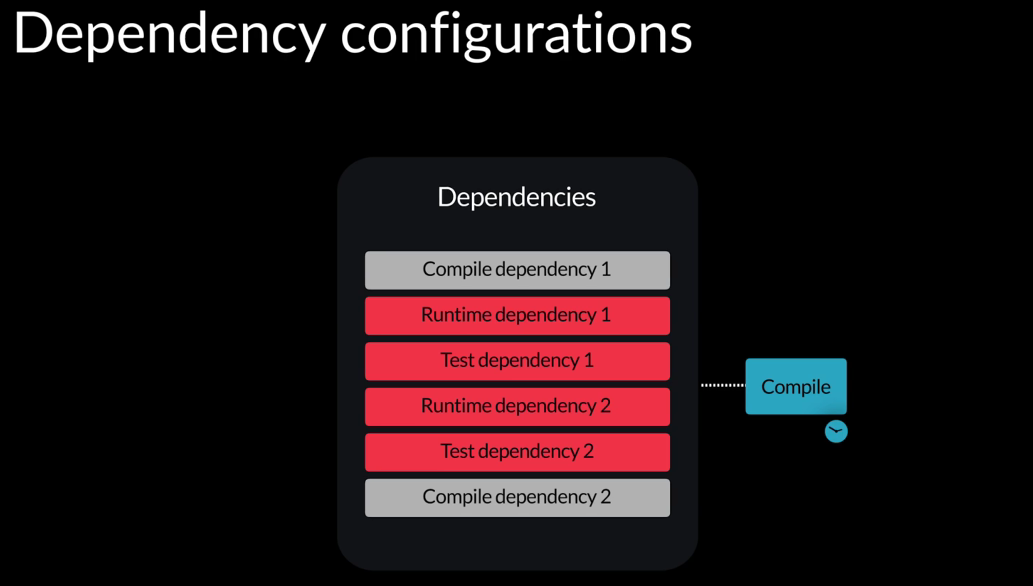
For example, if we search for the Guava library in Maven Centeral, here I’m using mavenrepository.com, we can see there’s a com.google.guava group where there are several libraries. And Guava is one of them.



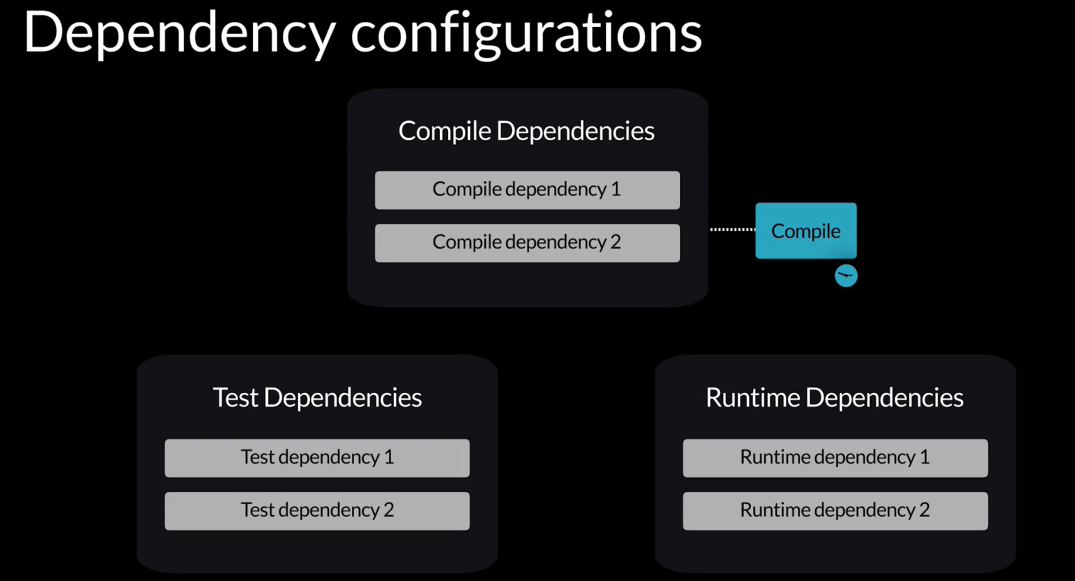
The full module ID is the group: library name, so com.google.guava:guava. In early versions of build tools, people would specify all their dependencies in one list. Over time, the number of dependencies would become large and people started to notice tasks such as Compile would take longer to run.



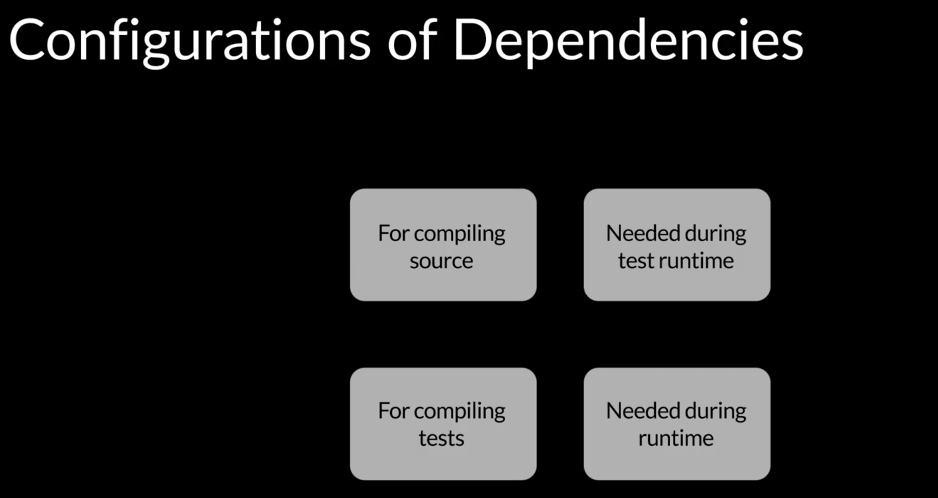
The Compile task has to inspect the dependencies to see which ones have classes that are being referenced by a code. So as the number of dependencies grew in size, the Compile task would take longer to do this. People noticed that not all dependencies were needed by the code.



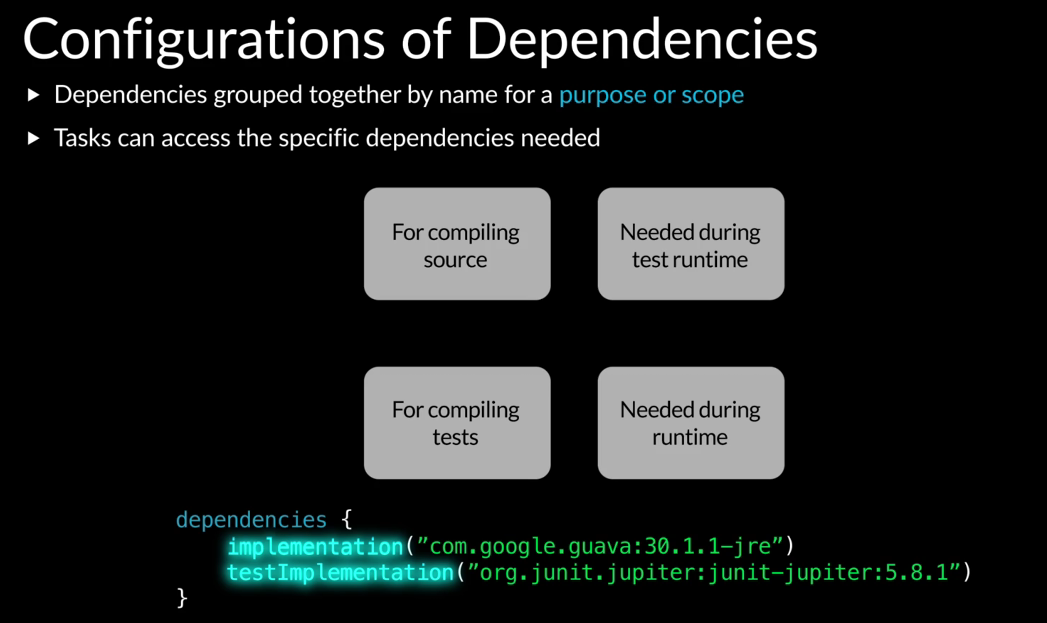
For example, some of them were needed at runtime. Others were needed during testing, but not during code compilation. So the Compile task was wasting time looking at dependencies it did not need to. Build tools evolved to allow you to group your dependencies by purpose or scope instead of having a single list. Now you could specify a list of compile dependencies, a list of runtime dependencies, etc.



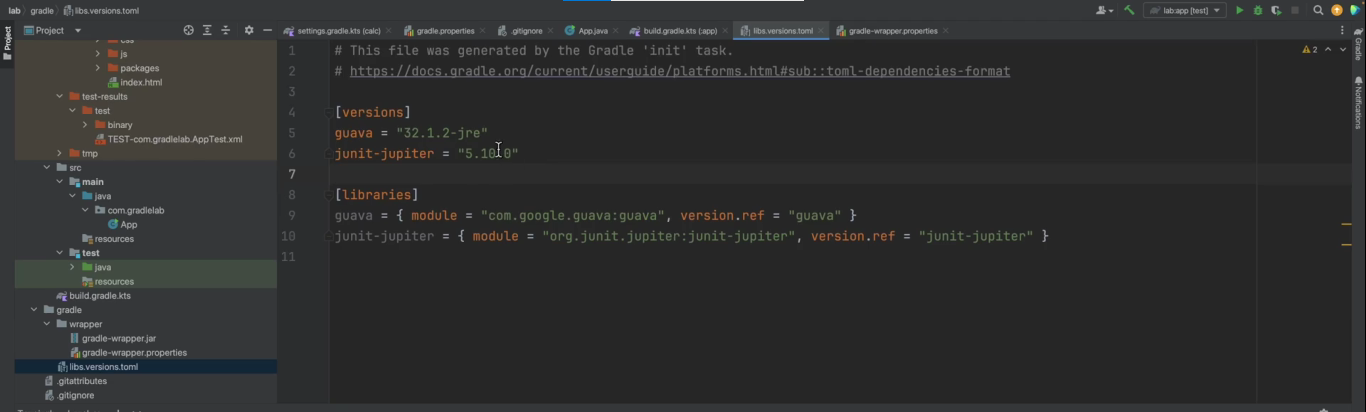
Tasks such as Compile would now have to look at a much smaller list of libraries and they ran significantly faster.



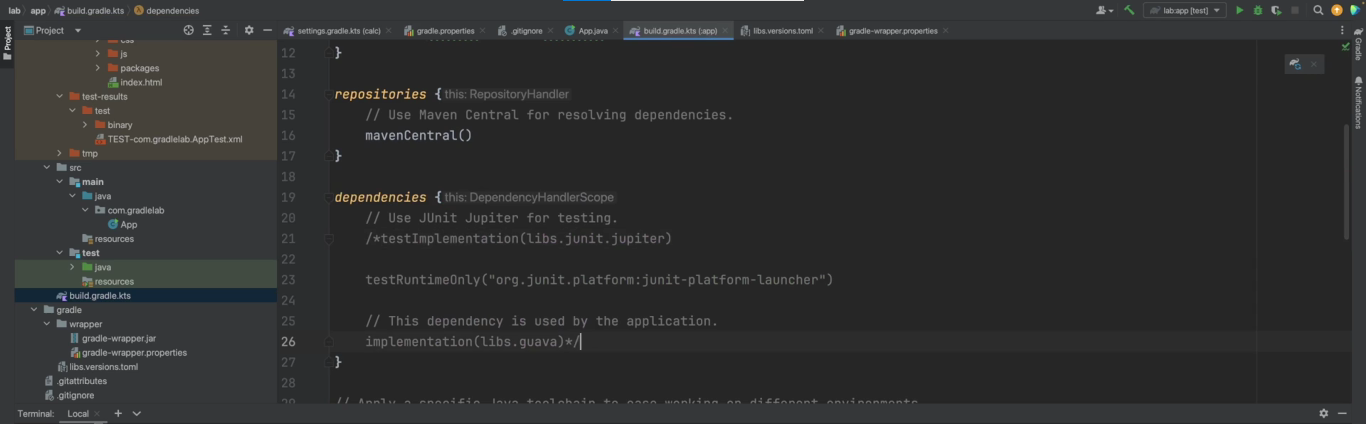
These groupings of dependencies are called dependency configurations. Dependency configurations allow grouping of dependencies by purpose or scope.



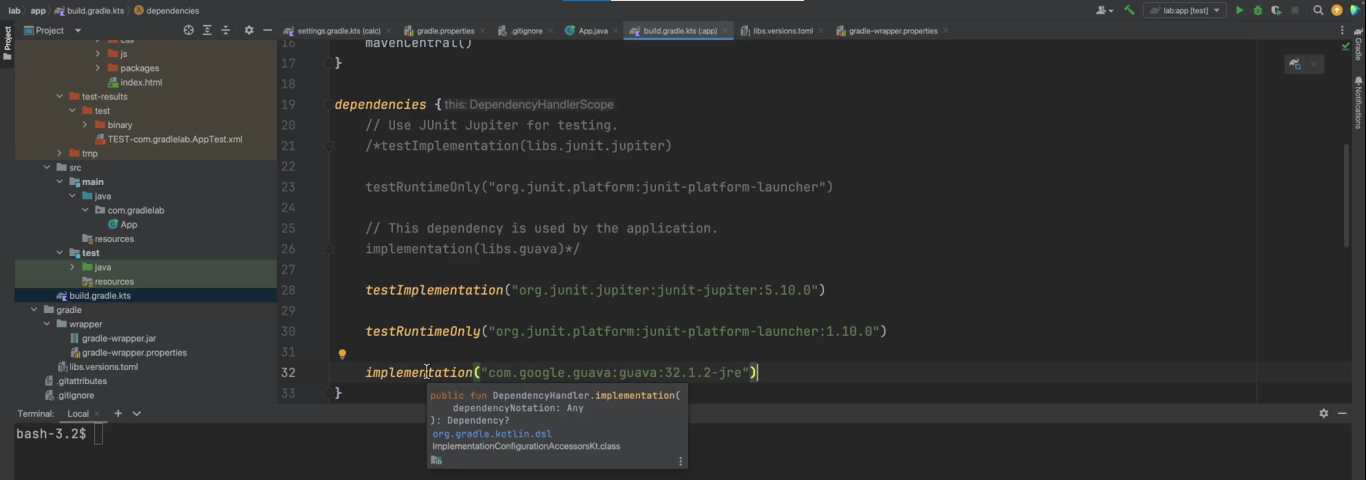
When declaring a dependency, you can define the dependency configuration in which the dependency should be in. We can see Gradle init added some dependencies to the app subproject, which are in the dependencies block. It looks different from what we’ve discussed so far because in Gradle build tool 8.5, gradle init uses the version catalog to define dependencies. So the module IDs and versions go over here.



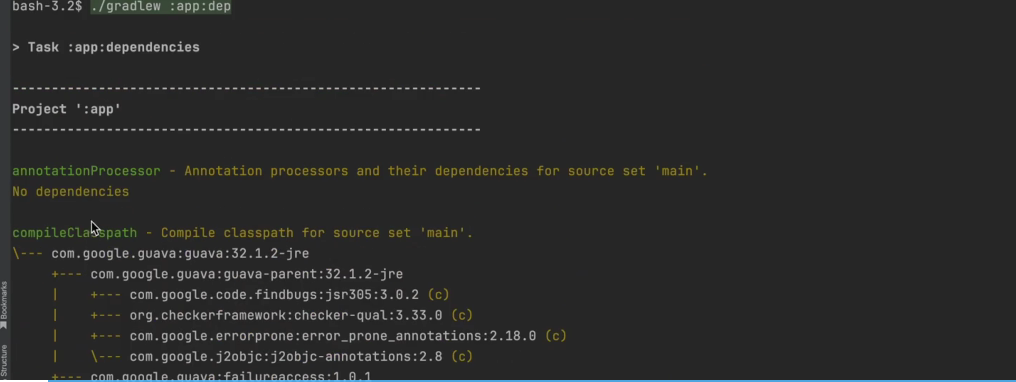
Also, this may look different for you if you’re using a different version of Gradle build tool. We will cover the version catalog later in the training.



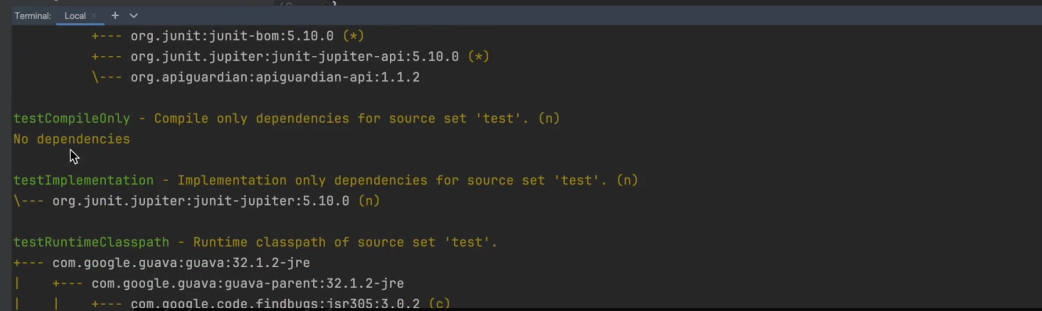
For now, I’m gonna comment this out, and replace it with the equivalent configuration using the syntax we’ve discussed so far. We can see dependencies added to the implementation, testImplementation and testRuntimeOnly dependency configurations.



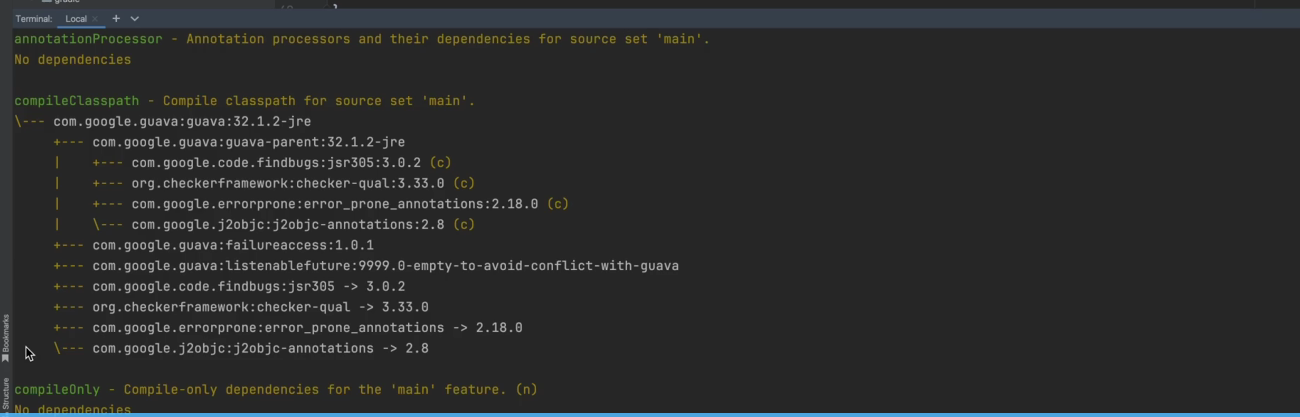
We can use the dependencies task to see more details.



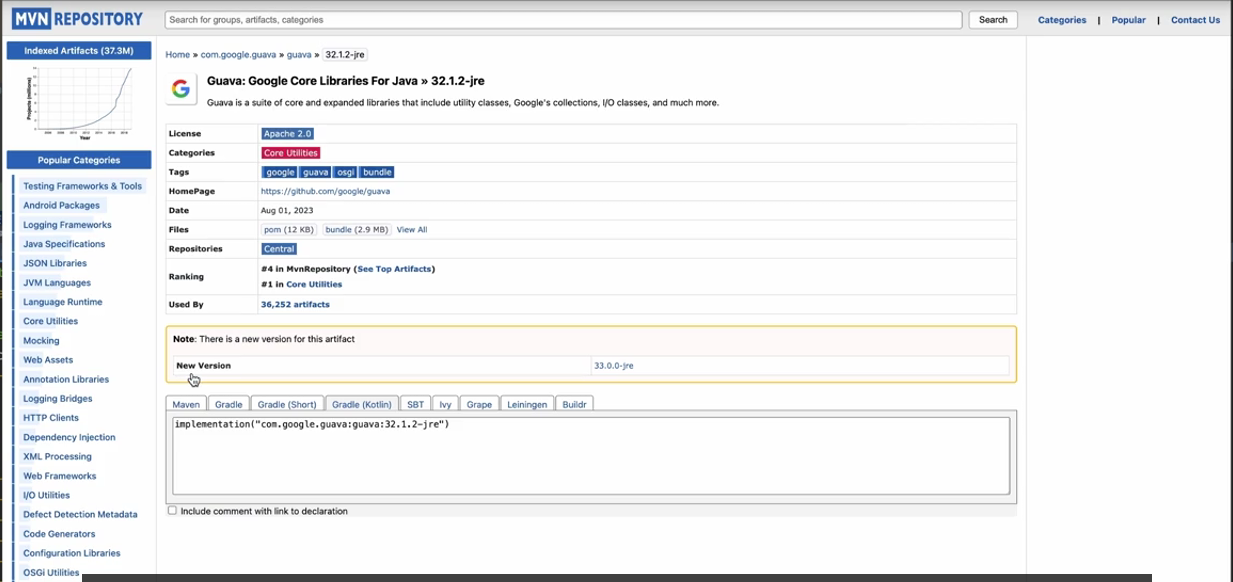
We can also just type dep, which works for this task. We can see there are a lot of dependency configurations.



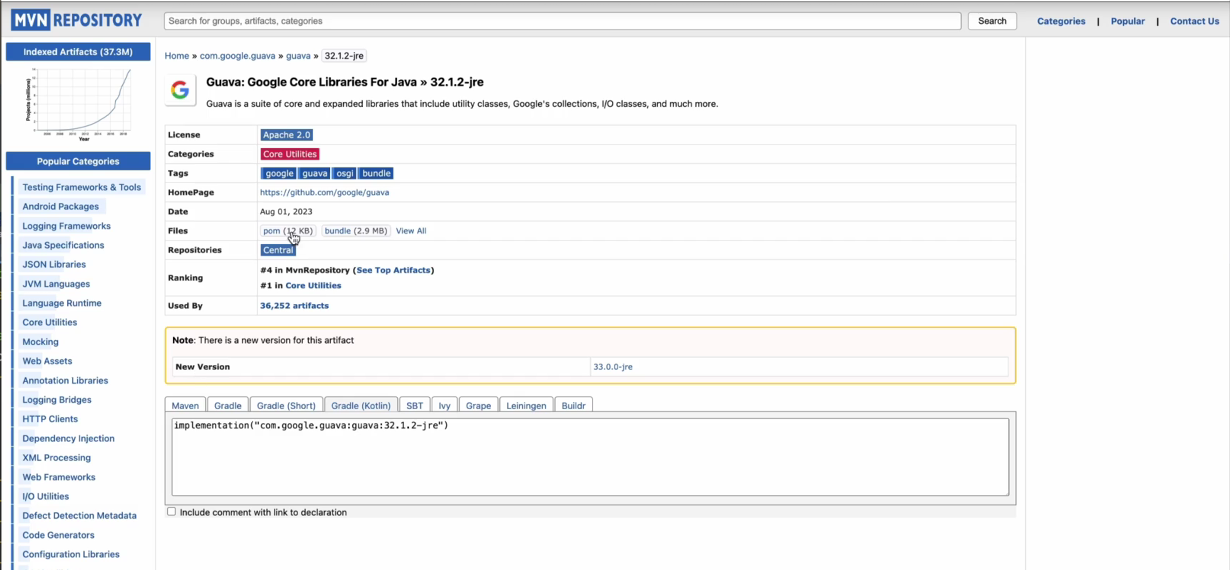
Some of them are empty, and we can see our testImplementation and implementation with the dependencies defined in the build file. We can also see other dependency configurations which are not defined in the build file and have a lot of dependencies. There’s runtimeClasspath, compileClasspath, and others. Where are these coming from?

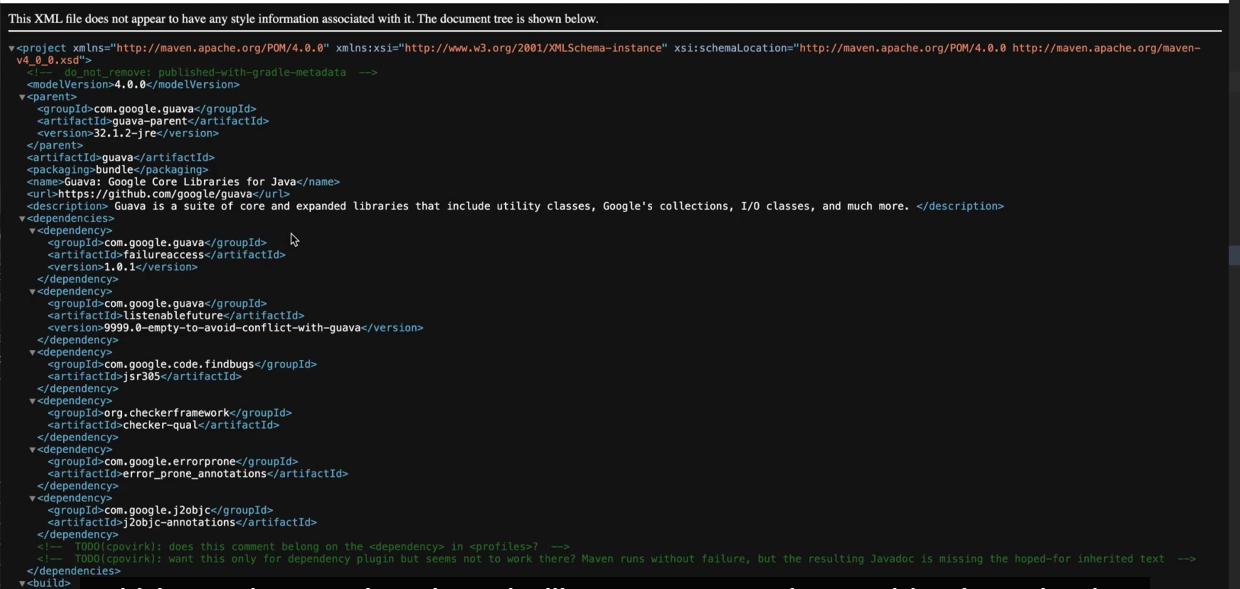


If we inspect the compileClasspath dependency configuration, we can see the Guava dependency. And also some additional dependencies being added, which are not in the build file. Remember that direct dependencies can have additional libraries they need, which are called transitive dependencies.

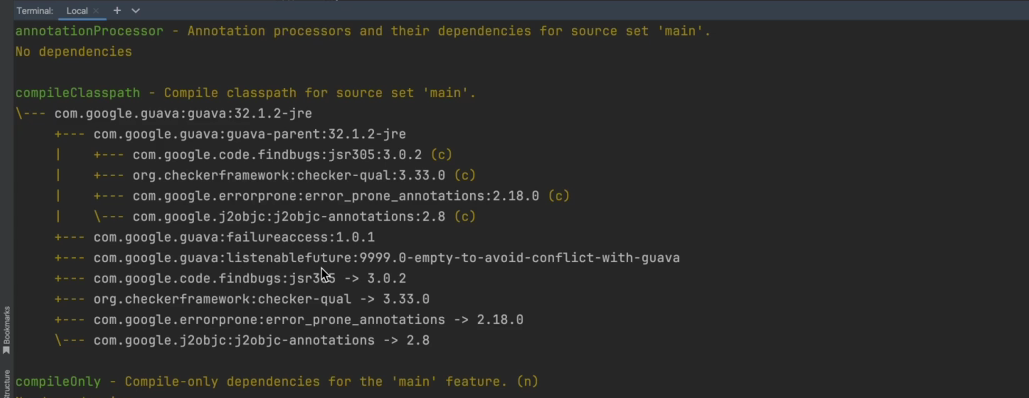


If we take a look at the Guava library and pick a recent version and look at the POM file, and pick a recent version and look at the POM file, which contains metadata about the library, we can see the transitive dependencies.

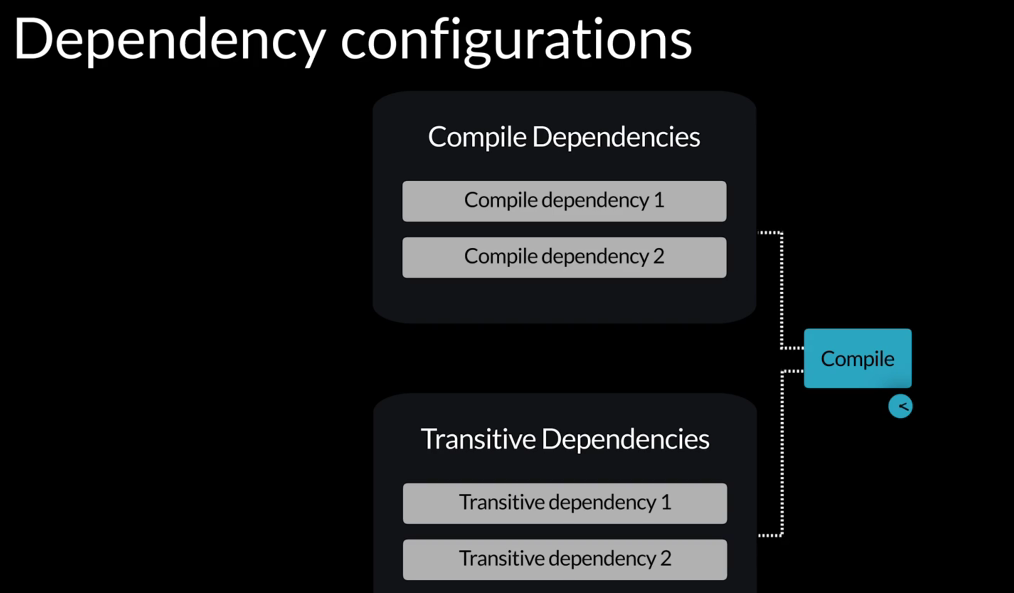




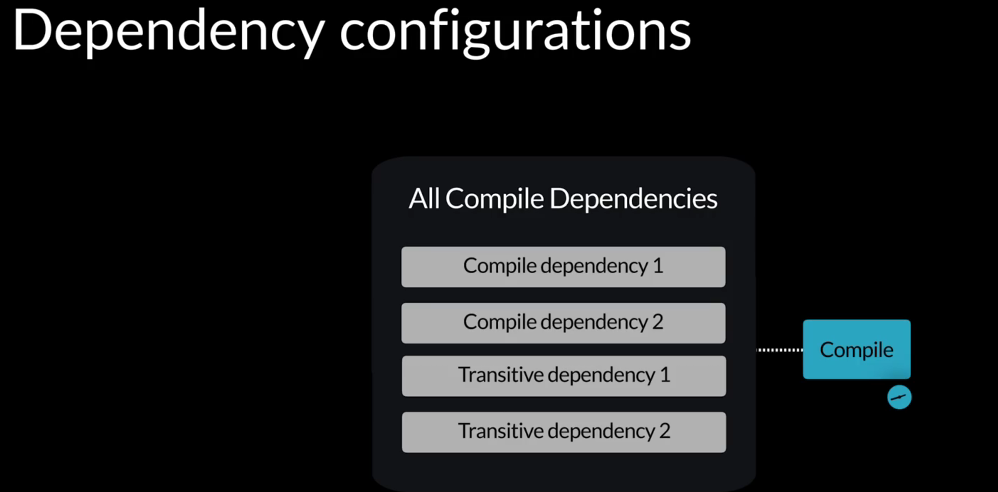
There’s failureaccess, listenablefuture, and so forth. These are the additional dependencies showing up in the dependencies output, failureaccess, listenablefuture, and so forth.



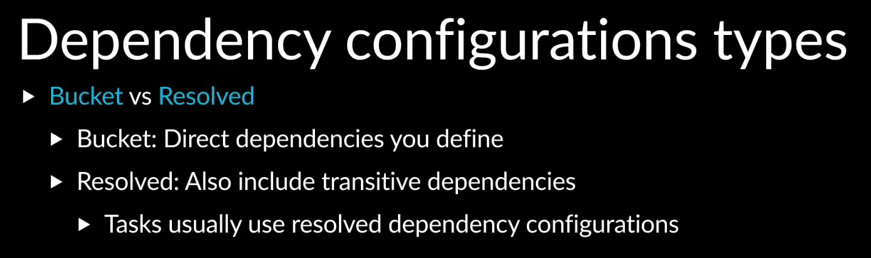
If we think about it, let’s say there’s a dependency configuration for compile dependencies.



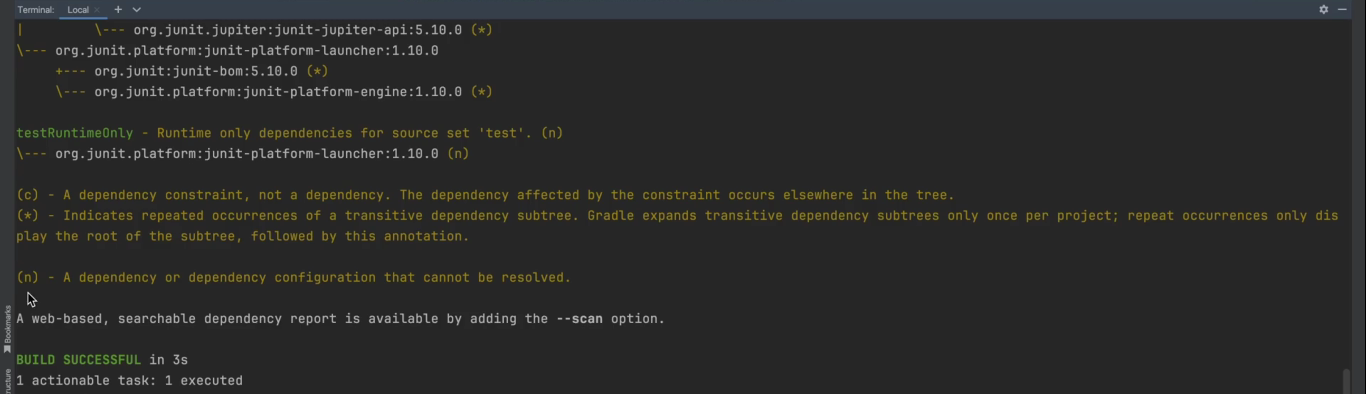
However, these may have transitive dependencies, which the compileJava task may need, so there needs to be a dependency configuration that contains the direct dependencies as well as the transitive dependencies.



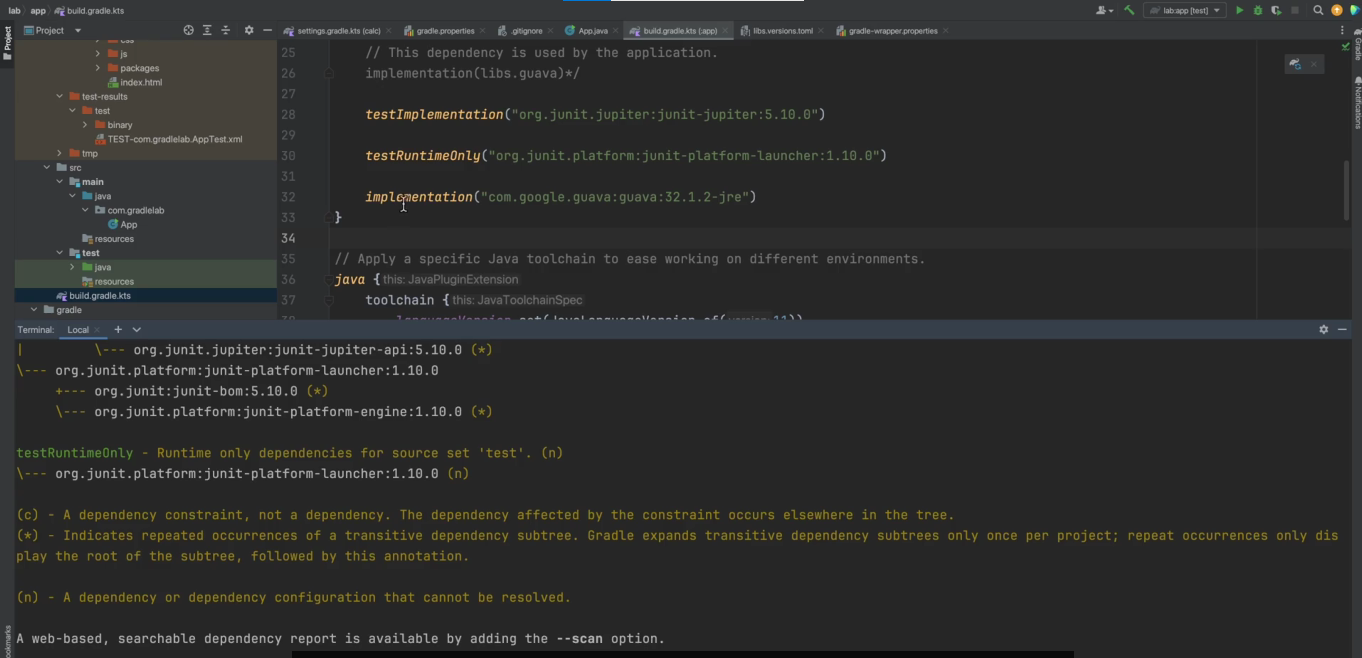
Gradle build tool automatically creates this dependency configuration, which includes the direct dependencies as well as the transitive dependencies, and this can be used by tasks such as the compileJava task.



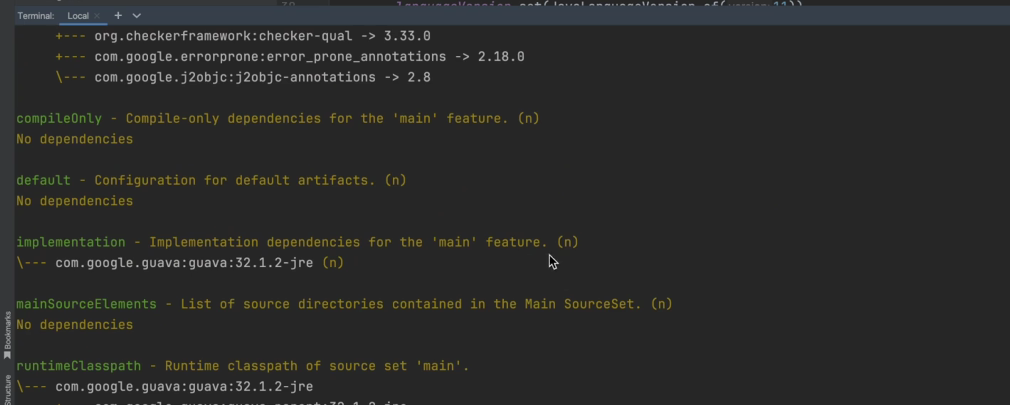
There are two types of dependency configurations, bucked and resolved. Bucket dependency configurations contain only direct dependencies, and we define them in the build file. Resolved dependency configurations are created for us that contain both the direct dependencies and transitive dependencies. Many tasks use the resolved dependency configuration because they need all the dependencies. In the dependencies output at the bottom, we can see a legend that says, (n) means a dependency configuration is not resolved.



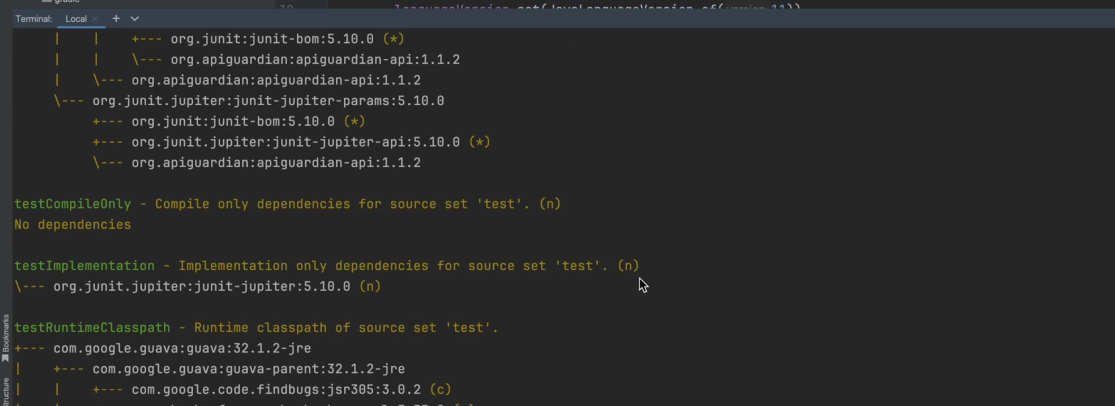
Therefore, it is a bucket. If we look at the dependency configurations, the ones with (n) next to them are buckets.



So for example, our implementation is a bucket,

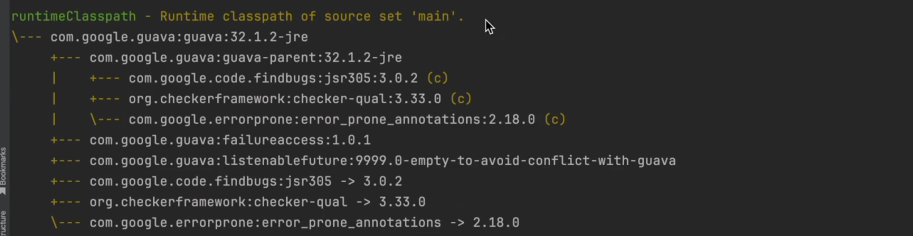


and it has an (n) next to it. testImplementation is also a bucket because we can define dependencies for it in the build configuration,

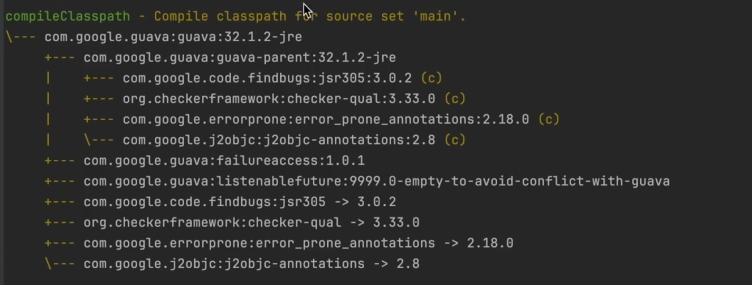


and we can see in the output there is an (n) next to it. The dependency configurations without a (n) next to them are resolved and contain transitive dependencies.

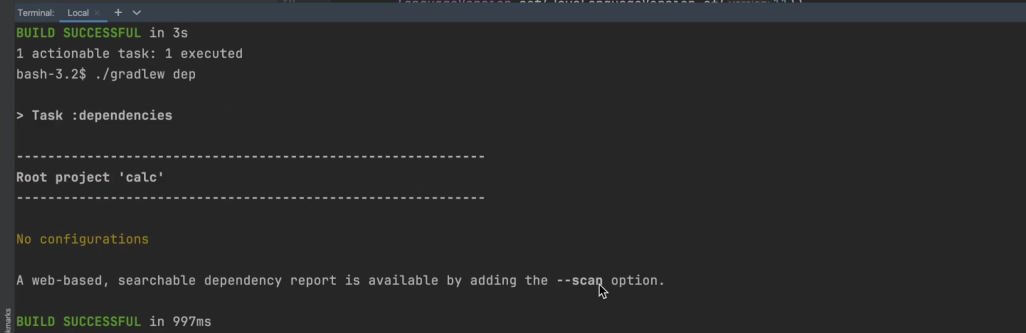
So for example, the runtimeClasspath does not have an (n) next to it, and it’s resolved.



It contains also the transitive dependencies. Similarly, the compileClasspath does not have a (n) next to it, it’s resolved,



and it also has the transitive dependencies. The dependencies task can be useful to inspect the full list of dependencies being used by a project. We mentioned earlier that it’s good to get into the habit of specifying on which subproject you want to run a task on. Some tasks when run on the root project will run on all subprojects, but not all tasks behave this way. The dependencies task is an example of a task that does not run on all subprojects when run on the root project.



When run on the root project, it only runs on the root project, which has no dependencies defined.