<https://dpeuniversity.gradle.com/courses/012de84f-fcd3-45d4-9c4c-284382eb3f3f/activities/2e775984-dba0-4e90-8d5d-94c4d690c133>

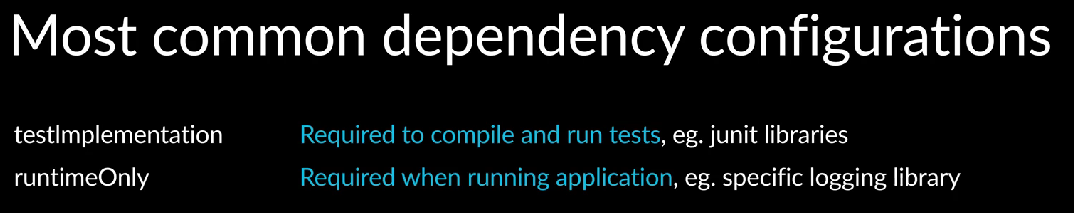
API vs Implementation – 12-54 minute

Let’s dive into the differences between the API and Implementation dependency configurations. We will discuss in which cases you’d want to use each one.

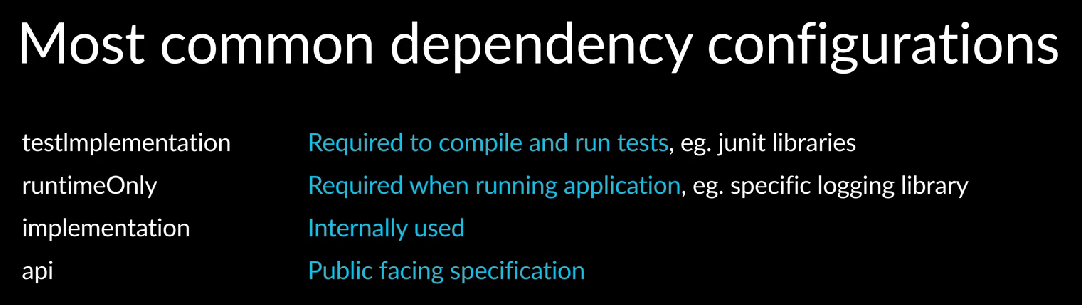
We have learned that in the build files, we will define bucket dependency configurations. Here are some common ones you will use.



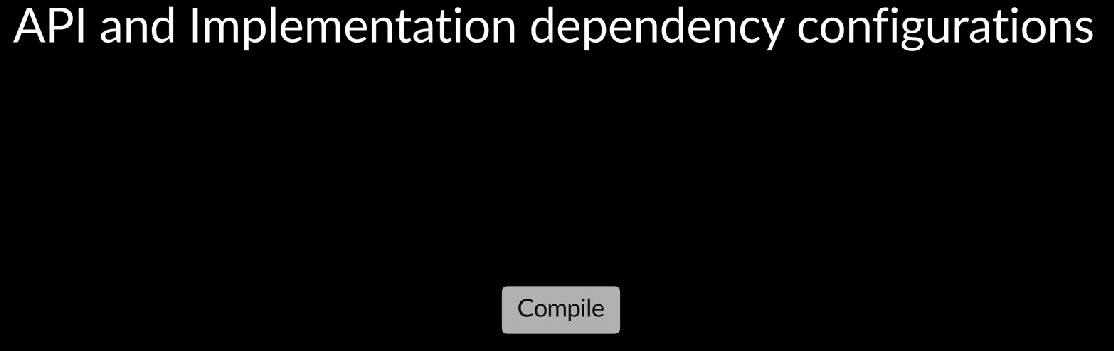
Test implementation is for dependencies needed to compile and run the tests. For example, the Junit libraries. Runtime only dependencies are those needed when the application is running. They’re not needed to compile the application.



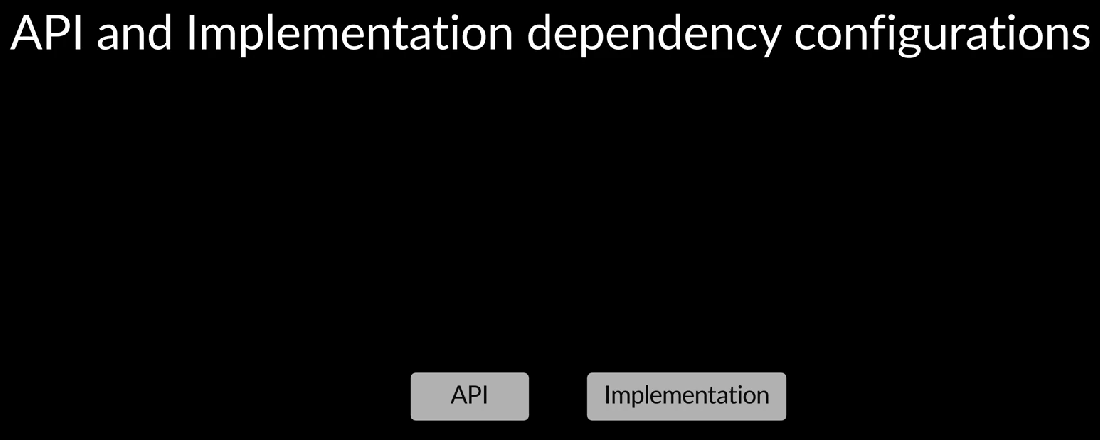
For example, there may be a logging library required at runtime. However, it’s not required during compilation.



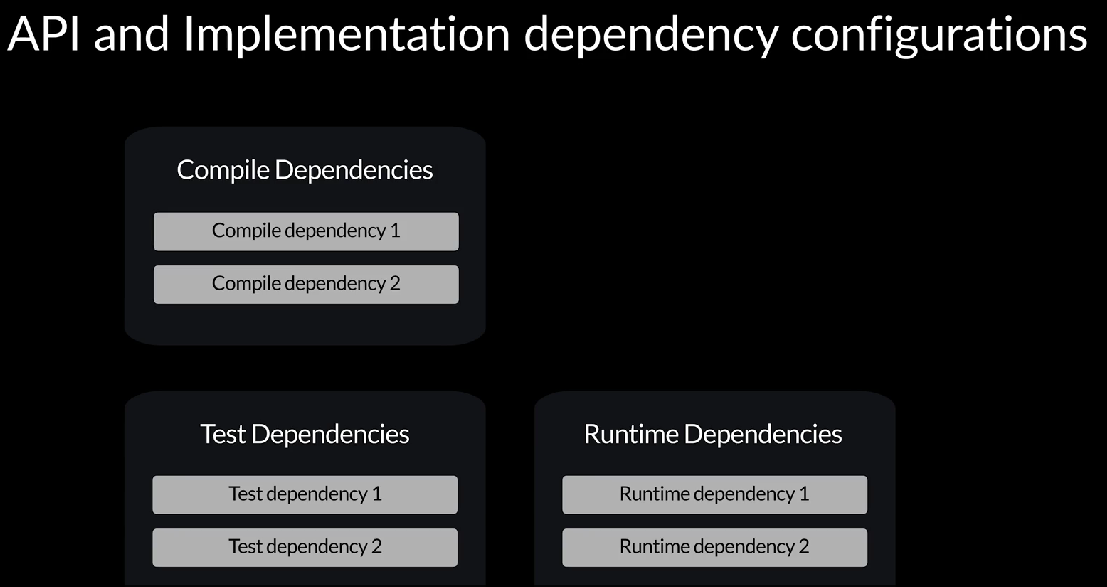
There are two compiled dependency configurations, implementation, and API, and it can get a little tricky to understand the difference between the two. In the earlier versions of Gradle build tool, there was only one bucket dependency configuration for compiled dependencies called compile.



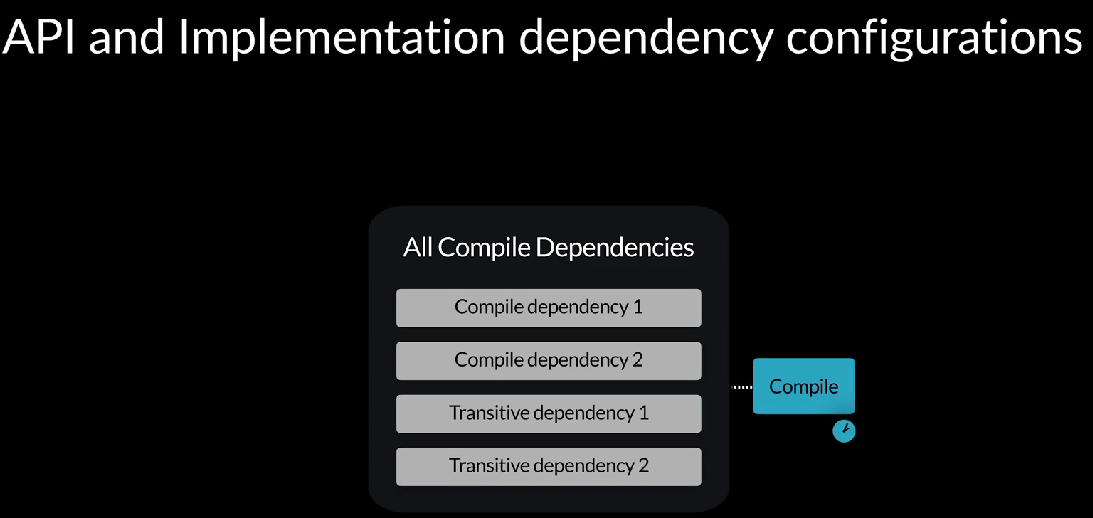
To enable improved compilation performance, it was split into two, implementation and API, and the compiled dependency configuration removed.



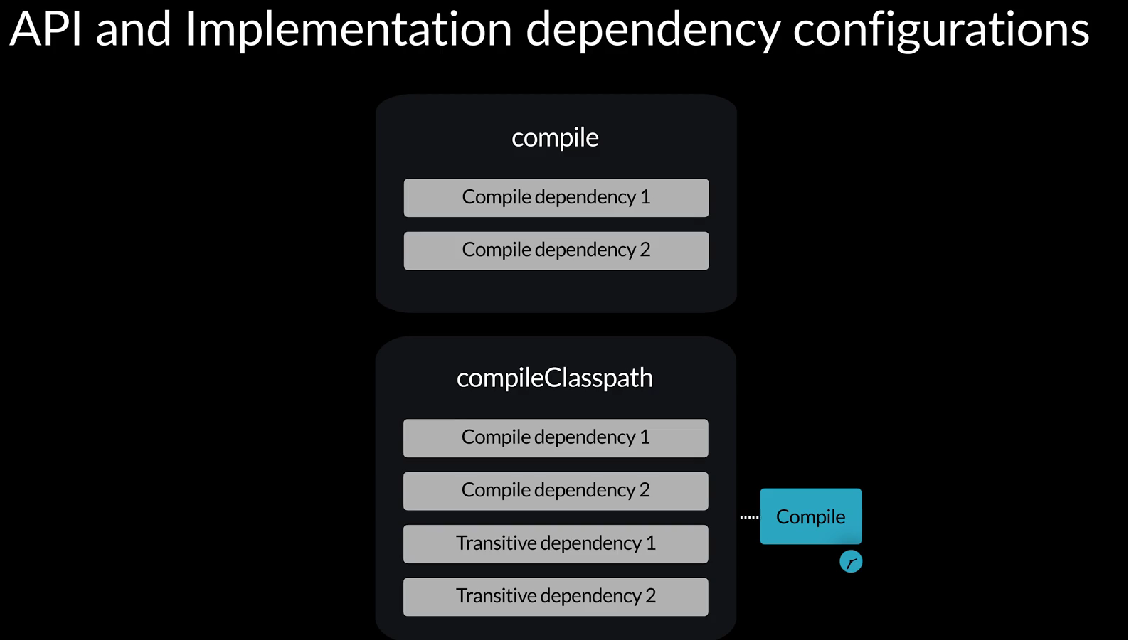
Let’s go over why? With dependency configurations, performance of tasks that dealt with dependencies improved significantly since they did not have to look at one big list of dependencies.

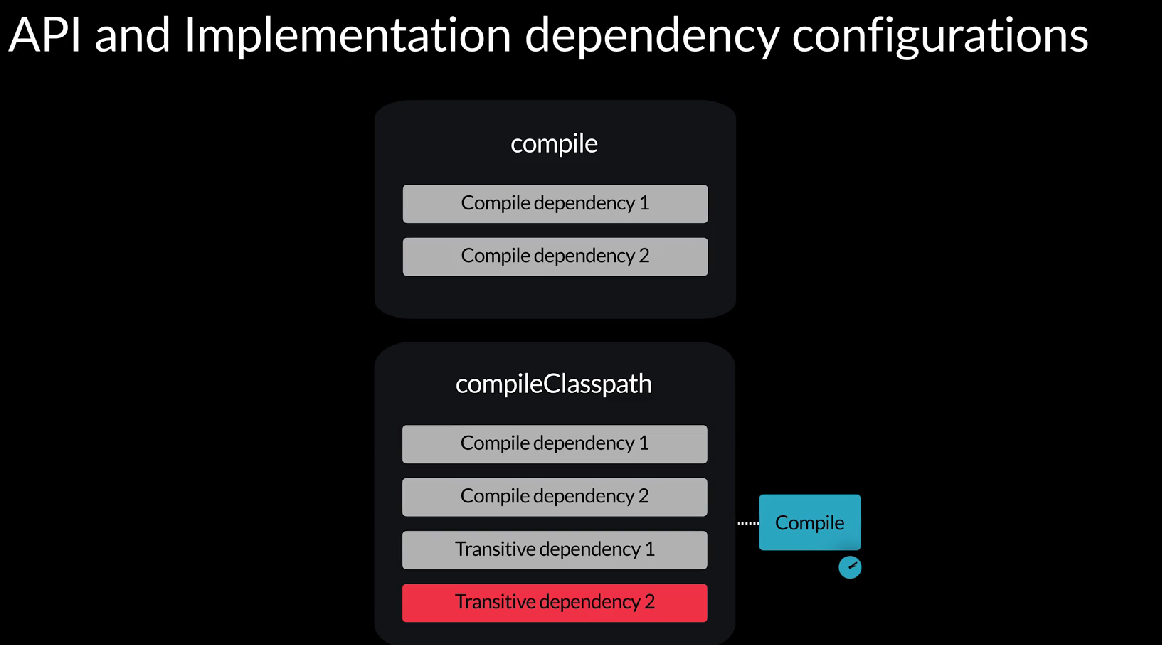


However, over time, as projects grew, tasks started to slow down again. The compilation tasks are particularly important in terms of performance, so further investigations were done to see what else could be improved. One thing that was observed was that even if projects did not have too many direct dependencies, those project dependencies sometimes had a large number of transitive dependencies, resulting in large resolved dependency configurations.

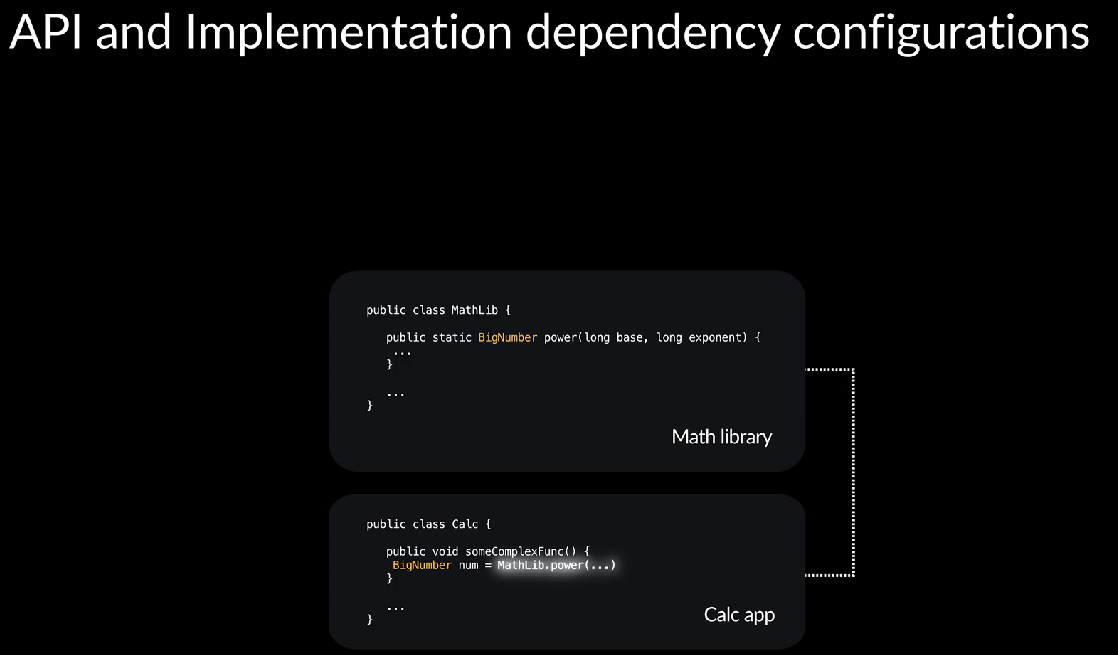


So for the compilation task, even if the compiled bucket dependency configuration was small, the compiled classpath resolved dependency configuration could be large.

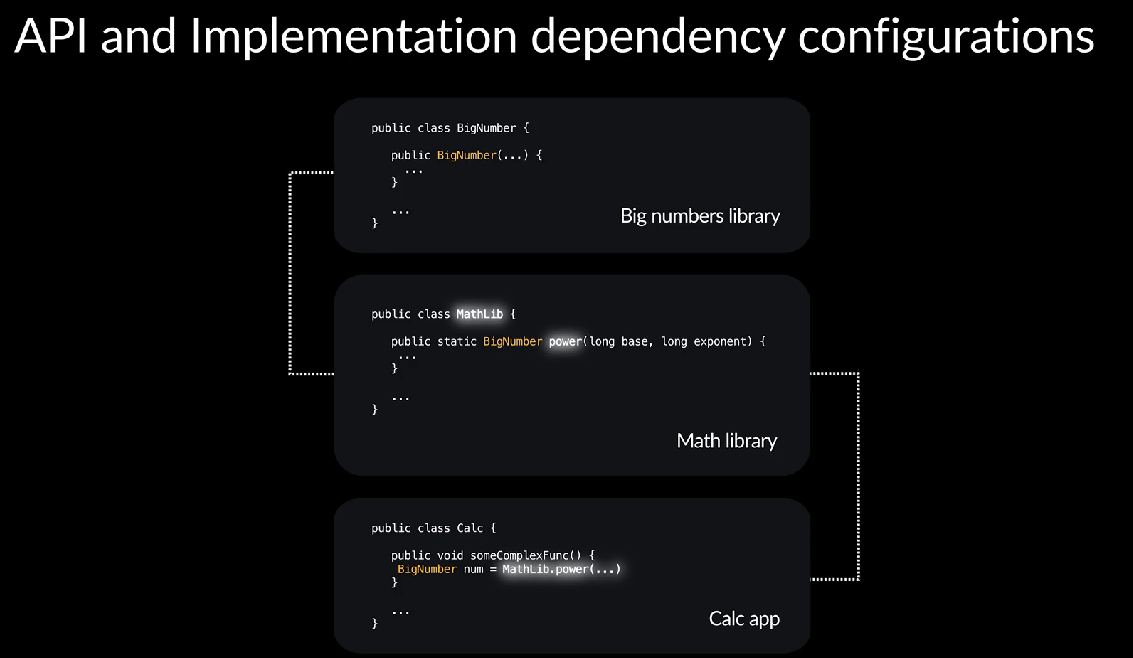




Therefore, compilation tasks had to look at a lot of libraries and hence took more time. It was also observed that not all transitive dependencies were needed for compilation. If classes from a transitive dependency were used in the classes of the direct dependencies API that is used as parameters or return types of public methods or with a type of a public field, then downstream users of the dependency would need to know about the transitive dependency during compilation. For example, let’s say we’re working on a calculator App that is making use of a math library direct dependency.



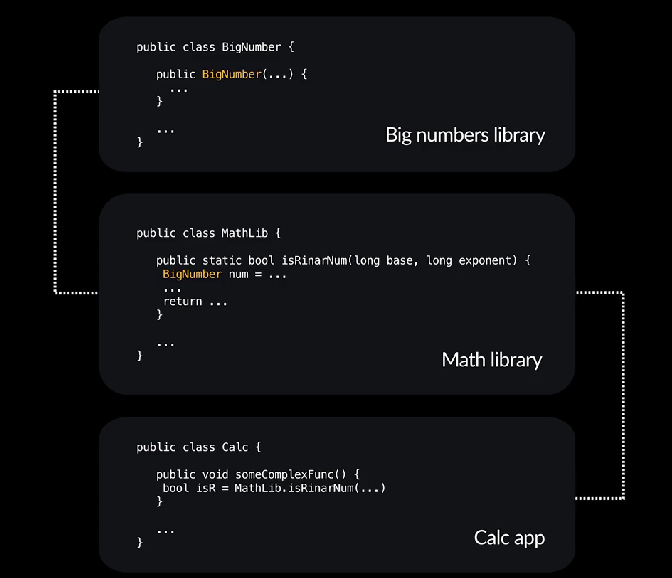
The math library in turn has a transitive dependency on a big numbers library.



There is a big number class that is used in the math library. It is the return type of a public method. In our calculator App, we will also use the big number class.

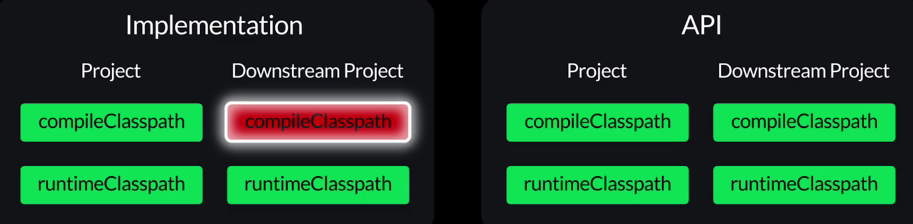


Therefore, during compilation, we need to have the big numbers library available for the compiler. However, if classes from a transitive dependency were used internally in a direct dependency, for example, within a method body, then downstream users would not need to know about the transitive dependency during compilation.

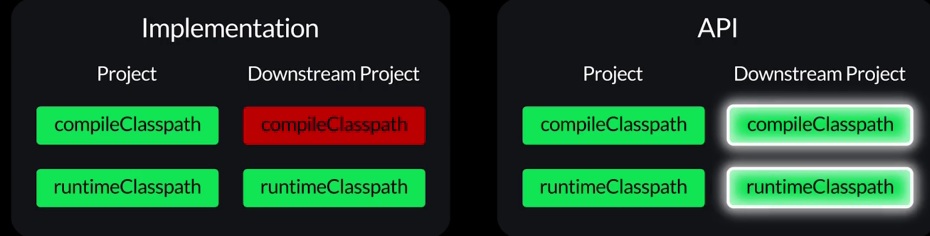


They would need the transitive dependency only during runtime. If our calculator App made use of the math library, however, the big number was used internally in the method implementation inside the math library and our calculator App never saw big number directly. Then during compilation, we don’t need to have the big numbers library in the compiled classpath. We only need it at runtime. If there was a way for authors of libraries to declare which transitive dependenices would be needed during compilation for downstream users, that could significantly reduce the number of transitive dependencies in the compiled classpath resolve dependency configuration for downstream users and therefore improve the performance of the compilation tasks.

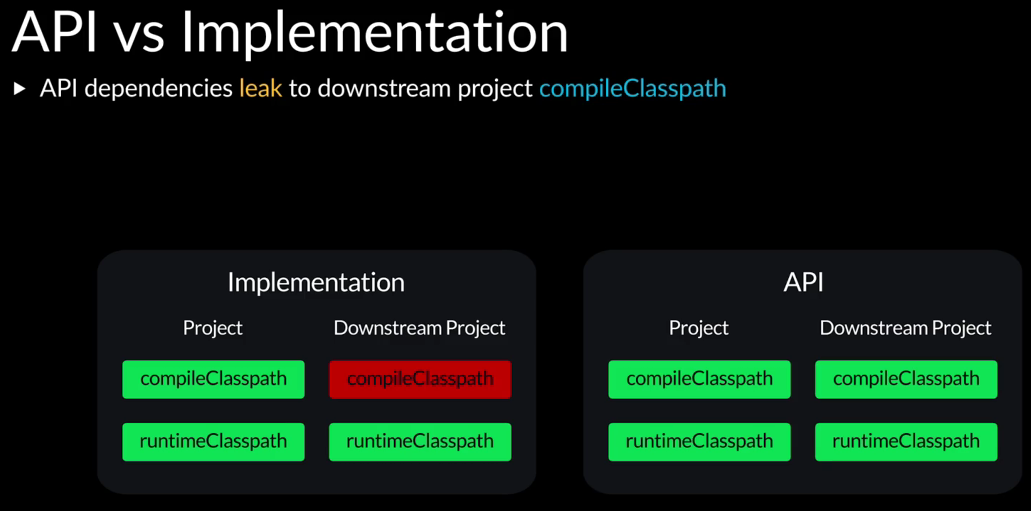
So the compiled bucked dependency configuration was split into two, API and implementation.



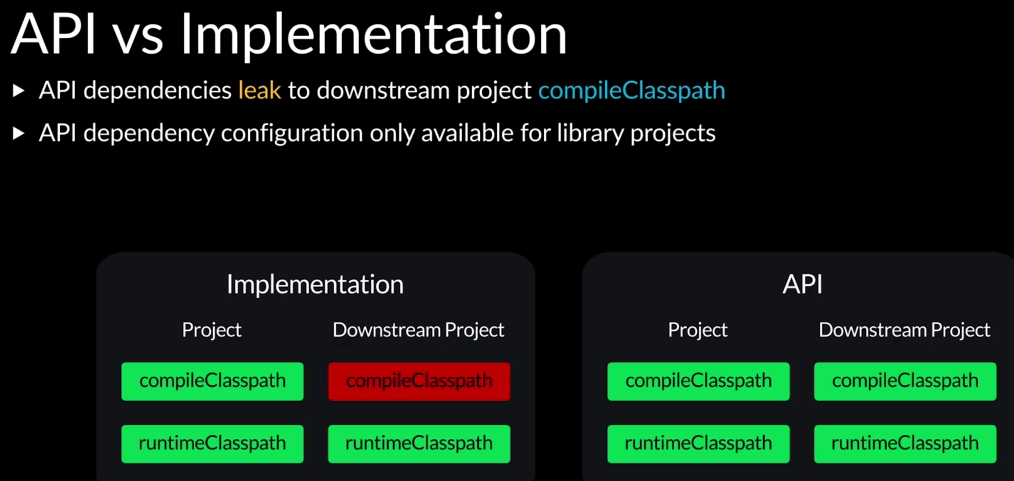
If a dependency is declared as part of the implementation bucket dependency configuration, it will not appear in the downstream user compiled classpath. It’ll appear only in the runtime classpath. Dependencies in the API bucket dependency configuration will appear in both runtime classpath and compiled classpath for downstream users.



The technical term we use is that the API dependencies leak to downstream projects compiled classpath.



It’s also important to note that the API bucket dependency configuration is available only if you are working on a library project.



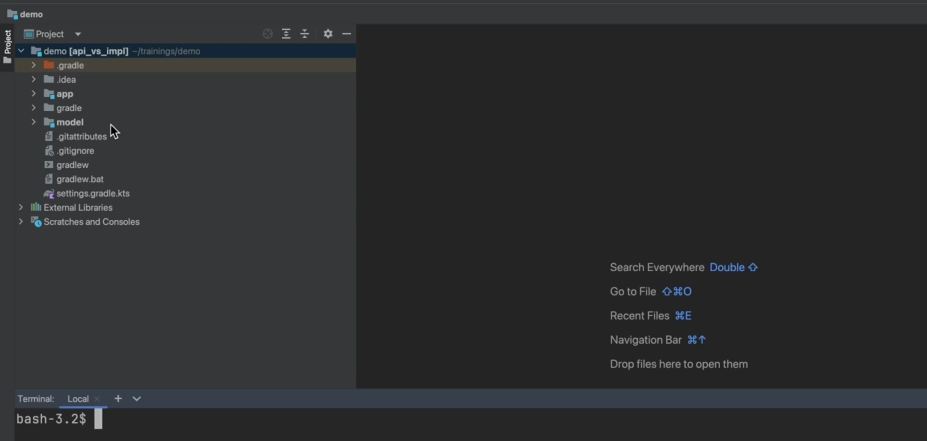
That is something that can be used as a dependency by others. Otherwise, only the implementation bucket dependency configuration will be available. Therefore, if you apply the Java Library plugin, which indicates the project will produce a library that can be published and used by others as a dependency, the API bucket dependency configuration will be available. If you apply the application plugin, the API bucket dependency configuration will not be available. All sorts of libraries can look at how they’re using dependencies and decides which ones to propagate to downstream users, compile classpaths and which ones to not.



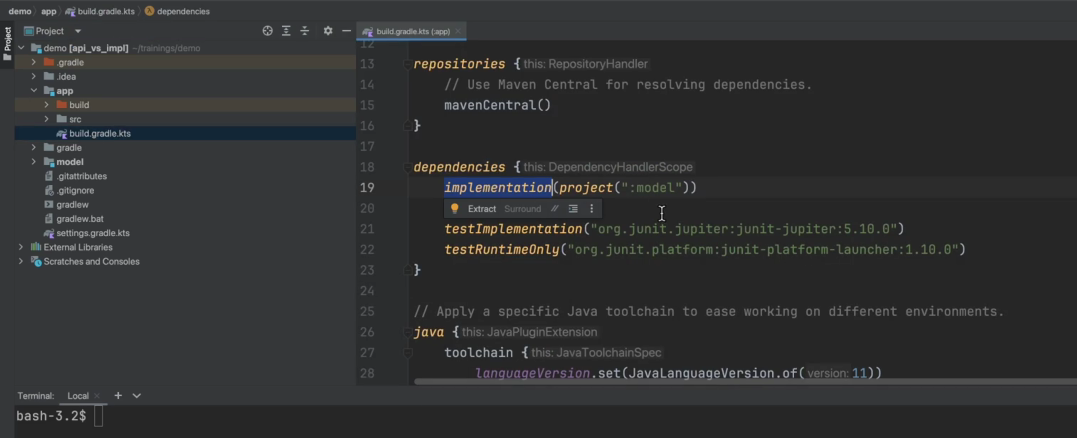
Let’s see how the API and implementation dependency configurations impact downstream users.



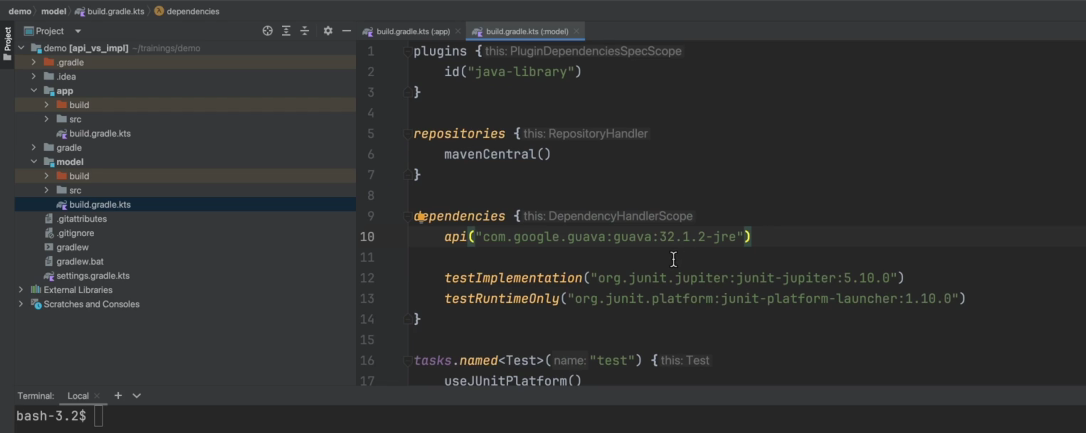
In this project, there are two sub projects, App and Model.



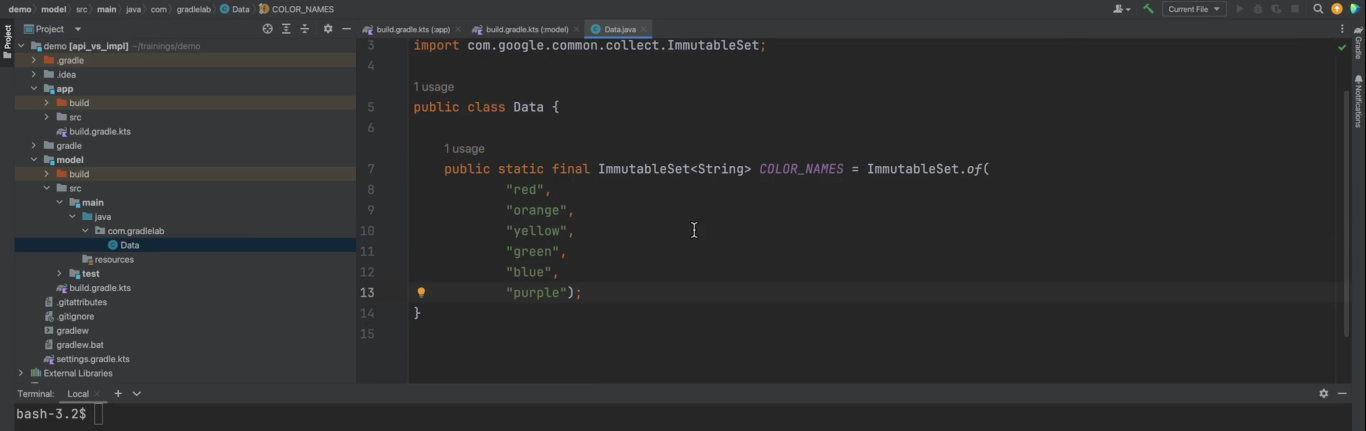
The App sub project has an implementation dependency on the model sub project.



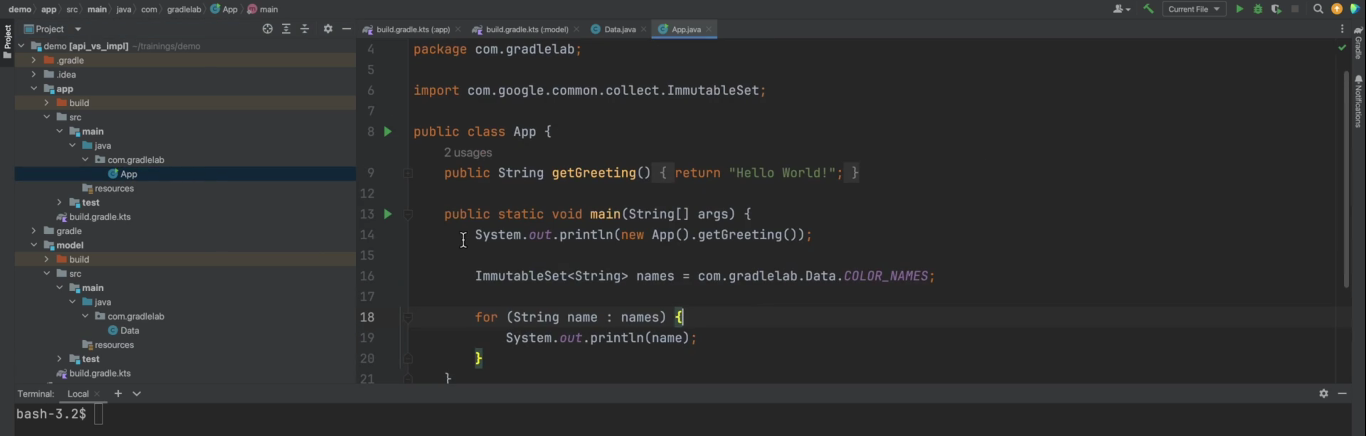
Therefore, App is a downstream user of Model. The model sub project has an API dependency on Guava.



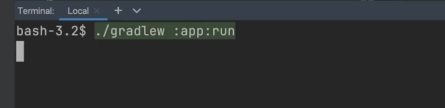
The model has a data class, that makes use of immutable set provided by the Guava library.

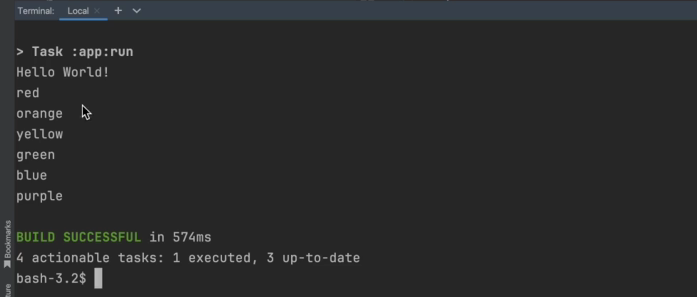


App is using this data class and getting the immutable set from it and using it.

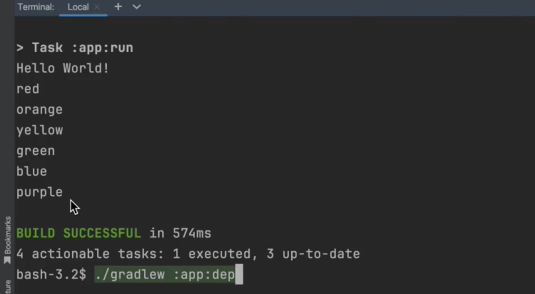


If we run the App,

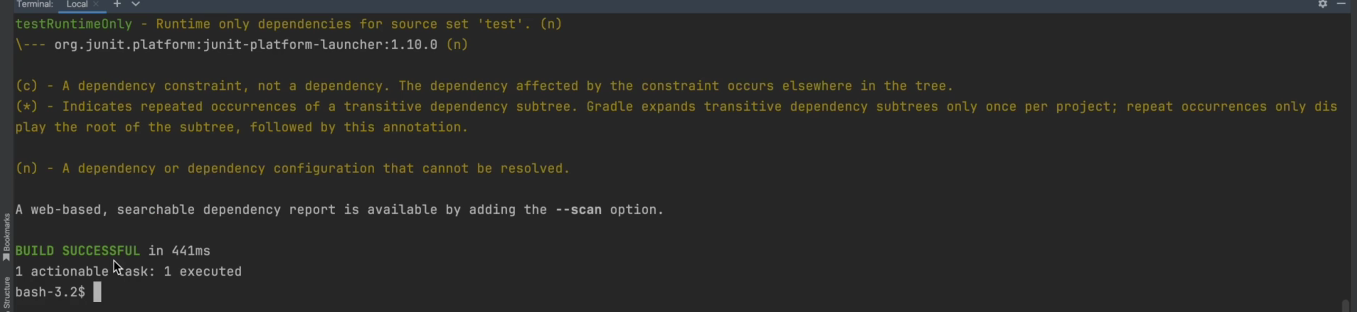




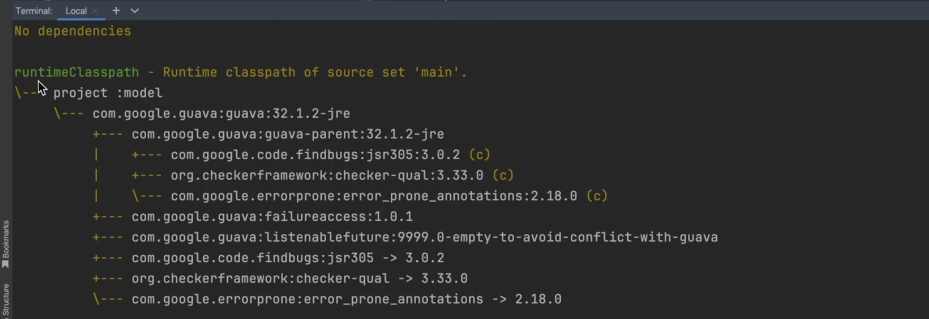
we can see the items from the collection being printed. When we run the dependencies task on the App sub project,



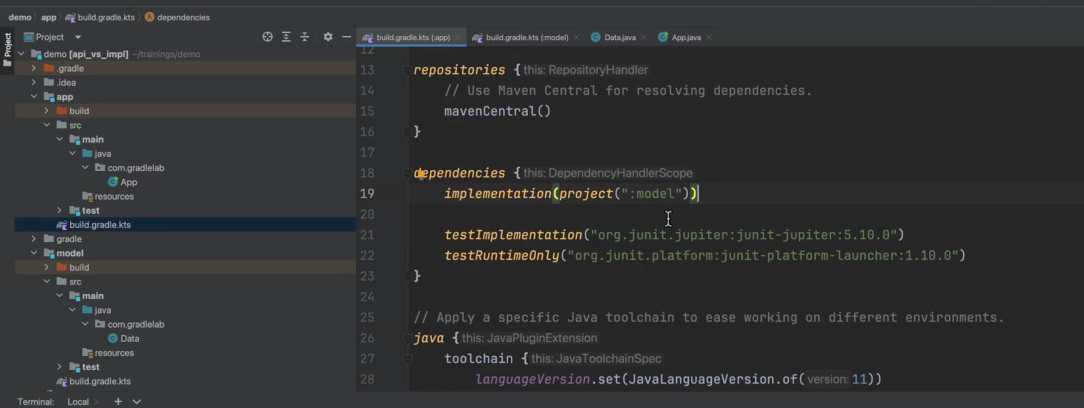
we can see the transitive dependencies,



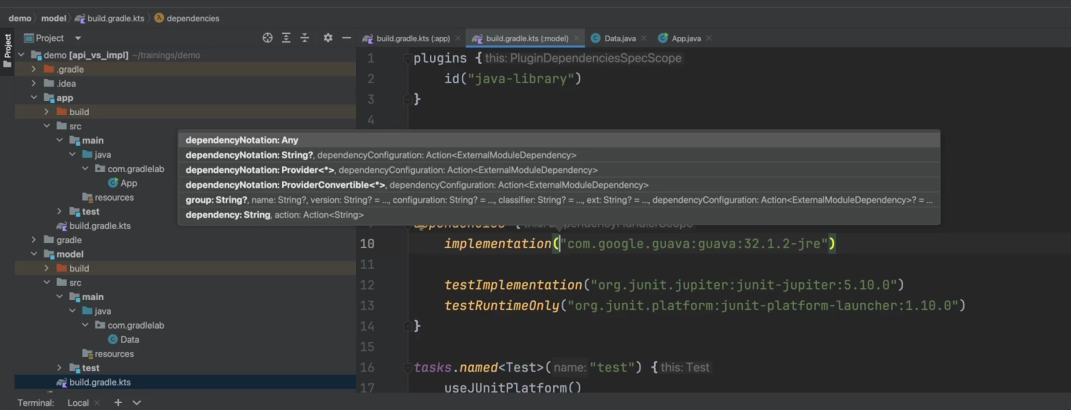
which is Guava and its dependencies show up in both the runtime classpath as well as the compiled classpath resolve dependency configuration.



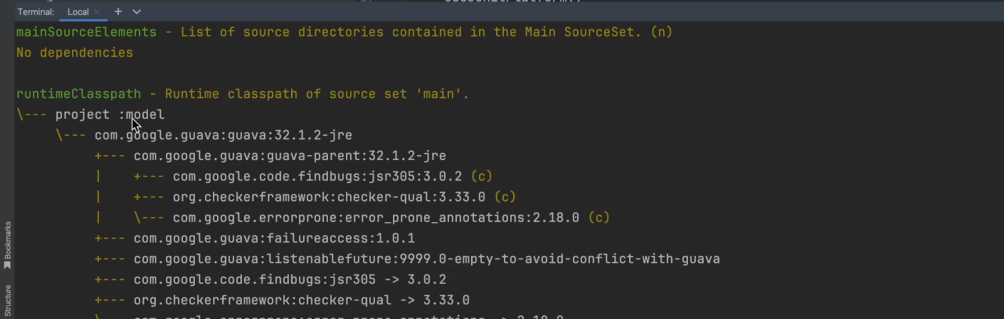
That means that the Guava transitive dependency is also available to the compile Java task in the App sub project. The code can access classes from the Guava library, even though App does not directly depend on Guava.

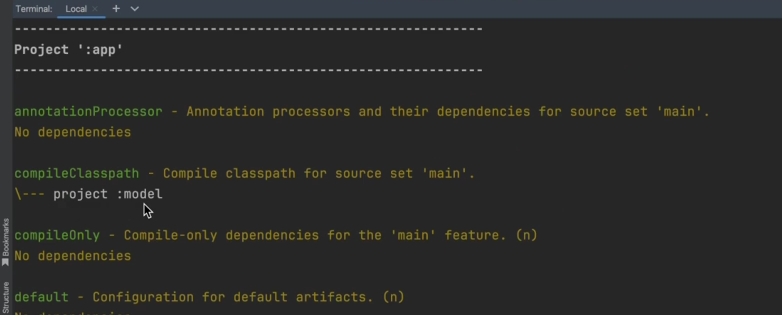


In the model if we change the Guava dependency to be an implementation dependency configuration

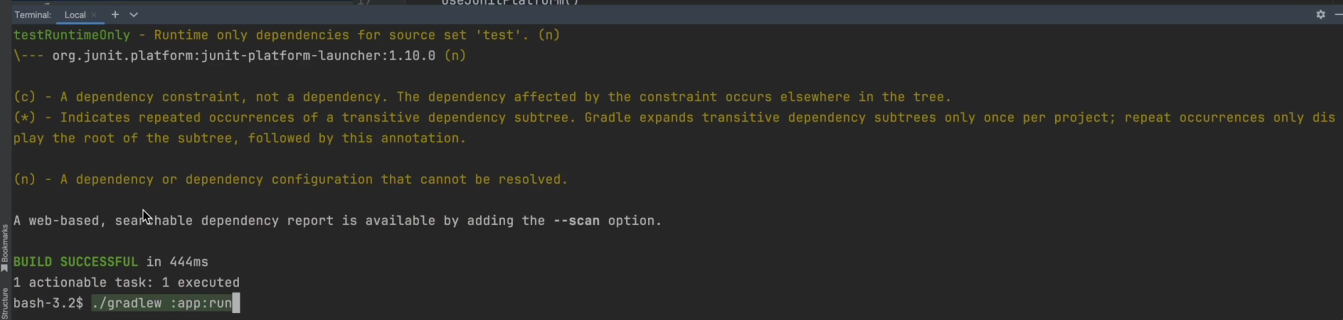


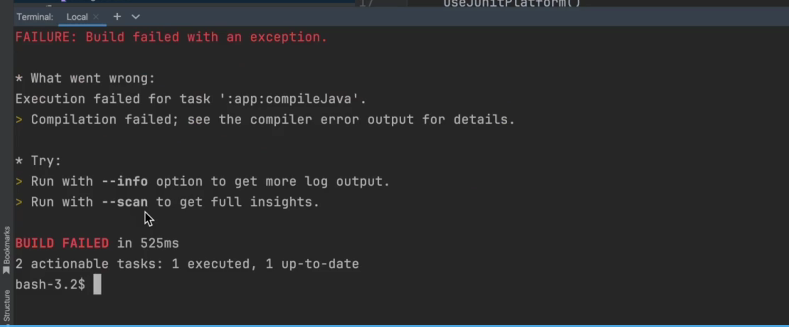
and run the dependencies task on the App sub project again, we can now see models transitive dependencies that is Guava and its dependencies only appear in the runtime classpath and not in the compiled classpath.



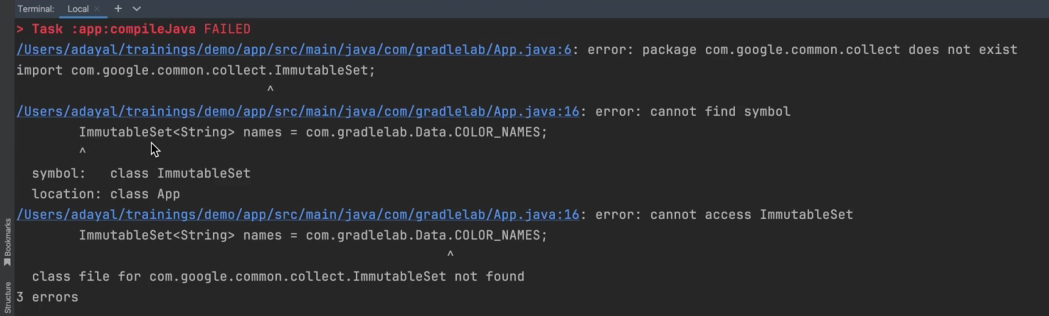


That means the Guava library is no longer available to the compiled Java task in the App sub project, and the code cannot see the classes available in the Guava library. Let’s try to run the App again.

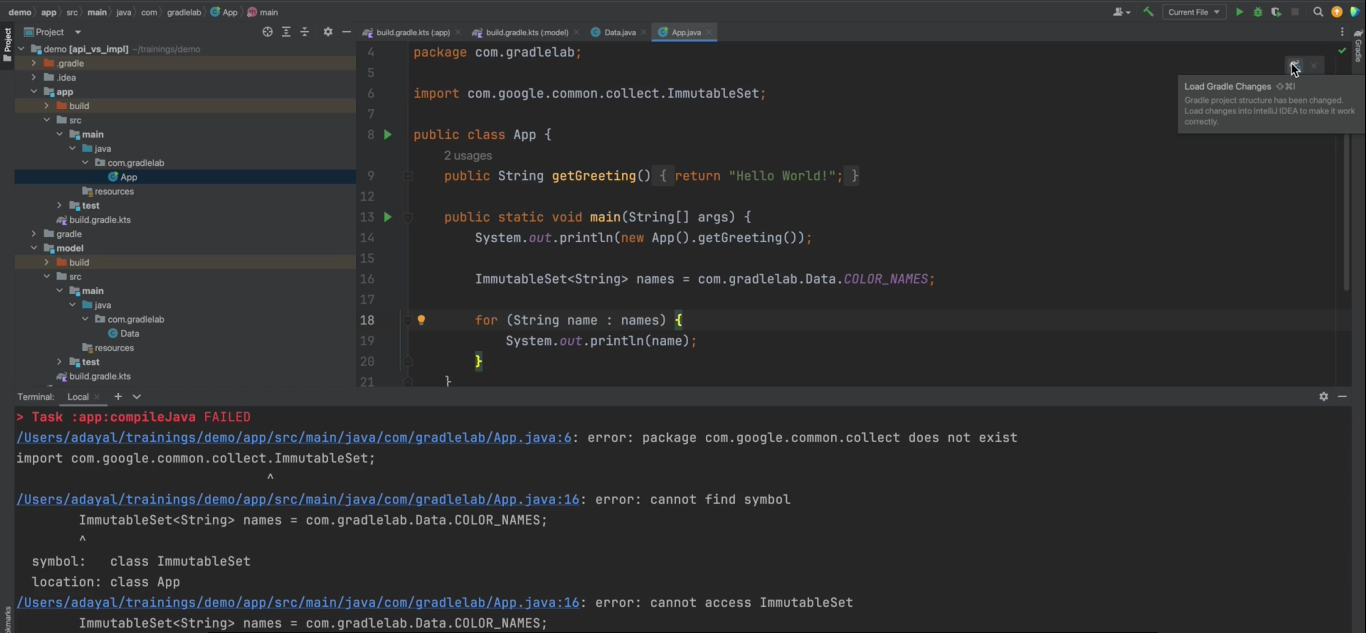




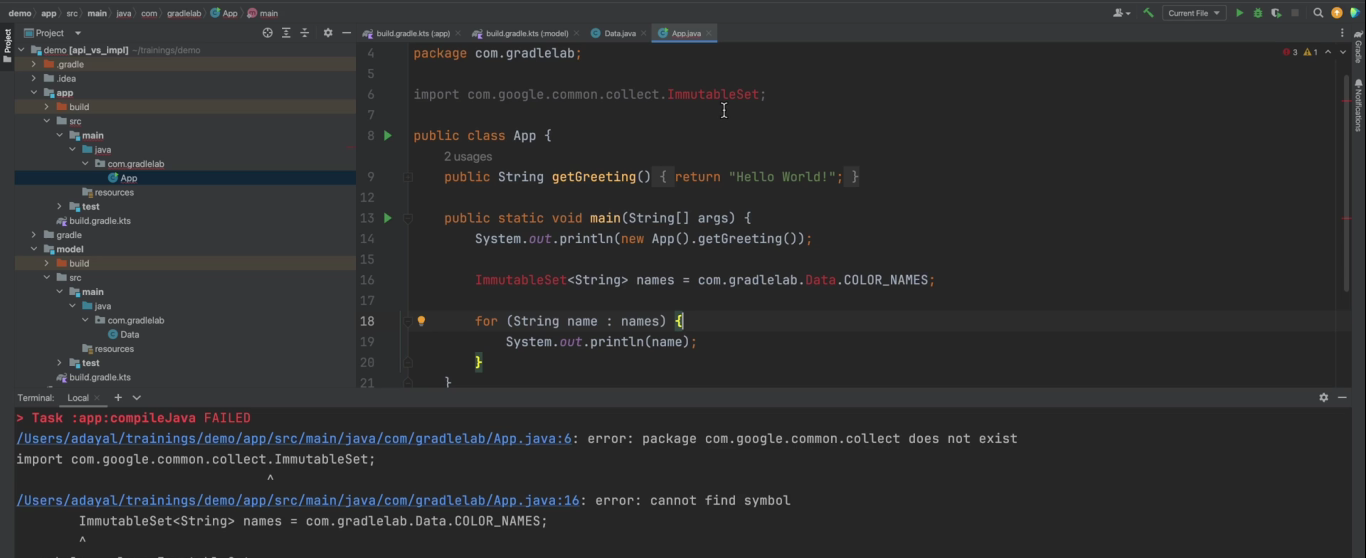
we can see compile failures. That indicate it could not find immutable set.



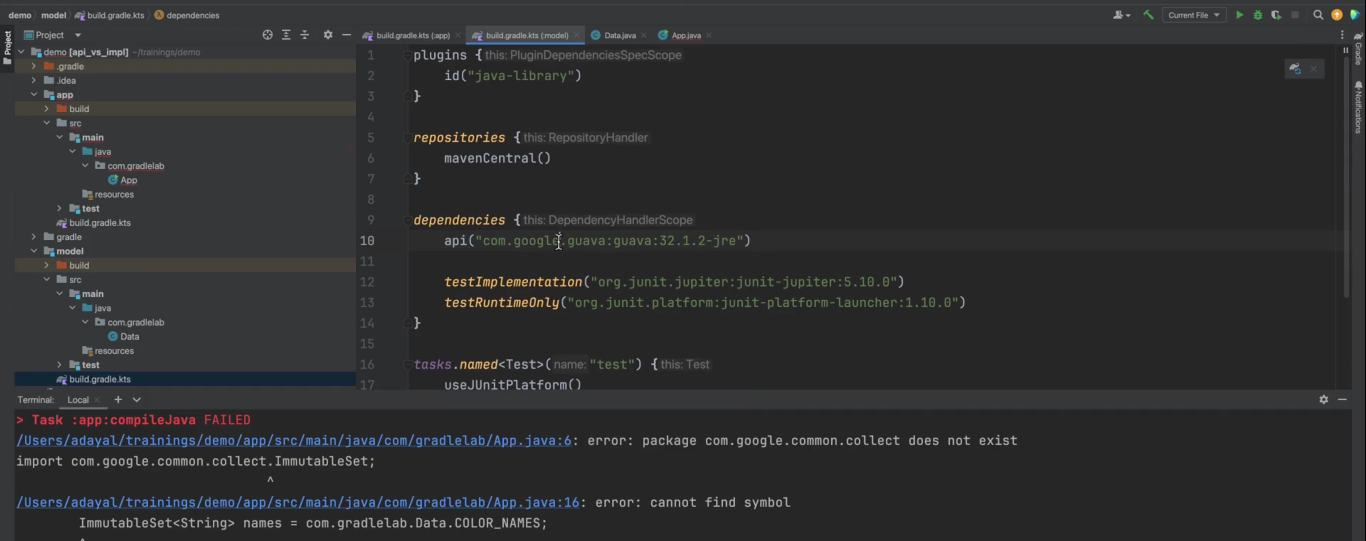
If we refresh the configs,



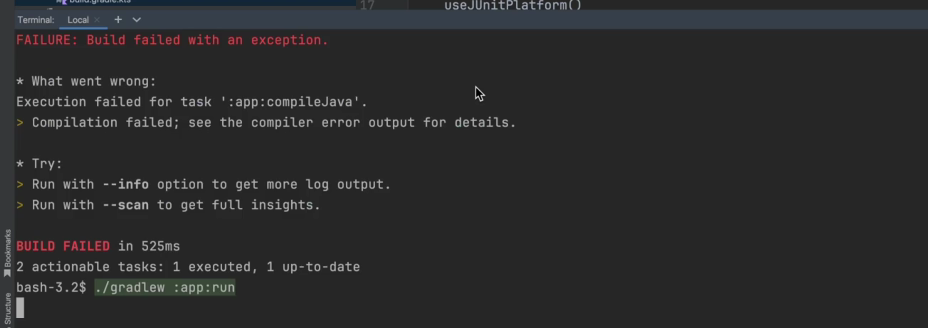
even the editor shows us that immutable set can no longer be recognized in the App sub project.

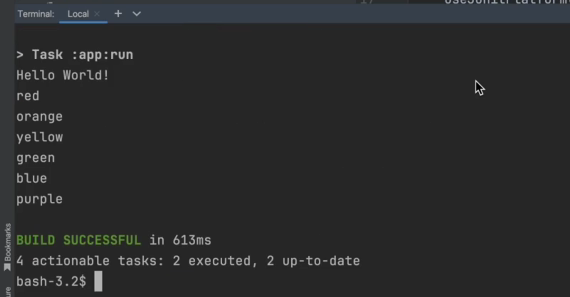


Let’s change Guava back to an API dependency in model.

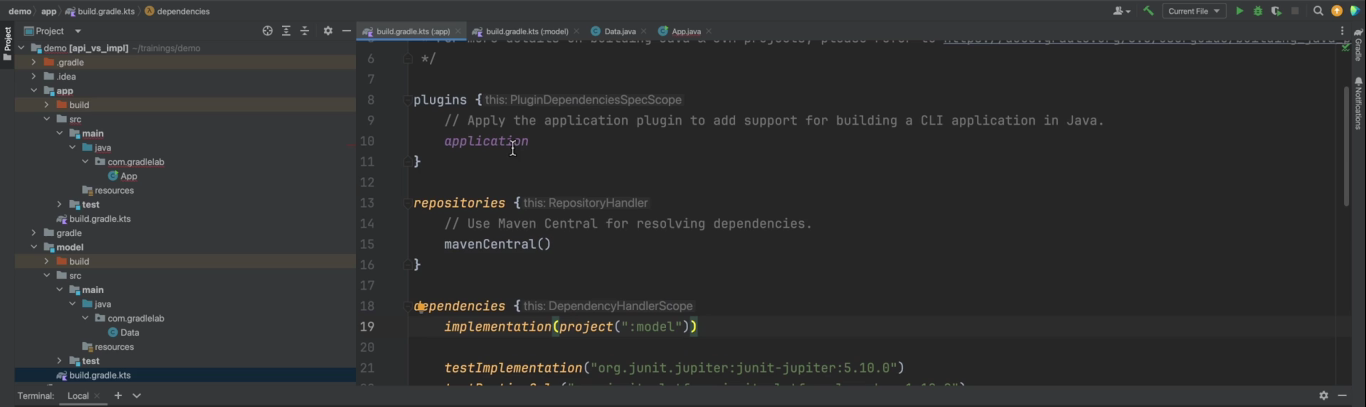


Now, if we run the App,

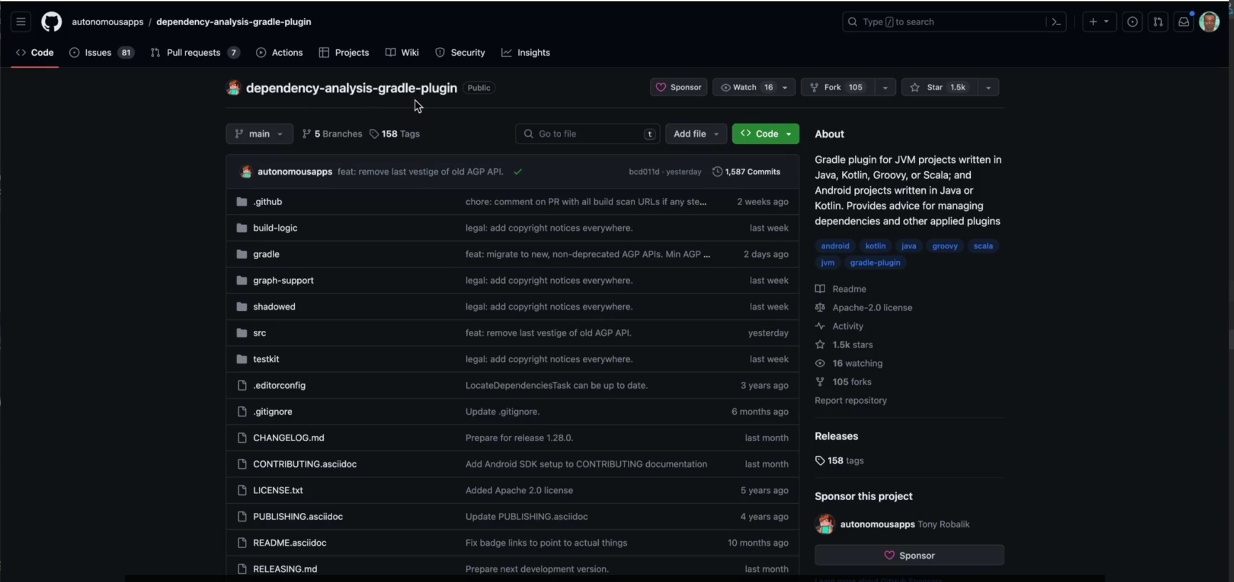




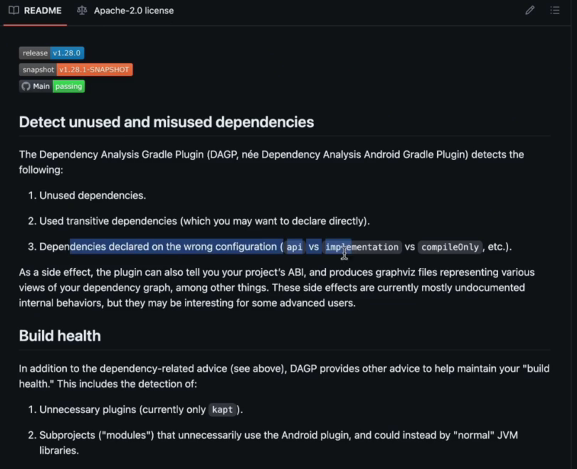
it works. Note that the API dependency configuration is only available in model. Because we are applying the Java Library plugin.



In App, we are applying the application plugin, so API is not gonna be available.



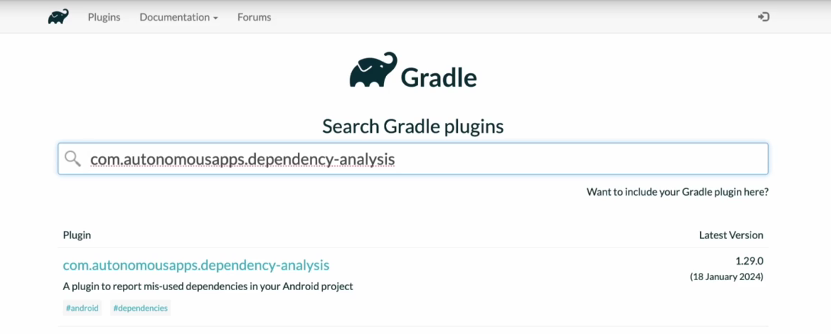
There’s a plugin called dependency analysis Gradle plugin for library authors that helps them determine whether dependencies need to be in the API or implementation bucket dependency configurations.



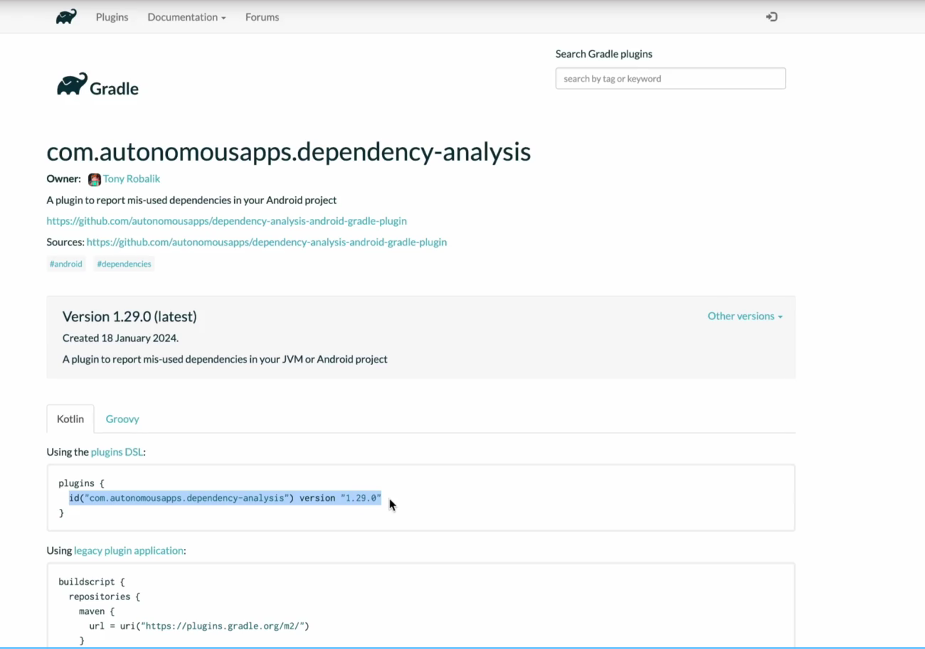
In the documentation, you can see the idea of the plugin.



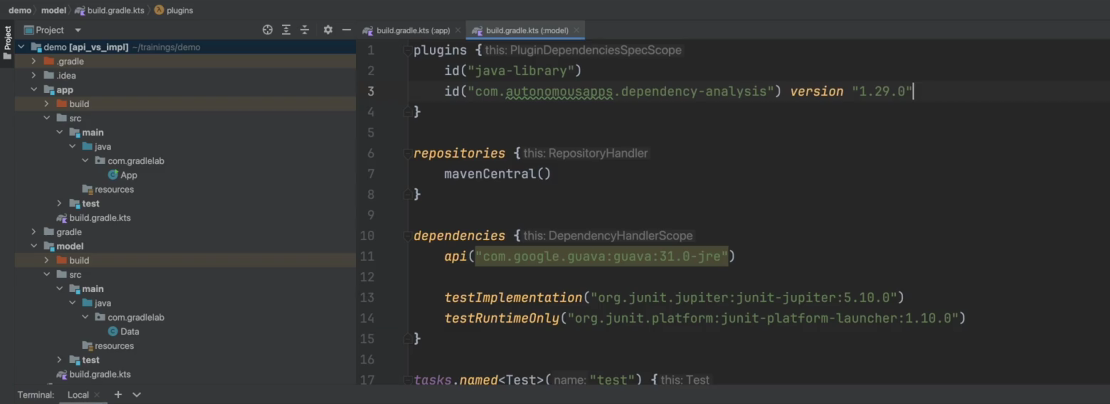
In the documentation, you can see the idea of the plugin. In the plugin portal, which is plugins.gradle.org. We can search for the latest version of the plugin. We can copy the string to apply the plugin for the Kotlin configuration.



We have two sub projects in this example.

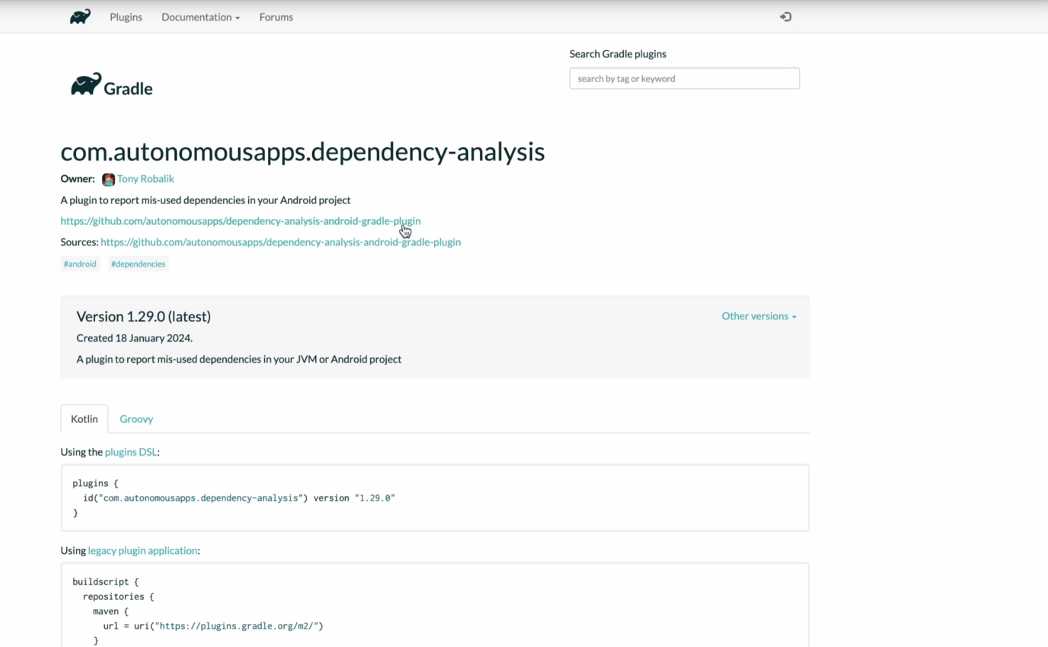


So should we add the plugin to both of them?

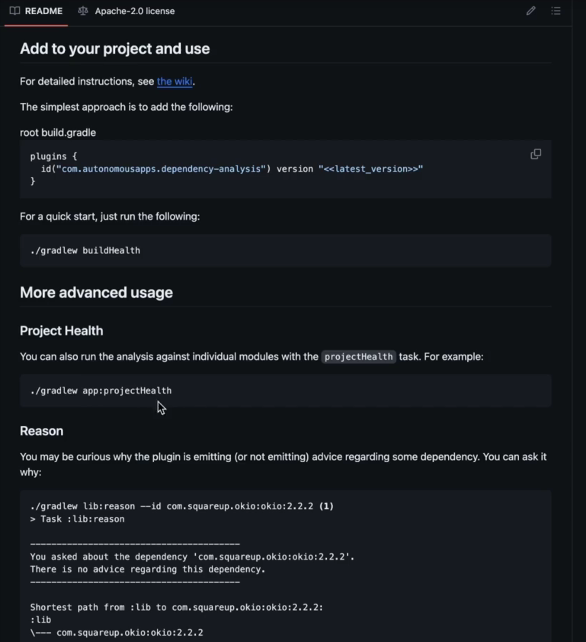


Let’s first try adding it to the model sub project.

If we look at the documentation again,

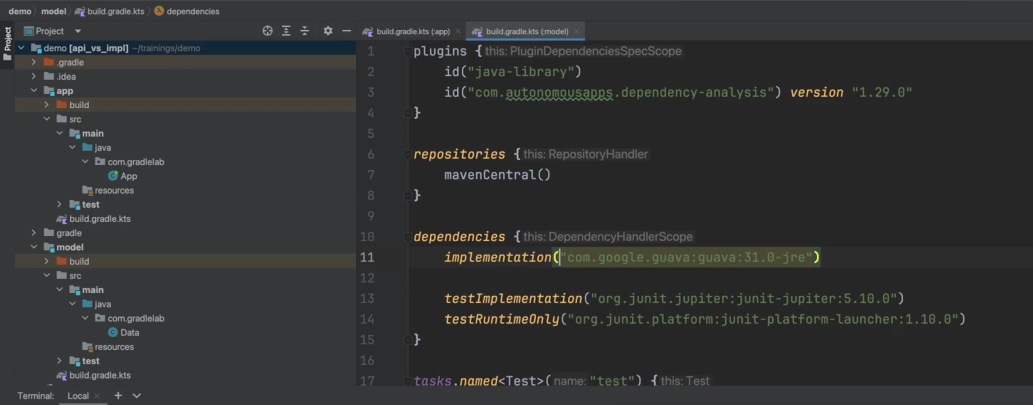


we can see there’s task

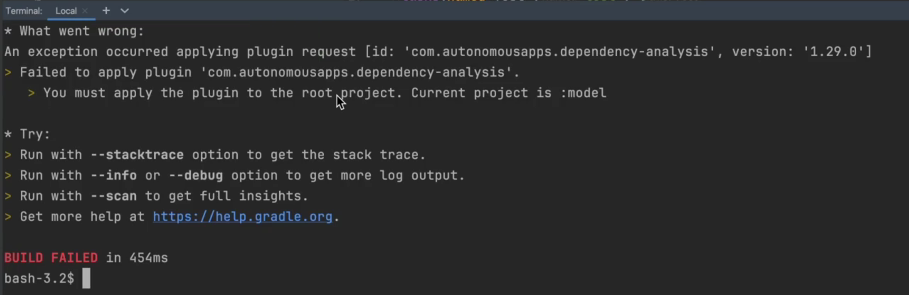


called Project Health that we can run. Let’s change the Guava dependency in the model sub project to the implementation bucket dependency configuration from the API bucket dependency configuration. The plugin should notify us that this is incorrect. Let’s run the task on the model sub project.

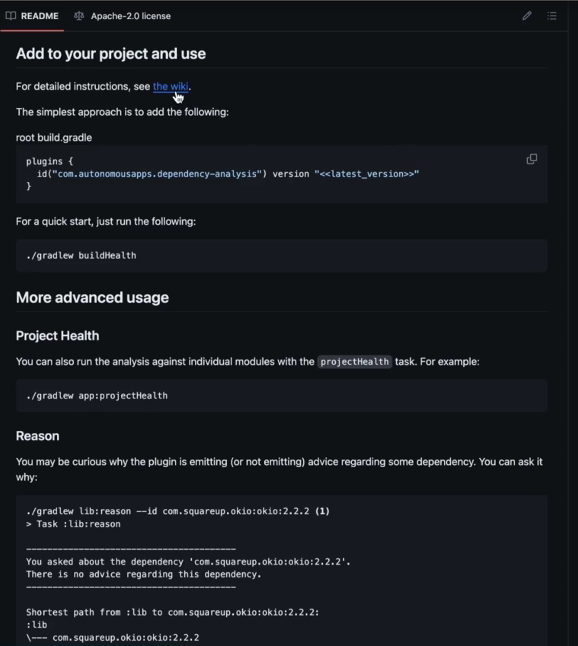
~~API~~ -> implementation



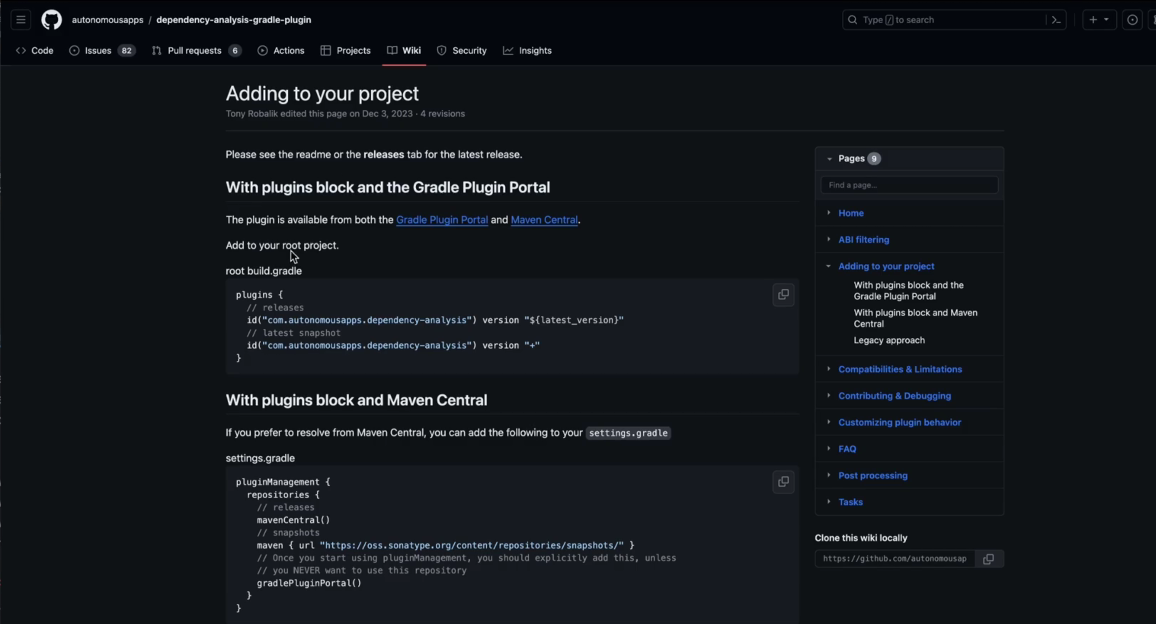




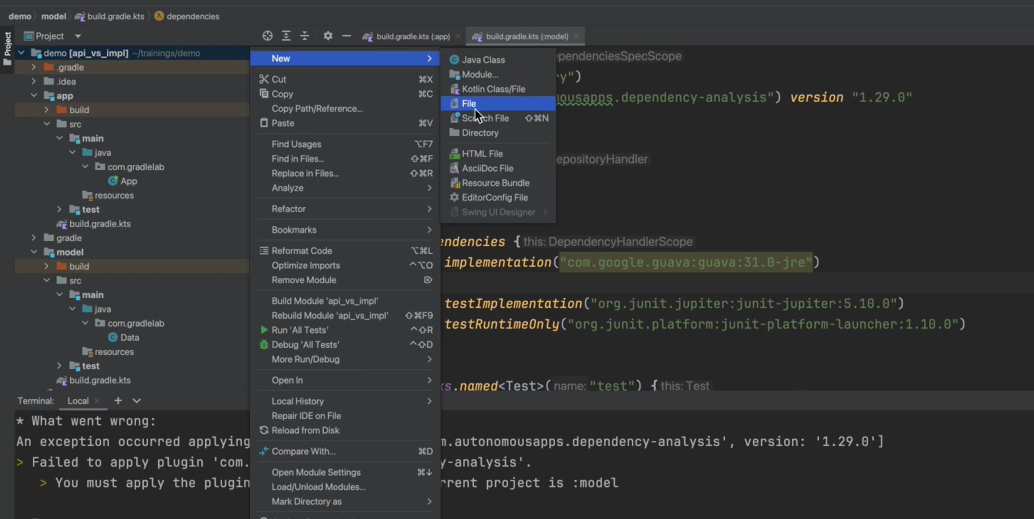
We see there’s an error message, which tells us that the plugin needs to be applied in the root project. If we go back to the documentation



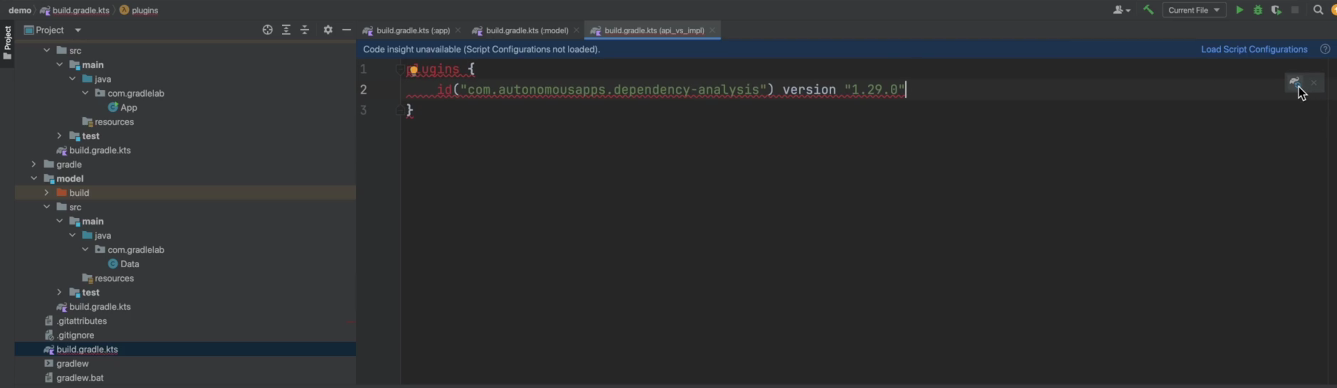
and click through the detailed instructions, we can see it documentation



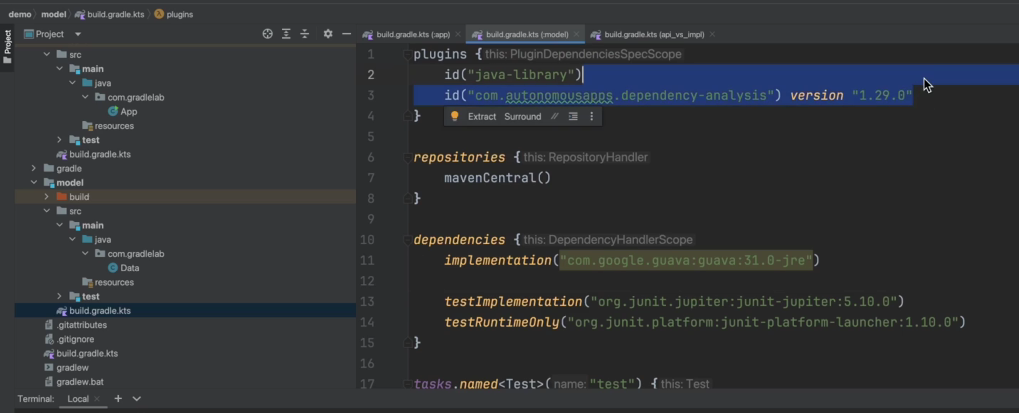
and click through the detailed instructions, we can see it mentions this as well. It’s worthwhile spending a few minutes reading through the documentation of a plugin before applying it. This is also an example of where we make use of the root project build file. Let’s create it. And apply the plugin.



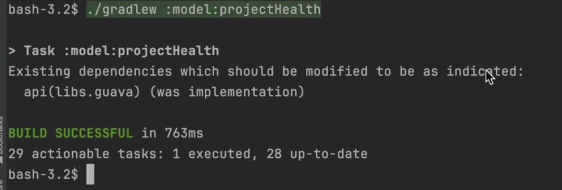




And remote it from the model sub project.



Now, if we run the task again,



we can see the plugin informed us that the Guava dependency should be in the API bucket dependency configuration, not in the implementation bucket dependency configuration.